Ein Kern für genetisches Programmieren in C++

Helmut Hörner

29. Mai 1996

Diplomarbeit, eingereicht an der Wirtschaftsuniversität Wien
April 1996

Inhaltsverzeichnis

1 Einführung **7**

2 Der einfache genetische Algorithmus **9**
2.1 Codierung der Problemparameter in Chromosomen 9
2.2 Die genetischen Operatoren 10
2.3 Varianten des einfachen genetischen Algorithmus 11

3 Vom einfachen genetischen Algorithmus zum genetischen Programmieren mit kontextfreien Grammatiken **14**
3.1 Kontextfreie Grammatiken und Ableitungsbäume 14
3.2 Genetisches Programmieren und genetische Operatoren für Ableitungsbäume 16

4 Die implementierten genetischen Operatoren **19**
4.1 Initialisierung 20
4.1.1 Suchraumberechnung 21
4.1.2 Initialisierung einer nach Ableitungstiefe der Individuen gleichverteilten Population 25
4.1.3 Initialisierung einer nach Ableitungstiefe und Form der Individuen gleichverteilten Population 26
4.2 Fitnessberechnung 29
4.2.1 Implementierte Berechnungsmethoden für die standardisierte Fitness 30
4.2.2 Implementierte Berechnungsmethoden für die adjustierte Fitness 31
4.2.3 Implementierte Berechnung der normalisierten Fitness 31
4.3 Implementierte Selektionsmethoden 31
4.3.1 Proportionale Selektion 32
4.3.2 Lineare Rangselektion 32
4.4 Implementierte Sampling-Verfahren 33
4.4.1 Stochastic Sampling With Replacement 33
4.4.2 Stochastic Universal Sampling 33
4.5 Implementierte Mating-Verfahren 34
4.5.1 Random Mating 35
4.5.2 Random Permutation Mating 35
4.6 Der implementierte Kreuzungsoperator 35
4.6.1 Selektionsheuristiken 36
4.7 Implementierter Mutationsoperator 37
4.8 Elitismus 37
4.9 Inkludieren von Unterableitungsbäumen in die Population 37

5 Dokumentation für den Anwender **38**
5.1 Die syntaktische Definition - die Sprachdefinitionsdatei 38
5.2 Die semantische Definition - der Sprachinterpreter 39

5.2.1 Interpretation des fertigen Strings (des Phänotyps) 39
5.2.2 In-Tree-Auswertung des Ableitungsbaums (des Genotyps) 41
5.3 Die Hauptschleife 44
5.3.1 Zur Verfügung stehende Klassen, Funktionen und Variablen . . 47
5.3.2 Initialisierung 51
5.3.3 Festlegen der genetischen Operatoren und die NextStep()-Methode 52
5.3.4 Bestimmen der Herkunft und der Eltern eines Individuums . . . 56
5.3.5 Coachen der Population 58
5.3.6 Speichern und Laden: Anwendung und Datenformat 60
5.3.7 Sonstige relevante Funktionen 63
5.3.7.1 Farm::Print() 63
5.3.7.2 StartNode::operator =() 64
5.3.7.3 StartNode::operator == 64
5.3.7.4 StartNode::PhenoTypeString() 64
5.3.7.5 StartNode::GenoTypeString() 64
5.3.7.6 StartNode::operator << 65
5.3.7.7 TreeNode::Depth() 65
5.3.7.8 TreeNode::NoOfSubTrees(...) 66
5.3.7.9 Language::Print() 66
5.3.7.10 Language::operator << 67
5.3.7.11 Language::PrintArbSearchSpace() 67
5.3.7.12 Language::PrintExactSearchSpace() 68

6 Programmdokumentation 70
6.1 Grammatikrepräsentation 70
6.1.1 Die Language-Klasse 73
6.1.1.1 Der Konstruktor Language() 73
6.1.1.2 Die Funktion SymbTabEntry* Symbol(char*) 74
6.1.1.3 Die Funktion Load(char*) 75
6.1.1.4 Die Funktion Print() 81
6.1.1.5 Der Operator << 83
6.1.1.6 Der Language()-Destruktor und die Funktion Clear() 85
6.1.2 Die SymbTabEntry-Klasse 85
6.1.2.1 Der Leerkonstruktor SymbTabEntry() 86
6.1.2.2 Konstruktor SymbTabEntry(char*, Language*) . . . 86
6.1.2.3 Die Funktion AddProdTabEntry() 87
6.1.2.4 Der Destruktor von SymbTabEntry() 88
6.1.3 Die ProdTabEntry-Klasse 89
6.1.3.1 Der Konstruktor ProdTabEntry() 89
6.1.3.2 AddDerivSymbol(SymbTabEntry* NewSymbol) 90
6.1.3.3 Der Destruktor von ProdTabEntry() und die Funktion Release() 91
6.1.4 Die Suchraumberechnung 92

6.1.4.1 Die Klasse SuperFloat 92
6.1.4.2 Suchraumberechnung in SymbTabEntry 101
6.1.4.3 Suchraumberechnung in ProdTabEntry 103
6.1.4.4 Language::ComputeArbSearchSpace(int) 105
6.1.4.5 Language::PrintArbSearchSpace() 106
6.1.5 Suchraumberechnung mit Genauigkeit bis zur letzten Stelle . . . 107
6.2 Ableitungsbäume 108
6.2.1 Die Funktionen der Klasse RootNode 110
6.2.1.1 RootNode() 110
6.2.1.2 RootNode(Language*, int, Population*) 111
6.2.1.3 operator =(StartNode&) 113
6.2.1.4 Set(char*) 114
6.2.1.5 CrossOver(CrossOverParameters) 115
6.2.2 Die Funktionen der Klasse StartNode 117
6.2.2.1 StartNode() 117
6.2.2.2 StartNode(Language*,TreeNode*,RootNode*) 118
6.2.2.3 operator =(StartNode&) 119
6.2.2.4 operator ==(StartNode&) 120
6.2.2.5 PrintParents() 121
6.2.2.6 GenoTypeString(int) 122
6.2.2.7 PhenoTypeString() 124
6.2.2.8 SubTree(unsigned int) 124
6.2.2.9 SubTree(SymbTabEntry, etc.) 127
6.2.2.10 AddCompleteSubTreesTo(Population*, etc.) 129
6.2.2.11 EnhanceWith(StartNode*) 130
6.2.2.12 ObjFuncVal() 131
6.2.3 Die Funktionen der Klasse TreeNode 131
6.2.3.1 TreeNode() 132
6.2.3.2 TreeNode(SymbTabEntry*, Language*, etc.) 132
6.2.3.3 Der Destruktor von TreeNode() 133
6.2.3.4 Print() 133
6.2.3.5 operator <<(ostream&, TreeNode&) 134
6.2.3.6 operator =(TreeNode&) 135
6.2.3.7 DerivationDepthUpdate() 136
6.2.3.8 AddCompleteSubTreesTo(Population*, etc.) 137
6.2.3.9 NoOfSubTrees(SymbTabEntry*, unsigned int) . . . 138
6.2.3.10 Depth() 139
6.2.3.11 CompareWith(TreeNode*) 140
6.2.3.12 BuildUpPhenoTypeStringIn(char*, int&) 140
6.2.3.13 PhenoTypeStringLength() 141
6.2.3.14 BuildUpGenoTypeStringIn(char*, int&, int) . . . 142
6.2.3.15 GenoTypeStringLength(int) 144
6.2.3.16 EnhanceWith(ProdTabEntry*) 146
6.2.3.17 RandomEnhance(unsigned int,unsigned int&) . . . 146

6.2.3.18 EnhanceWith(TreeNode*) 148
6.2.3.19 XEnhanceWith(TreeNode*, TreeNode*, etc.) 149
6.2.3.20 EnhanceWithString(char*, int&) 151
6.2.3.21 Evaluate(...) 153
6.3 Die Population 153
6.3.1 Die Datenstruktur 154
6.3.2 Erzeugung und Vernichtung der Population 155
6.3.2.1 Population() 155
6.3.2.2 Population(Language*, int, int) 156
6.3.2.3 GenerateVirtualPop() 157
6.3.2.4 GenerateVirtualPopIncSubTrees(...) 157
6.3.2.5 AddTreeToVirtualPop(StartNode*) 158
6.3.2.6 GenerateAlmostUniformActualPop(...) 159
6.3.2.7 Der Destruktor von Population() 164
6.3.3 Fitnessberechnungen 165
6.3.3.1 ApplyRawFitness() 165
6.3.3.2 ComputeBestAndWorstIndividual(int) 165
6.3.3.3 StandardizeWith(float, int) 167
6.3.3.4 InvertFitnessValues() 168
6.3.3.5 NormalizeFitnessValues() 168
6.3.4 Genetische Operatoren 169
6.3.4.1 PropSelect() 169
6.3.4.2 LinearRankSelect(float) 170
6.3.4.3 StochSamplWithRepl(int) 172
6.3.4.4 StochUnivSampl() 174
6.3.4.5 RandomMate() 175
6.3.4.6 RandomPermutationMate() 176
6.3.5 Hilfsfunktionen 177
6.3.5.1 ReSizeActualPop(int) 177
6.3.5.2 ReSizeVirtualPop(int,int) 178
6.3.5.3 ReSizeMatingList(int) 179
6.4 Die Farm 180
6.4.1 Die Datenstruktur 180
6.4.2 Erzeugung und Vernichtung der Farm 182
6.4.2.1 Farm() 182
6.4.2.2 Farm(int, int, int, char*, int) 183
6.4.2.3 Der Destruktor von Farm() 185
6.4.3 Speichern und Laden 185
6.4.3.1 Save(char*) 185
6.4.3.2 Farm(char*) 190
6.4.4 Auswahlfunktionen 201
6.4.4.1 SetFitnessScaling(unsigned int, int) 201
6.4.4.2 SetFitnessAdjustment(double (*) (double)) . . . 202
6.4.4.3 SetSelection(int) 203

6.4.4.4 SetSampling(int, int) 204
6.4.4.5 SetMating(int) 205
6.4.4.6 SetSelectionHeuristic(...) 205
6.4.4.7 SetCrossOver(int, float) 206
6.4.4.8 SetMutation(float) 207
6.4.4.9 SetSaveBest(int) 208
6.4.4.10 SetSmallestSubTreeOfVirtualPop(...) 208
6.4.4.11 SetMaxDerivDepth(int) 209
6.4.5 Fitnessberechnungen 210
6.4.5.1 StandardizeFitnessValues(int) 210
6.4.5.2 ApplyDynamicFitnessScaling(...) 211
6.4.5.3 AdjustFitnessValues() 213
6.4.5.4 ComputeFitnessValues() 213
6.4.6 Genetische Operatoren 214
6.4.6.1 NPointCrossOver(unsigned int, etc.) 214
6.4.6.2 SelectCrossOverPoints(StartNode*, etc.) 218
6.4.6.3 SelectCrossOverPointsUsingHeuristic(...) 220
6.4.6.4 Mutate(float, int) 226
6.4.6.5 SaveBestIndividual() 228
6.4.6.6 NextStep() 228
6.4.7 Hilfsfunktionen 230
6.4.7.1 Print() 230
6.4.7.2 SetDefaultMethods() 231
6.5 Zufallszahlengenerator 233
6.5.1 Uniran() 233
6.5.2 Uniran(int) 233
6.5.3 rand() 238
6.5.4 dice(int) 239
Literaturverzeichnis 240
Abbildungsverzeichnis 243
Tabellenverzeichnis 244

A Anhang 246
A.1 Listings 246
A.1.1 Die Headerdatei ga.h 246
A.1.2 Die Datei farm.cc 255
A.1.3 Die Datei tree.cc 298
A.1.4 Die Datei language.cc 320
A.1.5 Die Datei uniran.h 348
A.1.6 Die Datei uniran.cc 349

Zusammenfassung

In der vorliegenden Arbeit wird eine Einführung in einfache genetische Algorithmen über k-beschränkte kontextfreie Sprachen und in die wichtigsten genetischen Operatoren gegeben, und eine C++ Klassenbibliothek für genetisches Programmieren mit kontextfreien Grammatiken präsentiert. Das Programm ist flexibel gehalten, beinhaltet die wesentlichen genetischen Operatoren und kann jede in einer Datei in der *Backus-Naur-Form* vorliegende kontextfreie Grammatik interpretieren. Zudem wird auf die Problematik der Suchraumgrößenberechnung im Zusammenhang mit tiefenbegrenzten Ableitungsbäumen eingegangen.

Schlüsselworte: Genetisches Programmieren, Evolutionsstrategien, maschinelles Lernen.

This report gives a brief introduction in a variant of genetic programming (namely simple genetic algorithms over k-bounded context-free languages) and presents the most important genetic operators. A C++ class-library for genetic programming with context-free languages is presented within this report. The program is flexible and includes the most important genetic operators. It is able to interpret every context-free grammar in *Backus-Naur-Form* provided it is available in a file. In addition, this report deals with the problem of search-space-size calculation in connection with depth-bounded derivation trees.

Keywords: Genetic programming, evolutionary strategies, machine learning.

1 Einführung

Alles Leben, das heute auf der Erde zu finden ist, ist das Endprodukt eines Prozesses, der vor Jahrmillionen seinen Anfang genommen hat: Der Evolution. Und die Menschen, selbst auch nur ein Ergebnis dieser Evolution, betrachten staunend die Wunder, die diese hervorzubringen imstande war. Und obwohl die Techniker und Naturwissenschafter oft natürliche Vorbilder als Ausgangspunkt für ihre Maschinen heranzogen, ist die Maschine doch fast immer hinter dem natürlichen Vorbild zurückgeblieben. Keine Solarzelle arbeitet mit einem so hohen Wirkungsgrad wie die Photosynthese, kein U-Boot hat so einen geringen Strömungswiderstand in turbulenter Strömung wie der Delphin, kein Kunstfaden ist so elastisch wie der Faden einer Spinne... die Liste läßt sich wohl beliebig fortsetzen. Es ist daher kein Wunder, daß die Menschen versuchen, den Geheimnissen dieses Vorgangs, der derart perfekte Lösungen hervorgebracht hat, auf die Spur zu kommen. Und seit ein paar Jahrzehnten gibt es dazu ein Werkzeug, das dabei sehr hilfreich ist: Den Computer.

Schon in den frühen Sechzigerjahren begannen sich, unabhängig voneinander, John Holland und Ingo Rechenberg mit der Simulation der Evolution im Computer auseinanderzusetzen (siehe z.B. [Holland, 1973] und [Rechenberg, 1973]). Und obwohl die Rechenleistungen zu dieser Zeit noch sehr begrenzt waren, lieferten diese Simulationen bereits erstaunliche Ergebnisse. Seitdem hat sich die Rechenleistung und die Speicherkapazität von Computern um mehrere Zehnerpotenzen vergrößert. Auch unser Wissen über die Theorie der simulierten Evolution hat in der Zeit zugenommen, wenn auch nicht im gleichen Ausmaß. Ein relativ neuer Zweig auf dem Gebiet der Evolutionsstrategien ist das "genetische Programmieren" ("genetic programming"). Diesem Teilbereich ist die vorliegende Arbeit gewidmet. In ihr wird eine C++-Klassenbibliothek - ein Kern für genetisches Programmieren - vorgestellt, der es allen Interessierten ermöglicht, mit relativ geringem Programmieraufwand selber Simulationen in diesem Gebiet durchzuführen.

Dies kann einerseits von theoretischem Interesse sein, etwa um bestimmte Theorien in einer Simulation abzutesten, oder auch von praktischem Belang. Denn genetische Algorithmen gehören zu den stabilsten uns bekannten Optimierungsverfahren, die selbst in hochkomplexen, multidimensionalen Optimierungsproblemen immer noch gut konvergieren [Geyer-Schulz, 1992]. Gerade wenn für eine bestimmte Problemstellung kein exakter Lösungsweg bekannt ist, sondern nur mehr oder weniger gut funktionierende Heuristiken, dann bietet sich die Verwendung eines genetischen Algorithmus geradezu an.

Eine klassische Optimierungsaufgabe im Bereich der Wirtschaft ist z.B. die Maximierung des Gewinns, oder die Senkung der Lagerkosten. Die zur Verfügung stehenden Verfahren um diese Ziele zu erreichen, sind stets von der Natur einer Heuristik, und führen oft zu unbefriedigenden Ergebnissen. Ein Beispiel für die Anwendbarkeit von genetischen Algortihmen ist das MIT Beer Distribution Game [Sterman, 1989]. In diesem

am MIT ausgearbeiteten Managementspiel wird die Distributionskette vom Bierbrauer zum Konsumenten simuliert. Das System besteht aus einer Kette von vier Zwischenhändlern, die versuchen müssen in einem Umfeld von unsicherer Nachfrage, Zeitverzögerungen und Rückkopplungseffekten ihre Lagerkosten zu minimieren. Wegen der Komplexität des Systems (eine nichtlineare Differrenzengleichung dreiundzwanzigster Ordnung) ist eine exakte Lösung praktisch unmöglich. In [Geyer-Schulz, 1995] wird eindrucksvoll demonstriert, wie ein genetischer Algorithmus imstande ist, für diese Art von Problemen Lösungen zu liefern, die weitaus besser funktionieren als die bekannten Heuristiken.

Mit Hilfe des in dieser Arbeit vorgestellten Programms können ähnliche Berechnungen und Simulationen mit geringem Aufwand nachvollzogen werden. Dieses Programm - eine C++ - Klassenbibliothek für genetisches Programmieren - ermöglicht das einfache Erstellen von evolutionären Programmen im Bereich des genetischen Programmierens über k-beschränkte kontextfreie Sprachen. Um dem Leser eine entsprechende Einführung zu geben, wird zunächst im Kapitel 2 die Funktionsweise eines einfachen populationsbezogenen genetischen Algorithmus und die entsprechenden genetischen Operatoren erklärt.

Kapitel 3 erläutert den Übergang vom einfachen genetischen Algorithmus zum genetischen Programmieren mit kontextfreien Grammatiken. Der Zusammenhang zwischen Grammatikrepräsentation in Backus-Naur-Form, Ableitungsbaum und abgeleitetem Wort wird erklärt und die nötigen genetischen Operatoren werden umrissen.

Kapitel 4 geht sodann detailliert auf alle im vorliegenden Programm implementierten Operatoren aus theoretischer Sicht ein und erläutert ihre Funktionsweise.

Kapitel 5 beinhaltet alle für den Anwender der vorliegenden Klassenbibliothek relevanten Informationen. Die syntaktische und semantische Einbindung einer Grammatik, die Erzeugung einer Startpopulation, die Auswahl der gewünschten Operatoren, und alle sonstigen für den Anwender relevanten Funktionen werden in ihrer Anwendung erklärt. Details über die interne Funktionsweise des Programmes sind hierbei, soferne für den Anwender nicht unbedingt nötig, ausgespart.

Kapitel 6 wendet sich an Programmierer, die die interne Funktionsweise des Programms verstehen wollen, oder die vor haben, Erweiterungen zu implementieren. Alle Klassenfunktionen inklusive aller privater Funktionen sind hier aufgelistet und im Detail erklärt.

2 Der einfache genetische Algorithmus

Dieses Kapitel gibt eine kurze Einführung in die Funktionsweise einfacher genetischer Algorithmen. Die Funktionsweise eines einfachen populationsbezogener genetischer Algorithmus zur Funktionsoptimierung, wie er etwa in [Goldberg, 1989] vorgestellt wird, ist in Abbildung 1 dargestellt.

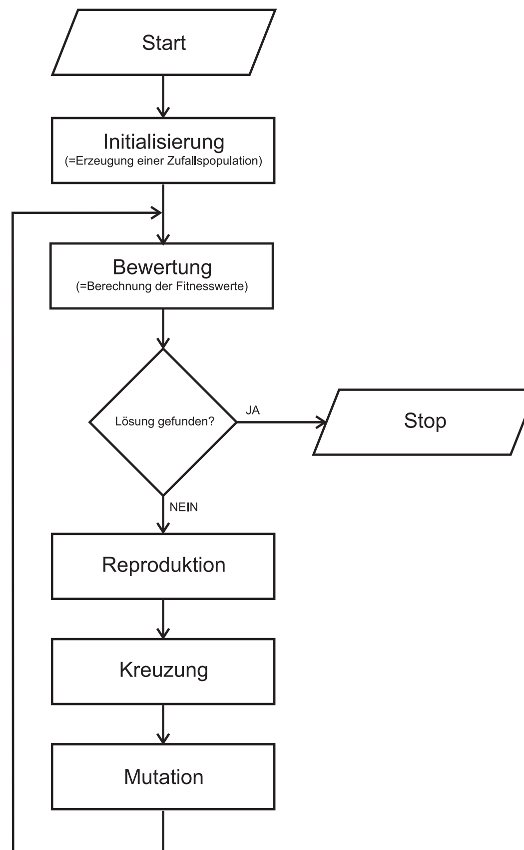


Abbildung 1: Funktionsweise des einfachen genetischen Algorithmus

2.1 Codierung der Problemparameter in Chromosomen

Einfache genetische Algorithmen arbeiten im allgemeinen mit Strings einer bestimmten Länge, die den in der Natur vorkommenden Chromosomen entsprechen. In jedem String können mehrere Parameter vercodiert sein. Zumeist bedient man sich bei einfachen genetische Algorithmen zur Codierung von numerischen Werten der Binärcodierung, aber auch Gray-Codierung wird empfohlen. Bei einer Funktionsoptimierungsaufgabe beispielsweise kann man n unabhängige Variable der Funktion im String als m-bit binärcodierte Zahlen aneinanderfügen. Man erhält dann einen String der Länge $n \times m$.

2.2 Die genetischen Operatoren

In einem einfachen genetischen Algorithmus gibt es zumindest die folgenden genetischen Operatoren (siehe Abbildung 1):

- Initialisierung
- Fitnessberechnung
- Reproduktion
- Kreuzung (Cross-Over)
- Mutation

Durch Anwendung dieser Operatoren erzeugt man zunächst eine Zufallspopulation, und sodann aus einer Generation von Strings die nächste Generation. Die Operatoren funktionieren folgendermaßen:

- **Initialisierung:**
  Ausgangspunkt des einfachen genetischen Algorithmus bildet eine Zufallspopulation von i Strings der Länge $n \times m$, die durch einen Initialisierungsoperator erzeugt werden müssen. Diese Strings bilden die sogenannten Genotypen, die eine bestimmte Problemlösung (z.B. n numerische Werte) repräsentieren.

- **Fitnessberechnung:**
  Jeder String wird zunächst auf seine Tauglichkeit in Bezug auf die Problemstellung gestestet. Das bedeutet, daß man den Genotypen (den String) in den Phänotypen (z.B. die Parameter der zu optimierenden Funktion) umwandelt, und danach diesem String einen sogenannten Fitnesswert zuordnet. Dieser Fitnesswert muß in irgendeiner Weise mit der Tauglichkeit des Individuums in Bezug auf das gestellte Problem korrelieren. Die Art der Skalierung und Berechnung sei an dieser Stelle noch dahingestellt. Dieser Problemkreis findet im Kapitel 4.2 nähere Beachtung.

- **Reproduktion:**
  An Hand dieser jedem Individuum der Population zuordenbaren Fitnesswerte werden nun in einem gewichteten Zufallsverfahren Individuen zur Reproduktion ausgewählt. Wichtig dabei ist, daß tauglichere Individuen mit höherer Wahrscheinlichkeit reproduziert werden, als weniger taugliche. Diese Vorgehensweise entspricht dem darwinistischen "survival of the fittest".

- **Kreuzung:**
  Die zur Reproduktion ausgewählten Individuen werden nun im sogenannten Kreuzungs-Verfahren ("Cross-Over") miteinander gekreuzt. Dieser Vorgang ist in Abbildung 2 dargestellt und geht folgendermaßen vor sich:

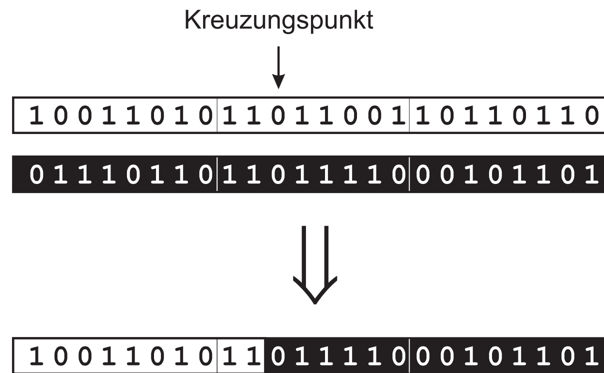


Abbildung 2: Kreuzen zweier Strings im einfachen genetischen Algorithmus

- In zwei zufällig gewählten Strings wird an der gleichen zufälligen Stelle ein Punkt markiert. Dies sei der Kreuzungsunkt.
- Aus diesen Strings wird nun ein neuer String erzeugt, der mit der Sequenz des ersten Strings beginnt, und am Kreuzungspunkt mit der Sequenz des zweiten Strings fortsetzt.

Auf die Codierung wird dabei beim einfachen genetischen Algorithmus keine Rücksicht genommen, was bedeutet, daß der Kreuzungspunkt auch mitten in einer m-bit-codierten Zahl liegen kann. In Abbildung 2 liegt der Kreuzungspunkt beispielsweise mitten in der zweiten 8-bit-codierten Zahl.

- **Mutation:**
  Die einzelnen Bits werden mit einer gewissen, geringen Wahrscheinlichkeit gekippt.

Obwohl dieses Verfahren reichlich primitiv anmutet, und obwohl die Zulässigkeit einer Kreuzung mitten in einer codierten Zahl a priori nicht vermuten läßt, daß der Kreuzungs-Operator zur Steigerung der Fitness der Individuen beiträgt, lehrt die aus der Simulation gewonnene Erfahrung etwas anderes.

In [Goldberg, 1989] wird erläutert, warum dies so ist. Es zeigt sich, daß der einfache genetische Algorithmus eine sogenannte implizite Parallelität aufweist, d.h. jede Selektion eines Individuums kann als Sample einer bestimmten Suchraumpartition aufgefaßt werden, die alle Schemata umfaßt, die zu dem gesampelten Individuum passen. Näheres dazu findet sich in [Holland, 1975] und in [Geyer-Schulz, 1994].

2.3 Varianten des einfachen genetischen Algorithmus

Es ist klar, daß die obige knappe Einführung bei weitem nicht alle Möglichkeiten selbst des einfachen genetischen Algorithmus abdeckt. Abgesehen davon gibt es inzwischen

unzählige Varianten des einfachen genetischen Algorithmus, deren bloße Aufzählung schon diesen Rahmen sprengen würde. Dennoch seien, um einen Eindruck von der Vielfalt der Möglichkeiten zu geben, an dieser Stelle ein paar Punkte aufgezählt:

- **Die Wahl der Parameter**
  Bei einem genetischen Algorithmus gilt es eine Menge Parameter zu wählen: Die Populationsgröße, die Codierung, die Fitness-Skalierung, die Anzahl der Kreuzungspunkte, die Mutationswahrscheinlichkeit, usw. Obwohl von verschiedener Seite Versuche gemacht wurden, Richtlinien für die Wahl dieser Parameter zu geben (etwa in [DeJong, 1975], [Goldberg, 1989] oder [Hoffmeister 1992]), sind die Ergebnisse doch sehr uneinheitlich. Andererseits kann man die Optimierung der Parameter des genetischen Algorithmus als eine Meta-Optimierungsaufgabe sehen, die selbst mit einem genetischen Algorithmus gelöst werden kann (siehe z.B. [Grefenstette, 1986]).

- **Die Codierung**
  Bei komplexeren Aufgabenstellungen reicht die Codierung in Form von m binärcodierten Zahlen oft nicht aus. So ist z.B. beim Traveling-Salesman-Problem eine Codierung nötig, die stets eine bestimmte Reihenfolge der zu besuchenden Städte beschreibt [Goldberg, 1989, S. 170-174]. Damit geht aber automatisch eine Änderung der genetischen Operatoren einher, da z.B. eine einfache Kreuzung im Traveling-Salesman-Problem mit hoher Wahrscheinlichkeit dazu führt, daß im neu erzeugten String bestimmte Städte gar nicht, und andere doppelt vorkommen. Dies bedeutet aber, daß die genetischen Operatoren problemspezifisch zu wählen sind, was der allgemeinen Verwendbarkeit und der Vergleichbarkeit der genetischen Algorithmen schadet.

- **Die genetischen Operatoren**
  Abgesehen davon, daß gewisse Codierungen gewisse Operatoren benötigen (z.B. der PMC-Kreuzungsoperator beim Traveling-Salesman-Problem [Goldberg, 1989, S. 170]), gilt es auch bei den Operatoren des einfachen genetischen Algorithmus aus einer Menge von Varianten zu wählen. Für die gewichtete Zufallsauswahl z.B. stehen unter anderem zur Verfügung:

  - Stochastic sampling with replacement
  - Stochastic sampling with partial replacement
  - Stochastic sampling without replacement
  - Stochastic universal sampling nach [Baker, 1987]

  Beim "stochastic sampling with replacement" beispielsweise werden m Individuen in m unabhängigen Zufallsprozessen ausgewählt, wobei eine Gewichtung an Hand der Fitness des Individuums erfolgt (bessere Individuen werden mit höherer Wahrscheinlichkeit ausgewählt als schlechtere). Diese Sampling-Methode kann

sehr gut mit Hilfe eines Rouletterades veranschaulicht werden, welches in n Sektoren unterschiedlicher Größe unterteilt ist (wobei n die Anzahl der Individuen in der Population darstellt). Beim "stochastic universal sampling" hingegen werden alle m Individuen in einem einzigen Schritt gezogen. Im Kapitel 4 sind alle im vorliegenden Programm verwendeten Varianten von genetischen Operatoren erläutert, unter anderem im Abschnitt 4.4 auch die beiden hier erwähnten Sampling-Verfahren.

- **variable Stringlänge**
  Um auch eine unterschiedliche Anzahl von Parametern oder eine komplexere Kodierung realisieren zu können, ist es nützlich die Restriktion einer fixen Stringlänge zu überwinden. In [Goldberg *et al*, 1990] wird z.B. ein genetischer Algorithmus mit positionsunabhängiger Codierung vorgestellt, der sowohl Überspezifikation, als auch Unterspezifikation zuläßt. Überspezifikation bedeutet, daß ein und dasselbe Gen mehrere Male im String vertreten ist. Die erste auftretende Version wird in der Evaluierung verwendet. Unterspezifikation bedeutet, daß manche Gene im String fehlen. Diese Variante ist schwieriger handzuhaben. Goldberg schlägt vor, die fehlenden Gene durch Gene zu ersetzem, die mittels lokaler Gradientensuche ermittelt werden.

- **Junk DNA**
  Untersuchungen von [Levenick 1991] haben ergeben, daß das Einbringen von bedeutungslosen Zufallssequenzen in den String (sogenannten Introns) die Performance des genetischen Algorithmus zu erhöhen scheint. Der Grund dafür dürfte darin liegen, daß sich die Anzahl der möglichen Kreuzungspunkte, die die einzelnen Gene unberührt lassen, erhöht.

- **usw.**

3 Vom einfachen genetischen Algorithmus zum genetischen Programmieren mit kontextfreien Grammatiken

In diesem Kapitel wird der Übergang vom einfachen genetischen Algorithmus zum genetischen Programmieren mit kontextfreien Grammatiken, wie es vom in dieser Arbeit vorgestellen Programm unterstützt wird, erläutert. Der Zusammenhang zwischen Grammatikrepräsentation in Backus-Naur-Form, Ableitungsbaum und abgeleitetem Wort wird erklärt und die nötigen genetischen Operatoren werden umrissen.

3.1 Kontextfreie Grammatiken und Ableitungsbäume

Abbildung 3 zeigt eine einfache kontextfreie Grammatik zur Erzeugung von Worten, die als boolsche Funktion interpretiert werden können. Diese Art der Darstellung wird Backus-Naur-Form (BNF) genannt [Duden Informatik, 1993, S. 52].

```
S := <fe> ;
<fe> := "(" <f0> ")" |
        "(" <f1> <fe> ")" |
        "(" <f2> <fe> <fe> ")" ;
<f0> := "D1" | "D2" ;
<f1> := "NOT" ;
<f2> := "OR" |"AND" ;
```

Abbildung 3: Die BNF der Grammatik L_{XOR}

Die in spitze Klammern gesetzten Symbole sind die sogenannten Nichtterminalsymbole der Grammatik, die unter Anführungszeichen gesetzten Symbole die Terminalsymbole. Durch Anwendung der in der Grammatik festgelegten Ableitungsregeln kann man nun Worte dieser Grammatik erzeugen. Mit Hilfe der BNF-Regeln in der Form

<*Nichtterminalsymbol*> `:=` *Ableitung* `|` *Ableitung* `|` ... *Ableitung* `;`

wird festgelegt, mit welchen Ableitungen ein bestimmtes Nichtterminalsymbol erweitert werden kann. Jede Ableitung besteht wiederum aus einer Folge von Terminal- und Nichtterminalsymbolen. Die Regel

`S :=` <*Startsymbol*>

legt fest, mit welchem Nichtterminalsymbol die Ableitung zu beginnen hat. Dieses Symbol ist das sogenannte Startsymbol. Das folgende Beispiel veranschaulicht, wie man aus obiger Grammatik das Wort `(NOT(OR(D1)(D2)))` herleiten kann:

- Zunächst schreibt man das Startsymbol an:
  <fe>

- Für <fe> setzt man nun die zweite mögliche Ableitung ein, wobei man die Anführungszeichen der Terminalsymbole weglassen kann:
  <fe> ⇒ (<f1><fe>)

- Für <f1> setzt man nun die einzig mögliche Ableitung NOT ein:
  (<f1><fe>) ⇒ (NOT<fe>)

- Das Nichtterminalsymbol <fe> wird nun mit der dritten möglichen Ableitung (<f2><fe><fe>) erweitert:
  (NOT<fe>) ⇒ (NOT(<f2><fe><fe>))

- Für das Nichtterminalsymbol <f2> setzt man die erste mögliche Ableitung, nämlich das Symbol OR ein:
  (NOT(<f2><fe><fe>)) ⇒ (NOT(OR<fe><fe>))

- Das erste <fe> wird mit der ersten der drei möglichen Ableitungen, nämlich mit (<f0>), erweitert:
  (NOT(OR<fe><fe>)) ⇒ (NOT(OR(<f0>)<fe>))

- Das Nichtterminalsymbol <f0> wird durch die erste der zwei möglichen Ableitungen, bestehend aus dem Symbol D1, ersetzt:
  (NOT(OR(<f0>)<fe>)) ⇒ (NOT(OR(D1)<fe>))

- Das noch nicht erweiterte Nichttermnialsymbol <fe> wird ebenfalls mit der ersten möglichen Ableitung (<f0>) erweitert:
  (NOT(OR(D1)<fe>)) ⇒ (NOT(OR(D1)(<f0>)))

- Und das Nichtterminalsymbol <f0> schließlich wird durch die zweite mögliche Ableitung, bestehend aus dem Symbol D2, ersetzt:
  (NOT(OR(D1)(<f0>))) ⇒ (NOT(OR(D1)(D2)))

Diese Herleitung des Wortes (NOT(OR(D1)(D2))) läßt sich auch in einem Ableitungsbaum wie in Abbildung 4 darstellen. In der reinen Backus-Naur-Form lassen sich nur kontextfreie Grammatiken darstellen. Der Baum in Abbildung 4 hat eine Ableitungstiefe von 8, was daran zu erkennen ist, daß er acht Nichtterminalsymbole beinhaltet. Kontextfrei bedeutet, daß ein Nichtterminalsymbol unabhängig von seinem Kontext, also unabhängig von den benachbart stehenden Zeichen ersetzt wird. Es ist also beispielsweise gleichgültig für die Erweiterung des letzten Nichtterminalsymbols <fe> der Ableitung (<f2><fe><fe>), mit welchen Ableitungen die vorangegangenen Nichtterminalsymbole <f2> und <fe> erweitert wurden. Dadurch lassen sich kontextfreie Grammatiken stets als Bäume darstellen [Duden Informatik, 1993, S. 352].

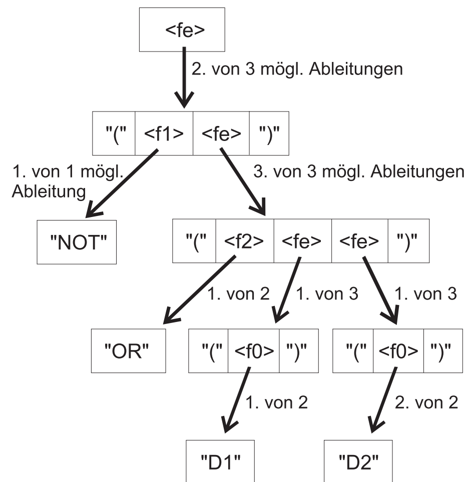


Abbildung 4: Der Ableitungsbaum für `(NOT(OR(D1)(D2)))`

3.2 Genetisches Programmieren und genetische Operatoren für Ableitungsbäume

Im Unterschied zu den einfachen genetischen Algorithmen besteht im vorliegenden Programm eine Population nicht aus Strings, sondern aus Ableitungsbäumen einer bestimmten Grammatik. Diese Modifikation stellt den Übergang vom einfachen genetischen Algorithmus zum genetischen Programmieren dar. Die Genotypen sind also nun nicht mehr einfache Strings, sondern komplette Ableitungsbäume. Und das aus jedem Ableitungsbaum der Population ablesbare Wort stellt den Phänotypen dar: Ein Programm. Der Ableitungsbaum in Abbildung 4 beispielsweise ist ein Genotyp, das Wort `(NOT(OR(D1)(D2)))` ist der Phänotyp dieses Baums: Ein kleines, interpretierbares Programm, welches zwei boolsche Variablen in einer ganz bestimmten Art und Weise auswertet. Die Verwendung von Ableitungsbäumen anstelle von Strings bedingt aber, daß man auf Bäume anwendbare genetische Operatoren verwendet. Insbesondere der Operator für die Zufallsinitialisierung, der Kreuzungs- und der Mutationsoperator müssen neu definiert werden:

- **Zufallsinitialisierung**: In den meisten Fällen wird eine zufällige Aneinanderreihung von Symbolen der Grammatik nicht zu einem syntaktisch richtigen Wort führen. Es muß daher, vom Startsymbol beginnend, bei jedem Individuum zufällig jedes Nichtterminalsymbol erweitert werden, bis ein kompletter Ableitungsbaum entstanden ist. Da bei den meisten Grammatiken (auch bei der in Abbildung 3 gezeigten) unendlich viele Worte erzeugt werden können, muß eine maximal zulässige Ableitungstiefe eingeführt werden, die nicht überschritten werden darf. Die Realisierung der Zufallsinitialisierung ist in Kapitel 4.1 beschrieben.

- **Kreuzung**: Es ist darauf zu achten, daß der durch den Kreuzungsoperator entstandene Ableitungsbaum wieder eine vollständige, korrekte Ableitung wieder-

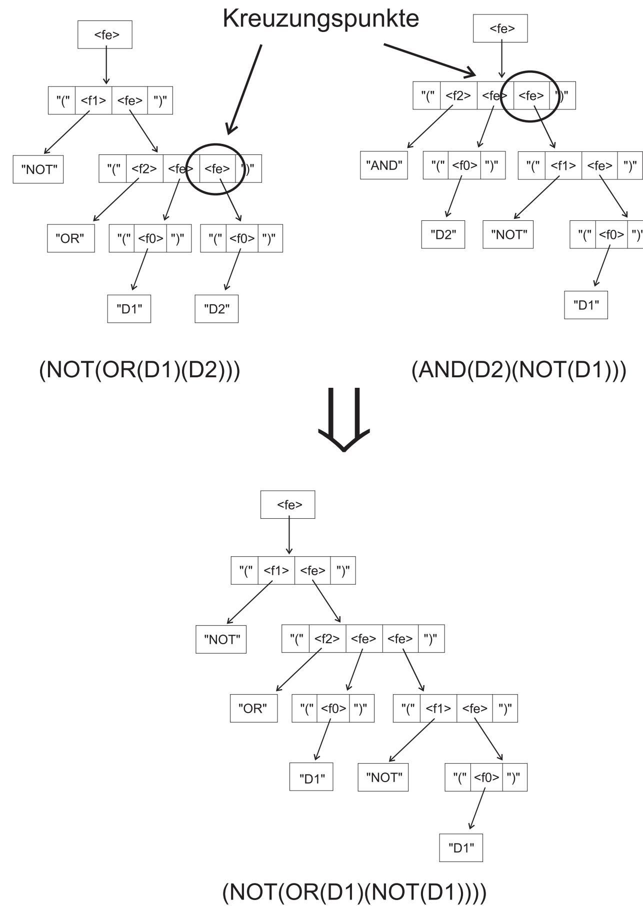


Abbildung 5: Der Kreuzungsoperator für Ableitungsbäume

gibt. Dies kann erreicht werden, indem man in den zu kreuzenden Bäumen jeweils zwei gleiche Nichtterminalsymbole heraussucht, und den so gefundenen Teilbaum des ersten Ableitungsbaumes durch den gefundenen Teilbaum des zweiten Ableitungsbaumes ersetzt. In Abbildung 5 wird an Hand eines Beispiels demonstriert, wie dies funktionert: In den beiden Ableitungsbäumen für die Worte (NOT(OR(D1)(D2))) und (AND(D2)(NOT(D1))) wird jeweils ein Knoten mit dem Symbol <fe> zufällig herausgesucht. Der erzeugte neue Baum im unteren Teil der Abbildung ist eine Kopie des ersten Baumes, mit dem Unterschied, daß der Teilbaum unterhalb des Kreuzungspunktes durch den Teilbaum des zweiten Ableitungsbaumes ersetzt wurde. Das resultierende Wort (NOT(OR(D1)(NOT(D1)))) ist wiederum syntaktisch korrekt. Im Kapitel 4.6 wird die Funktionsweise des implementierten Kreuzungsoperators detailliert erklärt.

- **Mutation** (siehe Kapitel 4.7): Ein Teilbaum des zu mutierenden Individuums, beginnend mit einem Nichtterminalsymbol, wird - ähnlich dem zuvor erläuterten Kreuzungsoperator - durch einen zufälligen Teilbaum, beginnend mit dem gleichen Nichtterminalsymbol, ausgetauscht. Dies stellt sicher, daß der derart modifizierte Baum wiederum eine korrekte Ableitung darstellt.

Im vorliegenden Programm kann die gewünschte Grammatik in ihrer BNF in einer Datei angegeben werden. Dadurch wird im Vergleich zum einfachen genetischen Algorithmus eine sehr viel größere Flexibilität erreicht. Jedes Problem, dessen Lösungen durch eine kontextfreie Grammatik beschrieben werden können, kann prinzipiell mit dem vorliegenden Programm bearbeitet werden, ohne daß man sich Gedanken um die Abbildung von Phänotypen in Genotypen machen müßte.

4 Die implementierten genetischen Operatoren

In diesem Kapitel werden die im vorliegenden Programm implementierten Operatoren zunächst kurz vorgestellt und ihre Funktionsweise anschließend genau erläutert. Durch die genetischen Operatoren werden nach der Erzeugung einer zufälligen Anfangspopulation in Generationsschritten immer neue Populationen erzeugt, bis schließlich das gewählte Abbruchkriterium erreicht ist. In einem Generationsschritt werden der Reihe nach folgende Operatoren aufgerufen (manche davon nur optional):

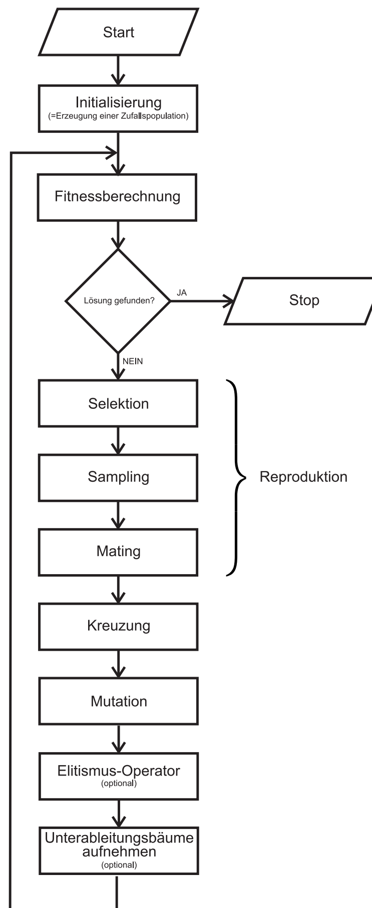


Abbildung 6: Die implementierten genetischen Operatoren

- **Fitnessberechnung:** Berechnung einer normalisierten Fitness für jedes Individuum.
- **Selektion:** Berechnung einer Auswahlwahrscheinlichkeit (*target sampling rate*, tsr) für jedes Individuum, die angibt, mit welcher Wahrscheinlichkeit das Individuum zur Selektion herangezogen wird.
- **Sampling:** Ziehen von n Individuen an Hand der Auswahlwahrscheinlichkeit (wobei n der Populationsgröße der nächsten Generation entspricht).

- **Mating:** Ermittlung eines Partners für jedes im Sampling gewählte Individuum.
- **Kreuzung:** Kreuzen der im Mating-Prozess einander zugeordneten Individuen mit einer gegebenen Wahrscheinlichkeit, wobei darauf zu achten ist, daß jede Kreuzung wiederum ein syntaktisch korrektes Wort liefert.
- **Mutation:** Mutation jedes der neuen Individuen (i.e.: Austausch eines zufällig gewählten Unterableitungsteilbaumes durch einen passenden, zufälligen Teilbaum mit einer gegebenen Wahrscheinlichkeit).
- **Elitismus-Operator:** Kopieren des besten Individuums der letzten Population in die neu erzeugte Population an eine zufällig gewählte Stelle (optional).
- **Aufnahme von kompletten Unterbäumen in die neue Population:** Unterbäume, welche selbst komplette Ableitungen darstellen und eine gewisse Größe n überschreiten, können in die neue Population aufgenommen werden (optional).

Im folgenden werden nun die einzelnen Operatoren detailliert erläutert.

4.1 Initialisierung

Bei der Initialisierung wird eine Zufallspopulation aus n Zufallsableitungsbäumen erzeugt (wobei n für die Größe dieser Population steht). Es ist nun aber für die Effizienz des Verfahrens von großer Bedeutung, wie diese Bäume erzeugt werden. Dies wird durch das folgende Beispiel offensichtlich.

Wie man aus der in Abbildung 3 gezeigten Notation erkennt, gibt es für jedes Nicht-Terminalsymbol eine gewisse Anzahl möglicher Ableitungen. Für `<f0>` beispielsweise kann entweder "`D1`" oder "`D2`" eingesetzt werden. Die einfachste Form der Erzeugung eines zufälligen Ableitungsbaumes besteht nun darin, zunächst mit dem Startsymbol `<fe>` zu beginnen und dieses um eine der drei möglichen Ableitungen zu erweitern. Dann wird solange jedes noch verbleibende Nichtterminalsymbol zufällig um eine seiner möglichen Ableitungen erweitert, bis ein vollständiger Ableitungsbaum entstanden ist, oder die maximale Ableitungstiefe erreicht wurde. Im letzeren Fall muß der unfertige Ableitungsbaum verworfen, und ein neuer Versuch gestartet werden. Abbildung 4 zeigt beispielhaft die Zufallserzeugung des Wortes `(NOT(OR(D1)(D2)))`. Man kann nun in einem Testlauf des Programmes feststellen, daß bei der gezeigten Art der Zufallserzeugung kürzere Worte mit weitaus höherer Wahrscheinlichkeit in der Ausgangspopulation vertreten sind, als längere Worte. Auch der naheliegende Gedanke, doppelte Individuen zu eliminieren, bringt keine tatsächliche Verbesserung, wie Tabelle 1 aus [Geyer-Schulz, 1994, S. 265] zeigt.

Begrenzt man nun die zulässige Ableitungstiefe, z.B. mit $i_{max} = 24$, dann kann man in einer Tabelle die Anzahl der möglichen Worte $card(L_{XOR}, i)$ (i.e.: die Suchraum-

Ableitungen	mit Duplikaten	ohne Duplikate
2	23	2
4	16	2
6	11	6
8	4	6
10	8	19
12	7	13
14	9	23
16	11	10
18	4	11
20	3	3
22	3	3
24	1	2

Tabelle 1: Anzahl von Wörtern in einer Testpopulation n=100

größe) für jede Ableitungstiefe bis $i = 24$ auflisten und jeder Ableitungstiefe i eine gewünschte Wahrscheinlichkeit $P(L_{XOR}, i, i_{max})$ zuordnen. In einer gleichverteilten Population müssen die Individuen entsprechend der ihnen zugeordneten Wahrscheinlichkeit $P(L_{XOR}, i, i_{max})$ vertreten sein.

Ein krasses Mißverhältnis zwischen Tabelle 2 und Tabelle 1 ist augenfällig. In einer Population mit $n = 100$ Individuen waren 23 Individuen der Größe $i = 2$ vertreten, und das, obwohl bei einer Wahrscheinlichkeit von $P(L_{XOR}, i, 24) = 0,0000006$ wohl kein einziges zu erwarten wäre.

4.1.1 Suchraumberechnung

Um dem eben beschriebenen Effekt entgegenzuwirken, ist es nötig, im Programm eine Suchraumberechnung zu implementieren, welche die Werte der obigen Tabelle ermittelt. In [Geyer-Schulz, 1994] wird der Weg zur Berechnung angegeben, und eine APL-Implementation vorgestellt, welche jedoch ein relativ ungünstiges Laufzeitverhalten aufweist. Im Folgenden soll nun versucht werden, die im vorliegenden Programm implementierte Berechnungsmethode zu veranschaulichen.

Wie bereits gezeigt wurde, gibt es zu jedem Nichtterminalsymbol eine bestimmte Anzahl von möglichen Ableitungen, welche ihrerseits wieder aus Symbolen zusammengesetzt sind. In folgendem Beispiel wird davon ausgegeangen, daß ein bestimmtes Nichtterminalsymbol $< \gamma >$ drei mögliche Ableitungen besitzt, nämlich X, Y und Z. Die Abbildung 7 zeigt das Nichtterminalsymbol $< \gamma >$, dessen Suchraum es bis zu $i = 4$ zu ermitteln gilt, sowie seine drei Ableitungen. Die bei den Ableitungen eingezeichneten

Ableitungen i	$card(L_{XOR}, i)$	$P(L_{XOR}, i, 24)$
2	2	0,0000006
4	2	0,0000006
6	10	0,0000032
8	26	0,0000082
10	114	0,0000360
12	402	0,0001269
14	1.722	0,0005435
16	6.890	0,0021745
18	29.794	0,0094032
20	126.626	0,0399640
22	556.778	0,1757227
24	2.446.138	0,7720167
Summe	3.168.504	1,0000000

Tabelle 2: Suchraumgrößen für L_{XOR}

Suchraumtabellen sind vorerst als gegeben zu betrachten, auf ihre Berechnung wird später eingegangen.

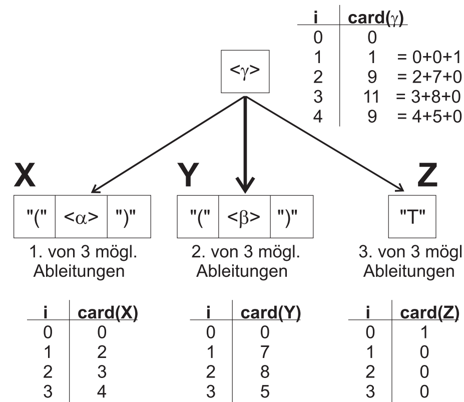


Abbildung 7: Das Nichtterminalsymbol $< \gamma >$ mit seinen drei möglichen Ableitungen X, Y und Z

Es ist unbedingt zu beachten, daß die Abbildung 7 **keinen Ableitungsbaum**, sondern einen **Baum möglicher Ableitungen** darstellt. Für jede der Ableitungen X, Y und Z ist im Bereich $0 \leq i \leq 3$ zu jedem i die entsprechende Anzahl möglicher Ableitungen $card(i)$ bekannt. Wenn z.B. bei der Ableitung X einem Wert von $i = 2$ der Wert $card(2) = 3$ zugeordnet ist, so bedeutet dies, daß von dieser Ableitung an - in zwei Ableitungsschritten - genau drei vollständige Ableitungsbäume erzeugt werden können. Ist $card(0) = 1$, wie dies bei Ableitung Z der Fall ist, so handelt es sich um ein

Terminalsymbol, da keine Ableitungsschritte mehr nötig sind. Die beim Symbol $< \gamma >$ eingezeichnete Tabelle gilt es zu ermitteln.

Die Lösung liegt auf der Hand: Für $i = 0$ muß $card(\gamma, 0) = 0$ sein, da es sich ja um kein Terminalsymbol handelt. Will man in einem Schritt einen vollständigen Ableitungsbaum erzeugen, so gibt es nur einen Weg: $< \gamma >$ um die Ableitung Z zu erweitern, die genau ein Terminalsymbol beinhaltet ($card(\gamma, 1)$ ist also gleich 1). Wieviele vollständige Ableitungsbäume können nun in $i = 2$ Schritten erzeugt werden? Macht man einen Ableitungsschritt von $< \gamma >$ zur Ableitung X, so steht noch ein weiterer Ableitungsschritt zur Verfügung, und in einem Schritt kann X genau zwei vollständige Bäume erzeugen. Wird $< \gamma >$ mit Y anstelle von X erweitert, so kann Y in dem verbleibenden einen Schritt genau sieben vollständige Bäume erzeugen. Ableitung Z kann, da es sich um ein Terminalsymbol handelt, lediglich in null Schritten einen Baum erzeugen. Daher ist $card(\gamma, 2) = 2 + 7 = 9$. Es gilt also im Rahmen des Beispiels:

$$card(\gamma, 0) = 0$$

und

$$card(\gamma, i) = card(X, i - 1) + card(Y, i - 1) + card(Z, i - 1)$$

Im allgemeinen Fall gilt:

$$card(\chi, i) = \sum_{j=1}^{m} card(deriv_j(\chi), i - 1)$$

wobei χ das Nichtterminalsymbol, dessen Suchraum es zu ermitteln gilt, ist, und $deriv_j(\chi)$ die j-te mögliche Ableitung von χ darstellt.

Im nun folgenden Beispiel wird das implementierte Verfahren der Suchraumberechnung für eine bestimmte Ableitung, welche aus mehreren Symbolen besteht, erläutert. In Abbildung 8 sieht man eine Ableitung, bestehend aus drei Nichtterminalsymbolen $< \alpha >$, $< \beta >$ und $< \gamma >$. Zu jedem dieser Symbole ist eine Tabelle $card(i)$ angegeben, die bei Wissen um die weiteren möglichen Ableitungen, wie zuvor gezeigt, leicht errechnet werden kann. Die Tabelle über der Klammer, welche $< \alpha >$ und $< \beta >$ verbindet, zeigt den Suchraum für die Ableitung $< \alpha >< \beta >$, und stellt ein Zwischenergebnis dar. Das gesuchte Endergebnis steht in der obersten Tabelle.

Im ersten Schritt werden nur die ersten beiden Symbole der Ableitung, nämlich $< \alpha >$ und $< \beta >$ betrachtet. Da beide Symbole keine Terminalsymbole sind, ist $card(\alpha\beta, 0) = 0$. In einem Ableitungsschritt kann nur entweder $< \alpha >$ (mit 3 möglichen Ableitungen) oder $< \beta >$ (mit 5 möglichen Ableitungen) erweitert werden, nicht aber beide gleichzeitig. Dazu sind zwei Schritte erforderlich, daher ist ebenfalls $card(\alpha\beta, 1) = 0$. Es sind also mindestens $i_{\alpha\beta} = 2$ Schritte erforderlich, um aus der

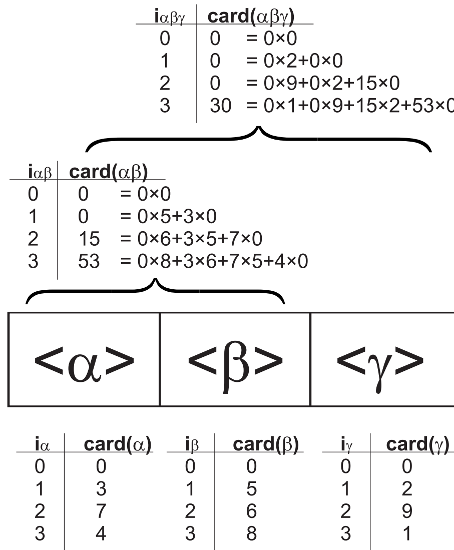


Abbildung 8: Die Ableitung $< \alpha >< \beta >< \gamma >$ und ihr Suchraum

Ableitung $< \alpha >< \beta >$ einen kompletten Ableitungsbaum zu erzeugen, und es gibt $3 \times 5 = 15$ verschiedene Kombinationen. Es gilt also: $card(\alpha\beta, 2) = 15$.

Überlegt man nun, welche Möglichkeiten es gibt, um in genau $i_{\alpha\beta} = 3$ Ableitungen zu einem kompletten Ableitungsbaum zu kommen, so sieht man, daß man entweder zwei Ableitungsschritte für $< \alpha >$, und einen Ableitungsschritt für $< \beta >$, oder einen Ableitungsschritt für $< \alpha >$, und zwei für $< \beta >$ "verbrauchen" kann. Im ersteren Fall gibt es $7 \times 5 = 35$, im letzteren Fall $3 \times 6 = 18$ Kombinationen. Ergibt zusammen: $card(\alpha\beta, 3) = 7 \times 5 + 3 \times 6 = 53$. Genau genommen müssen noch die Kombinationen $i_\alpha = 0$, $i_\beta = 3$ sowie $i_\alpha = 3$, $i_\beta = 0$ berücksichtigt werden:

$$
\begin{aligned}
card(\alpha\beta, 3) = \quad & card(\alpha, 0) \cdot card(\beta, 3) + \\
& card(\alpha, 1) \cdot card(\beta, 2) + \\
& card(\alpha, 2) \cdot card(\beta, 1) + \\
& card(\alpha, 3) \cdot card(\beta, 0)
\end{aligned}
$$

$$card(\alpha\beta, 3) = 0 \times 8 + 3 \times 6 + 7 \times 5 + 4 \times 0$$

$$card(\alpha\beta, 3) = 53$$

Die komplette Berechnung für den Vektor $card(\alpha\beta)$ ist in Tabelle 3 zusammengefaßt.

Der gesuchte Vektor $card(\alpha\beta\gamma)$ kann nun, wie in Tabelle 4 gezeigt wird, analog aus den bekannten Vektoren $card(\alpha\beta)$ und $card(\gamma)$ errechnet werden.

i	$i_\alpha + i_\beta$	$card$	$i_\alpha + i_\beta$	$card$	$i_\alpha + i_\beta$	$card$	$i_\alpha + i_\beta$	$card$	$\sum$
0	0+0	$0 \times 0 = 0$							0
1	0+1	$0 \times 5 = 0$	1+0	$3 \times 0 = 0$					0
2	0+2	$0 \times 6 = 0$	1+1	$3 \times 5 = 15$	2+0	$7 \times 0 = 0$			15
3	0+3	$0 \times 8 = 0$	1+2	$3 \times 6 = 18$	2+1	$7 \times 5 = 35$	3+0	$4 \times 0 = 0$	53

Tabelle 3: Suchraumberechnung für die Ableitung $< \alpha >< \beta >$

i	$i_{\alpha\beta} + i_\gamma$	$card$	$i_{\alpha\beta} + i_\gamma$	$card$	$i_{\alpha\beta} + i_\gamma$	$card$	$i_{\alpha\beta} + i_\gamma$	$card$	$\sum$
0	0+0	$0 \times 0 = 0$							0
1	0+1	$0 \times 2 = 0$	1+0	$0 \times 0 = 0$					0
2	0+2	$0 \times 9 = 0$	1+1	$0 \times 2 = 0$	2+0	$15 \times 0 = 0$			0
3	0+3	$0 \times 3 = 0$	1+2	$0 \times 9 = 0$	2+1	$15 \times 2 = 30$	3+0	$53 \times 0 = 0$	30

Tabelle 4: Suchraumberechnung für die Ableitung $< \alpha >< \beta >< \gamma >$

Auf die gezeigte Art und Weise können auch die Suchräume langer Ableitungen sehr effizient errechnet werden. In der vorliegenden Implementierung werden zudem einmal errechnete Vektoren für ein bestimmtes Symbol oder eine bestimmte Ableitung in einer Art Look-Up-Table gespeichert. Nähere Informationen dazu sind im Kapitel 6.1.4 enthalten.

4.1.2 Initialisierung einer nach Ableitungstiefe der Individuen gleichverteilten Population

Auf Wunsch kann nun im vorliegenden Programm die Ausgangspopulation nach einer von [Geyer-Schulz, 1994, S. 278ff] vorgeschlagenen Heuristik, welche eine in Bezug auf die Ableitungstiefe der Individuen gleichverteilte Population liefert, initialisiert werden:

1. Errechne eine Suchraumtabelle vom Startsymbol beginnend von $i = 0 \dots i_{max}$. Dieser Tabelle kann die Anzahl der zur Verfügung stehenden unterschiedlichen Individuen für jede Ableitungstiefe entnommen werden.

2. Errechne für jede Ableitungstiefe eine erwünschte Wahrscheinlichkeit $P(i)$, wie sie der Auftrittswahrscheinlichkeit in einer gleichverteilten Population entspricht (siehe Tabelle 2).

3. Ermittle mit der "stochastic universal sampling"-Methode (siehe Kapitel 4.4.2), eine Zielanzahl von Individuen jeder Ableitungstiefe i in der Population, wobei $P(i)$ als Wahrscheinlichkeitsgewichtung fungiert.

4. Erzeuge ein zufälliges Individuum und ermittle seine Ableitungstiefe.

5. Wenn die Zielanzahl für ein Individuum dieser Ableitungstiefe i größer Null ist, dann füge es der Population hinzu, soferne es nicht schon ein identes Individuum in der Population gibt, und reduziere die Zielanzahl und die Anzahl der zur Verfügung stehenden Individuen der Größe i um eins.

6. Wenn das Individuum nicht hinzugefügt wurde, weil es bereits ein identes Individuum in der Population gibt, und die Anzahl der zur Verfügung stehenden Individuen der Größe i gleich Null ist, dann füge es dennoch hinzu, und reduziere die Zielanzahl von Individuen der Größe i um eins (In diesem Fall sind nämlich mehr Individuen dieser Größe i zu erzeugen, als es überhaupt unterschiedliche gibt, Duplikate also unvermeidlich).

7. Fahre fort mit Schritt 4, bis die gewünschte Anzahl von Individuen der Population hinzugefügt wurde.

Es ist jedoch zu beachten, daß diese Heuristik, trotz der Elimination von Duplikaten, weit davon entfernt ist, eine Gleichverteilung nach Form der erzeugten Individuen zu gewährleisten.

4.1.3 Initialisierung einer nach Ableitungstiefe und Form der Individuen gleichverteilten Population

In [Geyer-Schulz und Böhm, 1996] wird ein Verfahren zur Initialisierung einer sowohl nach Ableitungstiefe, als auch nach Form der Individuen gleichverteilten Population vorgestellt. Im Rahmen des vorliegenden Programmes ist eine derartige Initialisierung im Moment noch nicht realisiert, jedoch ist eine Implementierung für die nächste Programmversion geplant. Ein Algorithmus zur Erzeugung einer gleichverteilten Population könnte etwa folgendermaßen aussehen:

1. Errechne eine Suchraumtabelle vom Startsymbol beginnend von $i = 0 \ldots i_{max}$. Dieser Tabelle kann die Anzahl der zur Verfügung stehenden unterschiedlichen Individuen für jede Ableitungstiefe entnommen werden.

2. Errechne für jede Ableitungstiefe eine erwünschte Wahrscheinlichkeit $P(i)$, wie sie der Auftrittswahrscheinlichkeit in einer gleichverteilten Population entspricht (siehe Tabelle 2).

3. Ermittle mit der "stochastic universal sampling"-Methode (siehe Kapitel 4.4.2), eine Zielanzahl von Individuen jeder Ableitungstiefe i in der Population, wobei $P(i)$ als Wahrscheinlichkeitsgewichtung fungiert.

4. Setze den Zähler $j = 1$

5. Erzeuge ein Startsymbol.
6. Gib dem Startsymbol die Anweisung “Erzeuge einen Baum der Tiefe $i = Zj$“, wobei Z_j der j-te Eintrag des in Punkt 3 errechnete Vektors von Zielgrößen entspricht.
7. Das “zur Erweiterung aufgeforderte“ Symbol “erfragt“ von jeder der möglichen Ableitungen, wieviele komplette Bäume sie in $i-1$ Schritten erzeugen kann, und gewichtet die Zufallsauswahl zwischen den Ableitungen nach dieser Anzahl.
8. Die ausgewählte Ableitung wird hinzugefügt und “erhält die Anweisung“ sich zu einen kompletten Baum der Größe $i-1$ zu erweitern.
9. Die Ableitung wählt eine der möglichen Erweiterungskombinationen zur Erzeugung eines Baumes der Größe $i-1$ nach der weiter unten erklärten Gewichtungsmethode aus, und gibt jedem Symbol die Anweisung, einen Baum der entsprechenden Größe zu erzeugen (wird weiter unten im Beispiel näher erläutert).
10. Fortfahren mit Schritt 7, bis der Baum vollständig ist.
11. Erhöhe j um eins.
12. Fortfahren mit Schritt 5, bis die Population voll ist.

Diese relativ knappe Beschreibung des Verfahrens soll nun an Hand eines kleinen Beispiels erhellt werden:

Abbildung 9 zeigt ein Nichtterminalsymbol $<\gamma>$ mit seinen drei möglichen Ableitungen X, Y und Z. Zu jeder Ableitung ist die bereits bekannte Suchraumtabelle bis $i = 3$ eingezeichnet. Das Symbol $<\gamma>$ bekommt nun die Anweisung: “Erzeuge einen Baum der Größe $i = 3$“. Im nächsten Schritt fragt das Symbol $<\gamma>$ bei seinen drei möglichen Ableitungen X, Y und Z nach, wieviele vollständige Bäume sie in genau $3-1 = 2$ Schritten erzeugen können. Jede Ableitung “sieht in der Suchraumtabelle nach“, und gibt die Antwort zurück an das Symbol $<\gamma>$. Im diesem Beispiel kann Ableitung X in $i = 2$ Schritten genau drei vollständige Bäume, Ableitung Y in $i = 2$ Schritten genau acht vollständige Bäume und Ableitung Z in $i = 2$ Schritten genau null vollständige Bäume erzeugen (Z beinhaltet ein einzelnes Terminalsymbol, und kann daher lediglich in $i = 0$ Schritten genau einen Baum erzeugen). Das Symbol $<\gamma>$ muß also seine Zufallsauswahl zwischen X, Y und Z also 3:8:0 gewichten, um keine der möglichen Ableitungsbäume zu bevorzugen. Die gewählte Ableitung wird dem Baum hinzugefügt, und erhält die Anweisung, einen Baum der Größe $i = 2$ zu erzeugen.

Die Frage, wie eine Ableitung eine Anweisung der Art “Erzeuge einen Baum der Größe $i = \ldots$“ zu bearbeiten hat, wird im folgenden Beispiel veranschaulicht:

In Abbildung 10 sieht man eine Ableitung $<\alpha><\beta> T$, welche die Anweisung erhält, einen Baum der Größe $i = 3$ zu erzeugen. Über der Ableitung stehen die Suchraumtabellen für die in der Ableitung enthaltenen Symbole $<\alpha>$, $<\beta>$ und “T“.

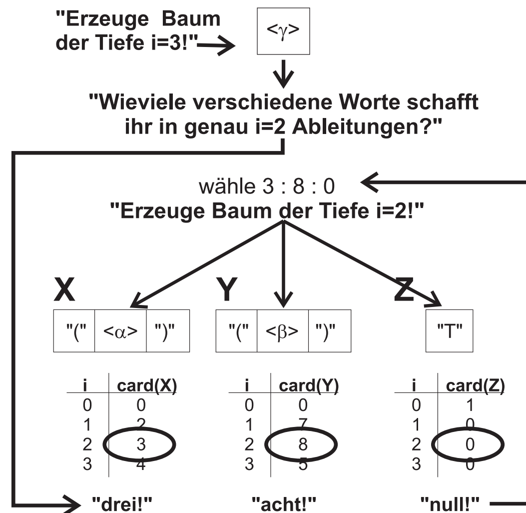


Abbildung 9: Symbol $< \gamma >$ bekommt die Anweisung, einen Baum der Tiefe $i = 3$ zu erzeugen.

Die große Tabelle im linken unteren Teil von Abbildung 10 listet alle theoretischen Kombinationen zur Erzeugung eines kompletten Baumes der Größe $i = 3$ auf:

- **Kombination A:** Null Ableitungen für $< \alpha >$, null Ableitungen für $< \beta >$ und drei Ableitungen für "T".
- **Kombination B:** Null Ableitungen für $< \alpha >$, eine Ableitung für $< \beta >$ und zwei Ableitungen für "T".
- **Kombination C:** Null Ableitungen für $< \alpha >$, zwei Ableitung für $< \beta >$ und eine Ableitung für "T".
- etc.

Daneben eingetragen ist die Anzahl der vollständigen Bäume, die tatsächlich mit der jeweiligen Kombination erzeugt werden kann. Bis auf zwei Kombinationen (G und I) enthält diese Spalte lauter Nullen. Dies hat seinen Grund darin, daß das Terminalsymbol in $i \neq 0$ Schritten keinen Baum, und die Nichtterminalsymbole in $i = 0$ Schritten keinen Baum erzeugen können. Somit ergeben alle Kombinationen, die $i_\alpha = 0$ oder $i_\beta = 0$ oder $i_T \neq 0$ enthalten, null. Es gibt also zwei Kombinationen (G und I), welche in genau $i = 3$ Schritten einen vollständigen Ableitungsbaum erzeugen können. Kombination G kann in $i = 3$ Schritten genau 18, Kombination I in $i = 3$ Schritten genau 35 verschiedene Bäume erzeugen. Die Gewichtung zur Wahl zwischen Kombination G und Kombination I muß also lauten:

$$G : I = 18 : 35$$

oder:

$$G : I = 0,34 : 0,66$$

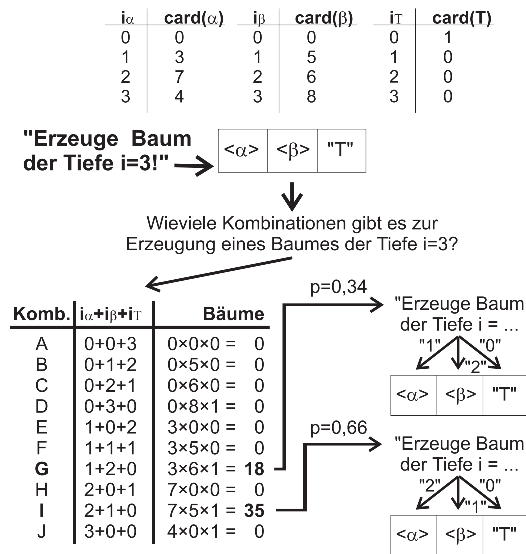


Abbildung 10: Ableitung $<\alpha><\beta>$“T“ bekommt die Anweisung, einen Baum der Tiefe $i = 3$ zu erzeugen.

4.2 Fitnessberechnung

Wesentlicher Bestandteil und Ausgangspunkt jedes genetischen Algorithmus ist die Berechnung einer **Rohfitness** für jedes einzelne Individuum der Population. Diese Rohfitness muß etwas über die Güte des Individuums in Bezug auf die Aufgabenstellung aussagen. Im Falle einer Aufgabe mit dem Ziel einer Kurvenanpassung könnte als Rohfitness die mittlere Abweichung der durch das Individuum beschriebenen Kurve zur anzupassenden Kurve herangezogen werden. Dieser solchermaßen ermittelte Rohfitnesswert wäre dann zu minimieren. Genausogut könnte aber auch die mittlere quadratische Abweichung, oder der Kehrwert der mittleren Abweichung herangezogen werden. Im letzteren Fall wäre die Rohfitness nicht zu minimieren, sondern zu maximieren.

Aus diesem einfachen Beispiel ist ersichtlich, daß eine gewisse Normierung der Fitnesswerte zur Abstraktion von einer ganz bestimmten Aufgabenstellung nötig ist. Ziel der Fitnessberechnung ist die Ermittlung einer **Auswahlwahrscheinlichkeit** (*target sampling rate*, tsr) für jedes einzelne Individuum der Population, welche angibt, ob das einzelne Indivduum über- oder unterdurchschnittlich selektiert wird.

Im vorliegenden Programm werden in jeder Generation folgende Fitnesswerte berechnet:

- **Rohfitness (raw fitness f_r):** Der ursprünglich ermittelte Fitnesswert ohne prinzipielle Begrenzung. Er ist je nach Aufgabenstellung und Art der Berechnung zu maximieren oder zu minimieren.
- **standardisierte Fitness (standardized fitness f_{st}):** [Geyer-Schulz, 1994, S. 218] Linear umgerechnete Rohfitness, wobei bessere Individuen kleinere stan-

dardisierte Fitnesswerte zugeordnet bekommen als schlechtere Individuen. Der bestmögliche standardisierte Fitnesswert sei Null.

- **adjustierte Fitness (adjusted fitness f_a):** [Koza, 1992, S. 97] Liegt für jedes Individuum zwischen 0 und 1, wobei bessere Individuen größere Werte zugewiesen bekommen. Die Ermittlung der adjustierten Fitness aus der standardisierten Fitness kann durch jede, lineare oder nichtlineare, Funktion erfolgen, die diese Bedingungen erfüllt. Um eine größere Flexibilität zu erreichen, wurde im vorliegenden Programm von der einengenden Bedingung eines maximalen Fitnesswertes von 1 abgesehen. Jede Funktion, die sicherstellt, daß bessere Individuen größere adjustierte Fitnesswerte zugewiesen bekommen, als schlechtere, ist geeignet, soferne der kleinstmögliche Wert Null ist. Mit Hilfe einer geeignet gewählten Funktion können zum Beispiel bessere Individuen stärker gewichtet werden als schlechtere, oder unerwünschte Nichtlinearitäten der Rohfitness ausgeglichen werden.

- **normalisierte Fitness: (normalized Fitness f_n):** [Geyer-Schulz, 1994, S. 218] Liegt zwischen 0 und 1, wobei die Summe der normalisierten Fitnesswerte aller Individuen einer Population genau 1 ergibt.

4.2.1 Implementierte Berechnungsmethoden für die standardisierte Fitness

Im vorliegenden Programm kann die Fitness sowohl über eine, als auch über die letzten n Populationen standardisiert werden. Die Berechnungsmethode lautet:

$$f_{st}(b_{i,t}) = f_r(b_{i,t}) - f_r(best_n)$$

wobei $f_{st}(b_{i,t})$ die standardisierte Fitness des Individuums i zu Zeitpunkt t, $f_r(b_{i,t})$ die Rohfitness des Individuums i zum Zeitpunkt t, und $f_r(best_n)$ die Rohfitness des besten Individuums über die letzten n Populationen ist.

Der Wert für $f_r(best_n)$ berechnet sich bei Maximierungsaufgaben:

$$f_r(best_n) = \max_{t=1}^{n}(\max_{i=1}^{m_t}(f_r(b_{i,t})))$$

wobei m_t die Anzahl der Individuen in der Population zum Zeitpunkt t und n die Anzahl der zu berücksichtigenden Populationen darstellt.

Bei Minimierungsaufgaben hingegen gilt:

$$f_r(best_n) = \min_{t=1}^{n}(\min_{i=1}^{m_t}(f_r(b_{i,t})))$$

4.2.2 Implementierte Berechnungsmethoden für die adjustierte Fitness

Innerhalb des vorliegenden Programms ist es ist in jedem Fall notwendig, für jedes Individuum der Population einen adjustierten Fitnesswert zu berechnenen, da die adjustierte Fitness den Ausgangspunkt zur Berechnung der normalisierten Fitness darstellt. Wünscht der Anwender keine spezielle Fitnessadjustierung, so berechnet sich die adjustierte Fitness für jedes Individuum als eine Art invertierte standardisierte Fitness:

$$f_a(b_{i,t}) = f_{st}(worst_t) - f_{st}(b_{i,t})$$

wobei $f_a(b_{i,t})$ die adjustierte Fitness des Individuums i zum Zeitpunkt t, $f_s(worst_t)$ die standardisierte Fitness des schlechtesten Individuums zum Zeitpunkt t, und $f_s(b_{i,t})$ die standardisierte Fitness des Individuums i zum Zeitpunkt t ist.

Damit ist gewährleistet, daß bessere Individuen größere adjustierte Fitnesswerte zugewiesen bekommen, als schlechtere, und der kleinste Wert nicht unter 0 liegt.

Wünscht der Anwender eine Adjustierung der Fitness, so kann er selber jede beliebige, geeignete Funktion dafür angeben. Im Programm vorbereitet ist bereits eine sehr gebräuchliche Funktion, vorgeschlagen von [Koza, 1992, S. 97]:

$$f_a(b_{i,t}) = \frac{1}{1 + f_{st}(b_{i,t})}$$

4.2.3 Implementierte Berechnung der normalisierten Fitness

Die normalisierte Fitness wird berechnet wie folgt:

$$f_n(b_{i,t}) = \frac{f_a(b_{i,t})}{\sum_{j=1}^{m_t} f_a(b_{j,t})}$$

wobei $f_n(b_{i,t})$ die normalisierte Fitness des Individuums i zum Zeitpunkt t, $f_a(b_{i,t})$ die adjustierte Fitness des Individuums i zum Zeitpunkt t, und m_t die Anzahl der Individuen in der Population zum Zeitpunkt t darstellen.

4.3 Implementierte Selektionsmethoden

Nach der Berechnung der Fitnesswerte wird jedem Individuum der Population an Hand der normalisierten Fitness eine Auswahlwahrscheinlichkeit (*target sampling rate*, tsr) zugewiesen, welche beim anschließenden Sampling bestimmt, mit welcher Wahrscheinlichkeit ein bestimmtes Individuum für die weitere Bearbeitung ausgewählt wird. Die implementierten Methoden sind:

- proportionale Selektion
- lineare Rangselektion

4.3.1 Proportionale Selektion

Die proportionale Selektion ist, nicht zuletzt wegen ihrer Einfachheit, die wohl bekannteste Selektionsmethode und wird bei den meisten Einführungen in genetische Lernverfahren zuerst vorgestellt (z.B. [Goldberg, 1989, S. 10ff] oder [Schöneburg *et al.*, 1994, S. 197ff]. Bei der proportionalen Selektion errechnet sich die Auswahlwahrscheinlichkeit (*target sampling rate*, tsr) - wenig überraschend - proportional zur normalisierten Fitness:

$$tsr_{b_{i,t}} = \frac{f_n(b_{i,t})}{\overline{f_n}(B_t)}$$

wobei: $tsr_{b_{i,t}}$ die Auswahlwahrscheinlichkeit (*target samling rate*) des Individuums i zum Zeitpunkt t, $f_n(b_{i,t})$ die normalisierte Fitness des Individuums i zum Zeitpunkt t, und $\overline{f_n}(B_t)$ die durchschnittliche normalisierte Fitness zum Zeitpunkt t darstellen.

Bei Wahl dieser Methode ist jedoch zu bedenken, daß die Gefahr eines abnehmenden Selektionsdruckes bei zunehmender Angleichung der Fitnesswerte in der Population besteht. Dem kann durch eine geeignete Berechnungsmethode der Fitnesswerte entgegengewirkt werden.

4.3.2 Lineare Rangselektion

Um der Gefahr des abnehmenden Selektionsdruckes zu entgehen, kann man auf die "linear rank selection" [Grefenstette und Baker, 1989] ausweichen. Hierbei werden die Individuen nach ihrer normalisierten Fitness aufsteigend sortiert und mit Rangnummern $rank(b_{i,t})$ (beginnend mit Rang 1) versehen. Die Auswahlwahrscheinlichkeit (*target sampling rate*, tsr) errechnet sich dann wie folgt:

$$tsr_{b_{i,t}} = a_{min} + (a_{max} - a_{min})\frac{rank(b_{i,t}) - 1}{n - 1}$$

wobei: $tsr_{b_{i,t}}$ die Auswahlwahrscheinlichkeit (*target sampling rate*) des Individuums i zum Zeitpunkt t, a_{min} die Auswahlwahrscheinlickeit (*target sampling rate*) für das schlechteste Individuum, a_{max} die Auswahlwahrscheinlichkeit (*target sampling rate*) für das beste Individuum, und $rank(b_{i,t})$ der Rang des Individuums i zum Zeitpunkt t nach f_n sortiert darstellt.

Da die Auswahlwahrscheinlickeit eines Individuums größer sein muß als 0, und die Summe aller Auswahlwahrscheiunlickeiten der Individuen einer Population genau n zu

ergeben hat (wobei n für die Anzahl der Individuen in der Population steht), gelten für a_{min} und a_{max} folgende einschränkenden Bedingungen:

$$1 \leq a_{max} \leq 2$$

$$a_{min} = 2 - a_{max}$$

4.4 Implementierte Sampling-Verfahren

Nachdem jedem Individuum der Population eine Auswahlwahrscheinlichkeit (tsr) zugeordnet wurde, muß ein entsprechendes Sample gezogen werden, wobei Individuen mit größerer tsr gegenüber solchen mit kleiner tsr zu bevorzugen sind. Die implementierten Sampling-Verfahren sind:

- Stochastic sampling with replacement (z.B. [Goldberg, 1989, S. 11ff])
- Stochastic universal sampling [Baker, 1987]

4.4.1 Stochastic Sampling With Replacement

Es handelt sich hierbei um die gebräuchlichste Sampling-Methode. Die einzelnen Individuen der Population werden in einem gewichteten Zufallsverfahren ausgewählt, wobei die Gewichtung an Hand der zuvor berechneten Auswahlwahrscheinlichkeit (tsr) erfolgt. Diese Sampling-Methode kann sehr gut mit Hilfe eines Rouletterades veranschaulicht werden, welches in n Sektoren unterschiedlicher Größe unterteilt ist (wobei n die Anzahl der Individuen in der Population darstellt). Die Größe jedes Sektors wird durch die tsr des entsprechenden Individuums bestimmt. Mit m Versuchen kann man nun m Individuen selektieren, wobei die Chance für jedes Individuum ausgewählt zu werden, mit der Größe des entsprechenden Sektors, und somit mit der tsr des Individuums korreliert. Man spricht wegen dieser anschaulichen Darstellung auch vom "Roulett-Wheel-Verfahren"

Problematisch hierbei ist jedoch, daß im Prinzip jedes Individuum mit einer positiven tsr durch Zufall die ganze Folgepopulation ausfüllen kann. Dies stellt vor allem bei kleinen Populationsgrößen ein echtes Problem dar. Zwar werden in der Literatur verschiedene Abwandlungen des Verfahrens vorgeschlagen, wie z.B. das "stochastic sampling with partial replacement" oder das "remainder stochastic sampling without replacement", welche diese Problematik zwar mildern, jedoch nicht zur Gänze beheben.

4.4.2 Stochastic Universal Sampling

Dieses Verfahren, vorgeschlagen von [Baker, 1987], bietet die beste Lösung für das angesprochene Problem. Die Grundidee besteht darin, alle m Individuen in nur einer

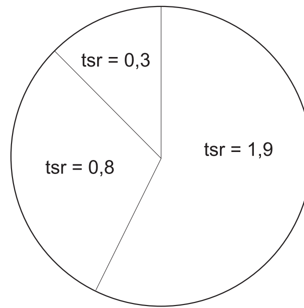


Abbildung 11: Roulett-Wheel Selektion

Zufallsselektion auszuwählen. Will man z.B. 5 Individuen selektieren, dann legt man - um beim Roulett-Beispiel zu bleiben - über das in Sektoren unterteilte Roulettrad ein weiteres Rad mit genau 5 Speichen in gleichem Abstand. Dreht man nun dieses Rad, so kommt nach dem Stillstand jede Speiche über einem bestimmten Sektor des darunter liegenden Roulettrades zu liegen. Damit sind 5 Individuen in einem Schritt selektiert worden, und es besteht auch bei kleinen Populationen keine Gefahr, daß ein einzelnes Individuum die ganze Nachfolgepopulation ausfüllt.

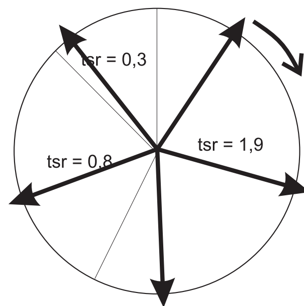


Abbildung 12: Stochastic Universal Sampling: Auswahl von 5 Individuen

In Abbildung 12 wird ein Beispiel von "Stochastic Universal Sampling" gezeigt, bei dem das Individuum mit der größten tsr drei mal, und die beiden anderen Individuen jeweils ein mal ausgewählt werden.

4.5 Implementierte Mating-Verfahren

Nach dem Sampling-Vorgang muß für jedes gezogene Individuum ein "Partner" für die spätere Kreuzung gefunden werden. Dazu stehen im vorliegenden Programm zwei Methoden zur Verfügung.

4.5.1 Random Mating

In diesem Fall wird jedem im Sampling-Vorgang ausgewählten Individuum zufällig ein Individuum aus eben dieser Liste der gesampelten Individuen zugeordnet. Die Problematik stellt sich sinngemäß ähnlich wie beim "stochastic sampling with replacement": Prinzipiell ist es (im Extremfall) möglich, daß allen Individuen nur ein einziges Individuum der Liste als Partner zugeordnet wird.

4.5.2 Random Permutation Mating

Um die eben angesprochene Problematik zu umgehen, wurde das auch von [Geyer-Schulz, 1994] verwendete Verfahren des "random permutation mating" implementiert. Hierbei wird dem Vektor aller ausgewählten Individuen eine Zufallspermutation eben dieses Vektors als Partnerliste zugeordnet.

4.6 Der implementierte Kreuzungsoperator

Im Gegensatz zu einfachen genetischen Algorithmen, bei denen die Gene aus einer Folge von Bits bestehen, und die Kreuzung an jeder beliebigen Stelle durchgeführt werden kann, ist die Sachlage im vorliegenden Fall nicht so einfach. Eine Kreuzung an einer beliebigen Stelle würde in der überwiegenden Mehrzahl der Fälle zu syntaktisch nicht korrekten Wörtern führen. Der Kreuzungsoperator muß daher dahingehend modifiziert werden, daß bei zwei vorliegenden Ableitungsbäumen nur **Unterbäume, beginnend mit dem gleichen Nicht-Terminalsymbol ausgetauscht werden**. Dieser Vorgang kann in folgende atomare Schritte zerlegt werden:

1. Zähle alle Nicht-Terminalsymbole im ersten Ableitungsbaum.
2. Wähle zufällig eines dieser Symbole aus, und stelle fest, um welches Symbol es sich handelt.
3. Zähle im zweiten Ableitungsbaum alle Symbole, die mit dem ausgewählten Symbol ident sind.
4. Gibt es im zweiten Ableitungsbaum keine entsprechenden Symbole, so kehre zurück zu Punkt 2, außer es wurde eine Schranke von n Versuchen bereits erreicht (In diesem Fall Abbruch).
5. Wähle zufällig eines der abgezählten Symbole im zweiten Ableitungsbaum aus.
6. Erzeuge einen neuen Ableitungsbaum, der ident ist mit dem ersten Ableitungsbaum, und tausche den Unterbaum, der mit dem ausgewählten Nichtterminalsymbol beginnt, gegen den ausgewählten Unterbaum des zweiten Ableitungsbaumes aus.

Das oben beschriebene Verfahren kann leicht (und wurde im vorliegenden Programm auch) zu einer n-Punkt-Kreuzung erweitert werden, indem die Schritte 2 bis 6 n mal wiederholt werden, wobei der erste Ableitungsbaum in den Schritten 2 bis n durch den zuletzt erzeugten Ableitungsbaum ersetzt wird. Auf eine Kontrolle, ob zufälligerweise die Kreuzung mehrfach am selben Punkt ausgeführt wird, wurde aus Performance-Gründen verzichtet. Wieviele Bäume getauscht werden sollten, um die beste Leistung des Verfahrens zu erzielen, ist derzeit noch nicht geklärt. Für einfache binäre genetische Algorithmen haben sich aber Werte $n > 2$ als ungünstig erwiesen [DeJong, 1975].

Im vorliegenden Programm wird der beschriebene Kreuzungsoperator mit einer gewissen, anzugebenden Wahrscheinlichkeit p auf jedes in Sampling und Mating ausgewählte Paar angewandt.

4.6.1 Selektionsheuristiken

Da ein Ableitungsbaum sich im allgemeinen von der Wurzel zu den Blättern hin stark verbereitert, ist es einsichtig, daß bei einer einfachen Zufallsauswahl von Nicht-Terminalsymbolen kurze Unterbäume, auf Grund ihrer grösseren Anzahl, stark bevorzugt werden. Es bietet sich daher an, die Wahrscheinlichkeit der Auswahl in irgend einer Weise nach der Größe des Unterbaumes zu gewichten [Geyer-Schulz, 1994, S. 208]. Implementiert wurden folgende Möglichkeiten:

- Keine Selektionsheuristik.
- Gewichtung nach der Größe des Unterbaumes (definiert durch die Anzahl der Nicht-Terminalsymbole im Unterbaum).
- Gewichtung nach der Größe des Suchraumes des Unterbaumes.

Die Umrechnung dieser Werte in tatsächliche Wahrscheinlichkeiten kann im vorliegenden Programm durch Funktionszuweisung mit Hilfe einer beliebigen Funktion erfolgen. [Geyer-Schulz, 1994] schlägt folgende Heuristiken vor:

- Auswahlwahrscheinlichkeit proportional zur Suchraumgröße.
- Auswahlwahrscheinlichkeit proportional zum Logarithmus der Suchraumgröße.
- Auswahlwahrscheinlichkeit proportional zur Quadratwurzel der Suchraumgröße.
- Auswahlwahrscheinlichkeit proportional zum Quadrat der Größe des Unterbaumes (definiert durch die Anzahl seiner Nicht-Terminalsymbole).

Alle diese Varianten sind im vorliegenden Programm durch Funktionszuweisung problemlos anzuwenden.

4.7 Implementierter Mutationsoperator

Ebenso wie bei der Kreuzung nicht beliebige Teile des Ableitungsbaumes ausgetauscht werden können, kann auch bei der Mutation nicht ein beliebiger Teil des Ableitungsbaumes völlig zufällig verändert werden. Im vorliegenden Programm wurde bei der Codierung der Zufallsmutation auf die Kreuzungs-Mechanismen zurückgriffen:

1. Erzeuge einen temporären, zufälligen Ableitungsbaum.
2. Wende den Kreuzungsoperator mit dem zu mutierenden Ableitungsbaum als ersten, und dem temporären Baum als zweiten Baum an.

Die Mutation ist also als eine Kreuzung des zu mutierenden mit einem zufällig erzeugten Baum implementiert. Dies bedeutet, daß alle Selektionsheuristiken wieder zur Anwendung kommen können. Parameter des Kreuzungsoperators sind also die anzuwendende Selektionsheuristik (bzw.: keine Heuristik), sowie die Wahrscheinlichkeit, mit der eine Mutation erfolgen soll.

4.8 Elitismus

Auf Wunsch des Anwenders wird nach erfolgter Mutation das beste Individuum der letzten Population der neu erzeugten Population an zufälliger Stelle hinzugefügt. Das Individuum, welches diesen zufällig gewählten Platz einnimmt, wird dabei überschrieben.

4.9 Inkludieren von Unterableitungsbäumen in die Population

Auf Wunsch können Unterbäume der Population, welche mit einem Startsymbol beginnen und somit selber komplette Ableitungen bilden, und eine gewisse Größe überschreiten, der Population hinzugefügt werden. Da im vorliegenden Programm dieses Hinzufügen nicht durch ein tatsächliches Hinzufügen, sondern durch ein Array von Pointern realisiert wurde, kann man derart ohne viel Performanceverlust die Populationsgröße gewissermaßen schlagartig vergrößern. Ob diese Vorgehensweise in der Praxis anwendbar ist, muß noch Gegenstand weiterer Forschung sein. Es wäre auch denkbar, die Aufnahme eines Unterbaumes in die Population von anderen Kriterien als der Größe abhängig zu machen, beispielsweise von der Fitness. Eine zusätzliche Aufnahme der letzteren Strategie ist in einer zukünftigen Version des Programms geplant.

5 Dokumentation für den Anwender

Dieses Kapitel liefert alle zur Anwendung des Programmes nötigen Informationen. Details über die interne Funktionsweise des Programmes sind hierbei, soferne nicht für den Anwender unbedingt nötig, ausgespart. Um ein lauffähiges Programm zu erstellen, muß der Anwender folgende Programmteile und Dateien implementieren:

- Eine Sprachdefinitionsdatei, in der die zu verwendende Grammatik festgelegt wird.
- Einen Interpreter, der in der Lage ist, Worte dieser Grammatik auszuwerten.
- Eine Funktion, die bestimmten Worten der Grammatik einen bestimmten Rohfitnesswert f_r zuordnet.
- Eine Hauptschleife, in der unter Verwendung der in diesem Kapitel beschriebenen Funktionsaufrufe die anzuwendenden genetischen Operatoren festgelegt und aufgerufen werden.

5.1 Die syntaktische Definition - die Sprachdefinitionsdatei

Ausgangspunkt des genetischen Programms ist eine Sprachdefinitionsdatei, in der der Syntax der zu erzeugenden Wörter in Backus-Naur-Form [Duden Informatik, 1993, S. 52] festgelegt wird. Ein Beispiel für eine gültige Sprachdefinitionsdatei zeigt Abbildung 3. Folgende Regeln sind bei der Erstellung einer korrekten Sprachdefinitionsdatei zu beachten:

- Terminalsymbole werden unter Anführungszeichen geschrieben (z.B.: "D1" oder "D2").
- Nichtterminalsymbole werden durch spitze Klammern gekennzeichnet (z.B.: <f0> oder <f1>).
- Terminalsymbole und Nichtterminalsymbole dürfen keine spitzen Klammern, keine Anführungszeichen und keine Strichpunkte enthalten (wohl aber Leerzeichen und alle anderen Sonderzeichen).
- Linke und rechte Seite einer BNF-Regel werden durch das Symbol := getrennt, die BNF wird mit einem Strichpunkt abgeschlossen (z.B.: f1 := "NOT";).
- Gibt es mehrere rechte Seiten für ein Nichtterminalsymbol, so können diese durch senkrechte Striche getrennt hintereinander geschrieben werden. (z.B. f2 := "OR" | "AND";).
- Das Startsymbol wird durch die Regel S := *Startsymbol*; definiert (z.B.: S := <fe>;).

Der im vorliegenden Programm integrierte Grammatik-Compiler ist aber sehr gutmütig, und kommt auch mit vielen nicht ganz korrekten Sprachdefinitionsdateien zurecht. Folgende Fehler werden (unter Ausgabe einer Warnung) korrigiert:

- Steht auf der linken Seite einer BNF-Regel ein Symbol unter Anführungszeichen, so wird dieses Symbol dennoch als Nichtterminalsymbol interpretiert.
- Steht auf der rechten Seite der BNF-Regel ein mit spitzen Klammern gekennzeichnetes Nichtterminalsymbol, welches nicht aufgelöst wird, so wird angenommen, daß es sich in Wahrheit um ein Terminalsymbol handelt.
- Ein Terminalsymbol kann auch mit einer spitzen Klammer beendet werden.
- Ein Nichtterminalsymbol kann auch mit einem Anführungszeichen beendet werden.
- Fehlt die Definition des Startsymbols, so wird das erste Nichtterminalsymbol als Startsymbol angenommen.
- Alle Zeichen außerhalb von Anführungszeichen und spitzen Klammern, mit Ausnahme von `S`, `<`, `>`, `|`, Anführungszeichen, Strichpunkt und der Zeichenfolge `:=` werden ignoriert.
- Wird beim letzten Symbol einer Ableitungsregel das Anführungszeichen oder die "spitze Klammer zu" vergessen, so wird die Regel dennoch richtig interpretiert.

Aufgrund dieser Eigenschaften ist es z.B. möglich, eine in einer Datei abgespeicherte Farm von Individuen, welche am Ende eine Sprachdefinition enthält, direkt in den Grammatik-Compiler zu laden. Dieser ignoriert dann alle nicht zur Sprachdefinition gehörenden Teile der Datei.

5.2 Die semantische Definition - der Sprachinterpreter

Vor Anwendung des Programmes ist es nötig, ein Programmmodul zur Verfügung zu stellen, welches die vom genetischen Algorithmus erzeugten Worte auswertet und ihnen einen Rohfitnesswert zuordnet. Dazu sind im vorliegenden Programm zwei Methoden vorgesehen.

5.2.1 Interpretation des fertigen Strings (des Phänotyps)

Bei dieser Methode ist das Programm, welches die erzeugten Individuen auswertet, völlig vom genetischen Algorithmus getrennt. Schnittstelle zwischen genetischem

Algorithmus und Auswertung ist die Methode `float StartNode::ObjFuncVal()` welche vom Anwender der Problemstellung entsprechend auszuformulieren und dem Programm hinzuzufügen ist. Dazu folgende Information: Das vorliegende Programm hält alle erzeugten Individuen als vollständige Ableitungsbäume im Speicher. Ein kompletter Ableitungsbaum hat immer ein Objekt der Klasse `StartNode` als Wurzel. Zur Fitnessberechnung ruft das vorliegende Programm von allen Objekten der Klasse `StartNode` in der Population die eben angesprochene Methode `float StartNode::ObjFuncVal()` auf, und erwartet die Rückgabe eines Rohfitnesswertes. Hierzu ein kleines Beispiel:

Gegeben sei wiederum die Grammatik aus Abbildung 3. Das zu lösende Problem bestehe darin, mit den zur Verfügung stehenden boolschen Operatoren `AND`, `OR` und `NOT` eine Repräsentation für `(XOR(D1)(D2))` zu finden. Eine mögliche Ausformulierung von `StartNode::ObjFuncVal()` wäre:

```
      /*------------------------------------------------------
      float Tree::ObjFuncVal()

      Implementation for calculating the raw
 5    fitness for trees of language xor by using
      a separate interpreter.

      My_Evaluation{char*, int, int)

10    Trees, representing a solution for XOR((D1)(D2)),
      get a objective function value 4, because of
      4 hits in the truth-table.
      ------------------------------------------------------
    */
15
    float StartNode::ObjFuncVal()
    {
            char* x;

20          x = PhenoTypeString();

            return(
                    (!My_Evaluation(x,0,0)) +
                      My_Evaluation(x,0,1)  +
25                    My_Evaluation(x,1,0)  +
                    (!My_Evaluation(x,1,1))
                  );

}
```

Die Funktion `char* PhenoTypeString()`, die in diesem Kontext zur Anwendung kommt, liefert einen C++-String mit dem Phänotyp des Ableitungsbaumes - eben das erzeugte Wort der Grammatik. Die vom Anwender auzuformulierende Funktion `int My_Evaluation(char* x, int y, int z)` hat die Aufgabe, diesen String `x` zu interpretieren, für `D1` und `D2` die Werte von `y` und `z` einzusetzen und das Ergebnis dieser Evaluation zurückzugeben.

In der oben gezeigten Variante von `float StartNode::ObjFuncVal()` wird jeder "Treffer" in der Wahrheitstabelle für alle möglichen Kombinationen von `D1` und `D2` mit eins

belohnt. Stimmen alle vier Kombinationen mit dem gewünschten Ergebnis überein, so ergibt das einen Rohfitnesswert von vier. Stimmt keine der möglichen Kombinationen (das entspricht der Funktion `(NOT(XOR(D1)(D2)))`), so ergibt das einen Rohfitnesswert von null.

Es ist empfehlenswert, in `float Tree::ObjFuncVal()` die Funktion `PhenoTypeString()`, so wie im obigen Beispiel, nur einmal aufzurufen und das Ergebnis in einer temporären Variablen zu sichern, da bei jedem Aufruf von `PhenoTypeString()` der String rechenzeitaufwendig neu erzeugt wird.

Der Vorteil der eben gezeigten Methode der Interpretation des ganzen Strings besteht in der völligen Kapselung der Auswertung des Individuums vom genetischen Algorithmus. Dieser Vorteil wird jedoch mit zusätzlichem Programmieraufwand und schlechterem Laufzeitverhalten im Vergleich zur nächsten gezeigten Methode erkauft. Denn zuerst wird durch die Funktion `char* GenoTypeString()` der bereits im Speicher vorliegende Ableitungsbaum in einen String übersetzt, welcher dann von der (vom Anwender auszuprogrammierenden) Funktion `float My_Evaluation(char*, int, int)` zur Interpretation wieder in einen Baum zurückübersetzt werden muß. Über Programmiertechniken im Compilerbau kann man sich beispielsweise in den Einführungswerken [Aho, Sethi und Ullman, 1988] oder [Wirth 1986] informieren.

5.2.2 In-Tree-Auswertung des Ableitungsbaums (des Genotyps)

Wenn nicht bereits ein fertiger Interpreter für Worte der verwendeten Grammatik vorliegt, kann es unter Umständen einfacher sein, die Evaluierung des Individuums direkt dem bereits vorliegenden Ableitungsbaum hinzuzufügen. Dazu sind jedoch ein paar zusätzliche Informationen über die Funktionsweise des vorliegenden Programms vonnöten:

- Alle Ableitungsbäume im vorliegenden Programm setzen sich aus Objekten der Basisklasse `TreeNode` zusammen (auch `StartNode` ist von `TreeNode` abgeleitet).
- Jedes Objekt der Basisklasse `TreeNode` repräsentiert ein Symbol im Ableitungsbaum.
- Jedes Nichtterminalsymbol hat seine Ableitung im Array `TreeNode** Derivs` gespeichert (das ist also ein Array von `TreeNode`- Pointern), welches Teil der `TreeNode`-Datenstruktur ist. `Derivs[0]` liefert also z.B. einen Pointer auf den ersten Knoten der Ableitung.
- Welches Symbol durch ein bestimmtes Objekt der Basisklasse `TreeNode` repräsentiert wird, ist dem Inhalt der Variablen `SymbTabEntry* Symbol` zu entnehmen, welche Teil der `TreeNode`-Datenstruktur ist. Diese `SymbTabEntry`-Pointer können als Token der Symbole angesehen werden.

- Im Kontext von Objekten der Basisklasse `TreeNode` kann das Token eines bestimmten Symbols durch Aufruf von `TreeLanguage->Symbol("`*Textdarstellung des Symbols*`")` ermittelt werden.

Das folgende Listing zeigt eine mögliche Evaluierungs-Implementierung für Ableitungsbäume der Grammatik L_{XOR} (siehe Abbildung 3).

```
    /*-------------------------------------------------------
      int TreeNode::Evaluate(int, int)

      In-tree implementation for evaluating trees
 5    of language xor. Correct built trees are
      excepted, no syntax check included. (For
      example, there are only 2 variables, D1 and
      D2. If a symbol f0 is found, indicating
      either D1 or D2, and the variable
10    is not D1, it must therefore be D2. There is no
      check if it really is D2.)
      -------------------------------------------------------
    */

15  int TreeNode::Evaluate(int a, int b)
    {
            static SymbTabEntry* f0   = TreeLanguage->Symbol("f0");
            static SymbTabEntry* D1   = TreeLanguage->Symbol("D1");
            static SymbTabEntry* AND  = TreeLanguage->Symbol("AND");
20          static SymbTabEntry* NOT  = TreeLanguage->Symbol("NOT");

            int i;

            SymbTabEntry* Operator = NULL;
25          int x1 = 0;
            int x2 = 0;

            if(Derivs[1]->Symbol==f0)
                    return(( Derivs[1]->Derivs[0]->Symbol==D1  ? a : b ));
30          else
                    if(Derivs[1]->Derivs[0]->Symbol==NOT)
                            return(!Derivs[2]->Evaluate(a,b));
                    else
                            if(Derivs[1]->Derivs[0]->Symbol==AND)
35                                  return( (Derivs[2]->Evaluate(a,b)) &&
                                            (Derivs[3]->Evaluate(a,b))
                                          );
                            else
                                    return( (Derivs[2]->Evaluate(a,b)) ||
40                                          (Derivs[3]->Evaluate(a,b))
                                          );
            return(0);
    }
```

In den Zeilen 17-20 werden die Tokens für die Symbole `f0`, `D1`, `AND` und `NOT` ermittelt und in statischen Variablen gespeichert. Diese Vorgangsweise ist sehr empfehlenswert, da jeder Aufruf von `TreeLanguage->Symbol(char*)` im Programmablauf Zeit kostet, und die Evaluierungsfunktion sehr oft aufgerufen wird. Es wird von syntaktisch korrekten Ableitungsbäumen der Grammatik L_{XOR} ausgegangen, was die Evaluierung zusätzlich beschleunigt. Ist z.B. ein Symbol `f0` gefunden worden, so wird geprüft, ob

seine Ableitung `D1` lautet. Ist dies nicht der Fall, so wird ohne weitere Überprüfung angenommen, daß es sich um `D2` handelt. Daher wird auch das Token für `D2` und für viele andere Symbole der Grammatik L_{XOR} nicht abgefragt.

Und so funktioniert die Funktion `TreeNode::Evaluate(int a, int b)`: Der erste Aufruf erfolgt immer am Startknoten des Ableitungsbaumes. In Zeile 28 wird abgefragt, ob der zweite Knoten der Ableitung (`Derivs[1]` ist der Pointer darauf) `f0` enthält. Ist dies der Fall, so wird in Zeile 29 geprüft, ob sich dieser `f0`-Knoten in `D1` erweitert (`Derivs[1]->Derivs[0]->Symbol==D1`). Tut er das, wird `a` zurückgegeben, ansonsten `b`.

In Zeile 31 wird geprüft, ob der erste Knoten der Ableitung ein `NOT` beinhaltet. Falls dies der Fall ist, muß der dritte Knoten der Ableitung ein Startsymbol `fe` beinhalten (der zweite enthält ja lediglich eine Klammer). Für diesen Startsymbol-Knoten wird in Zeile 32 rekursiv wieder `TreeNode::Evaluate(int a, int b)` aufgerufen, und das Ergebnis invertiert (da es sich ja um ein `NOT` handelt.

In Zeile 34 wird analog das `AND` abgehandelt, und was dann noch übrigbleibt, muß wohl ein `OR` sein (Zeile 39). Die Anweisung `return(0)` in Zeile 42 wird niemals aufgerufen und wurde nur inkludiert, um eine fehlerfreie Kompilierung sicherzustellen. Da die Anzahl und Art der Parameter der Evaluierung von der Problemstellung abhängig sind, ist es unter Umständen nötig, einen entsprechenden Prototypeintrag im Header-File `ga.h` hinzuzufügen. Für folgende Methoden gibt es bereits Prototypeinträge:

```
int Evaluate();
int Evaluate(int);
int Evaluate(int, int);
int Evaluate(int, int, int);
int Evaluate(int, int, int, int);
float Evaluate(float);
float Evaluate(float, float);
float Evaluate(float, float, float);
float Evaluate(float, float, float, float);
```

Nun fehlt nur noch die Implemetierung der Funktion `float StartNode::ObjFuncVal()`:

```
    /*-------------------------------------------------------
      float Tree::ObjFuncVal()

      In-tree implementation for calculating the raw
 5    fitness for trees of language xor.

      Trees, representing a solution for XOR((D1)(D2)),
      get a objective function value 4, because of
      4 hits in the truth-table.
10    -------------------------------------------------------
    */
    float StartNode::ObjFuncVal()
    {
            return( (!Evaluate(0,0)) + Evaluate(0,1) +
15                  Evaluate(1,0) + (!Evaluate(1,1))
                  );
    }
```

Dieses Listing unterscheidet sich nicht wesentlich vom im vorigen Kapitel dargestellten. Einziger Unterschied ist die Tatsache, daß die Evaluierungsfunktion nun keinen Phänotyp-String mehr als Parameter benötigt.

Es ist zu beachten, daß die hier gezeigte Art der Evaluierung auch auf die Benennung der Nichtterminalsymbole und überhaupt auf die gesamte Struktur der Grammatik abzielt. Ändert man z.B. in der Grammatik L_{XOR} das Nichtterminalsymbol `f0` gegen `f9`, so versagt die Evaluierung bereits, obwohl die produzierten Wörter (die Phänotypen) gleich bleiben. Dasselbe gilt natürlich umso mehr für den Austausch der Grammatik gegen eine äquivalente andere Grammatik. Hier ist die im vorigen Kapitel dargestellte Evaluierungsmethode im Vorteil.

5.3 Die Hauptschleife

Das folgende Listing zeigt, wie einfach das vorliegenden Programm in seiner Anwendung ist. Nach Implementierung der Rohfitnessermittlung `float StartNode::ObjFuncVal()`, samt dazu nötiger String- oder In-Tree-Evaluierung, reichen wenige Zeilen Programmcode aus, um mit dem “genetic programming“ zu beginnen:

```
    #include "ga.h"
    #include "xor.cc"
    #define MAXDERIVDEPTH 30
    #define POPSIZE 100
 5  #define HISTORY 5
    #define LANGUAGEDEF "xor.def"

    /*----------------------------------
       main()
10    ----------------------------------
    */
    int main()
    {
            unsigned int i;
15          Farm MainFarm(POPSIZE, HISTORY, SIZEUNIF, LANGUAGEDEF, MAXDERIVDEPTH);

            i = 0;
            while(i<500)
            {
20                  i++;

                    cout << endl << i << ": Fitness " <<
                    MainFarm.Current->BestIndividual->RawFitness << ": ";
                    MainFarm.Current->BestIndividual->Print();
25
                    if(MainFarm.Current->BestIndividual->RawFitness==4)
                            break;

                    MainFarm.NextStep();
30          }

            cout << endl;
            MainFarm.Current->BestIndividual->PrintParents();
    }
```

Die in Zeile 2 inkludierte Datei `xor.cc` enthält die im Kapitel 5.2.2 dargestellte In-Tree-Evaluierung für die Grammatik L_{XOR}, sowie die Rohfitnessberechnung `float StartNode::ObjFuncVal()`. Die in Zeile 3 definierte Präprozessorvariable `MAXDERIVDEPTH` steht für die maximal zulässige Ableitungstiefe, `POPSIZE` in Zeile 4 für die gewünschte Größe der zufälligen Anfangspopulation. `HISTORY` in Zeile 5 gibt an, wieviele Generationen zurückverfolgt werden können. Dies ist einerseits wichtig zur Feststellung von "Stammbäumen", andererseits für alle Berechnugen, die über mehrere Generationen laufen, z.B.: standardisierte Fitness über mehrere Generationen (siehe Kapitel 4.2.1). Die in Zeile 6 definierte Präprozessorvariable `LANGUAGEDEF` legt den Namen der Sprachdefinitionsdatei fest. In Zeile 15 wird ein Objekt `MainFarm` der Klasse `Farm` erzeugt. Eine Farm enthält mehrere Populationen, die aktuelle und die `HISTORY` letzten, und jede Population enthält eine gewisse Anzahl von Individuen. Die Anzahl kann bei einer etwas abgeänderten Hauptschleife auch von Generation zu Generation unterschiedlich sein. Als Initialisierungsparameter werden angegeben:

- die gewünschte initiale Populationsgröße `POPSIZE`,
- die Anzahl der zurückverfolgbaren Generationen `HISTORY`,
- die Art der Initialisierung (gleichverteilt nach Größe der Individuen - `SIZEUNIF`),
- der Name der Sprachdefinitionsdatei `LANGUAGEDEF`,
- die maximal zulässige Ableitungstiefe `MAXDERIVDEPTH`.

Auf die jeweils aktuelle Population in der Farm kann über den Pointer `Current` zugegriffen werden (siehe Zeile 22ff). Das beste Individuum der Population wird über den Pointer `BestIndividual`, das schlechteste über den Pointer `WorstIndividual` angesprochen (siehe ebenfalls Zeile 22ff). Hinter dem Begriff "Individuum" verbirgt sich in diesem Fall der Startknoten `StartNode` eines Ableitungsbaumes. Damit hat es folgende Bewandtnis: Jeder Ableitungsbaum im vorliegenden Programm besteht aus Objekten der Klasse `TreeNode`. Startknoten des Ableitungsbaumes (das sind Knoten, die ein Startsymbol beinhalten) sind abgeleitete Objekte der Klasse `StartNode:public TreeNode`. Und der Wurzelknoten jedes Ableitungsbaumes schließlich ist ein Objekt der Klasse `RootNode:public StartNode`. Wenn also hier von "Pointern auf Individuen" die Rede ist, so handelt es sich tatsächlich um Pointer auf Objekte der Klasse `StartNode` oder `RootNode:public StartNode`. Im Datenteil dieser Objektklasse sind z.B. die aktuellen Fitnesswerte des Individuums abrufbar, und zwar in den Variablen `RawFitness`, `StandardizedFitness`, `AdjustedFitness` und `NormalizedFitness` (siehe Zeile 23). Ebenso stehen dem Anwender verschiedene Funktionen zur Verfügung, z.B. die Methode `TreeNode::Print()`, welche den Phänotyp des Individuums ausdruckt und in Zeile 24 zur Anwendung kommt. In Zeile 28 schließlich wird die Methode `Farm::NextStep()` aufgerufen, welche die nächste Population erzeugt. Mit diesen Informationen dürfte die Funktionweise der obigen Hauptschleife

leicht zu verstehen sein: Sie wird fünfhundertmal durchlaufen, außer das beste Individuum der Population hat einen Rohfitnesswert von vier. Dann ist die Lösung gefunden, und die Schleife wird vorzeitig verlassen. (Zeilen 26 und 27). Zuletzt werden durch Aufruf der Methode `StartNode::PrintParents()` die Eltern des gefundenen Individuums ausgegeben. Selbstverständlich nützt diese kurze Hauptschleife bei weitem nicht alle zur Verfügung stehenden Möglichkeiten aus. So werden z.B. keine expliziten Angaben über die Art der Fitnessberechnung und der Fitness-Skalierung, der Selektionsmechanismen usw. gemacht. Das Klassenobjekt `Farm` greift in diesem Fall auf Default-Einstellungen zurück. Und hier ist der Output der obigen kurzen Hauptschleife:

```
Computing search space size up to n = 30...
done.
Generating population...
done...

1: Fitness 3: (OR(OR(D1)(AND(NOT(NOT(D2)))(NOT(OR(D1)(D2)))))(NOT(NOT(D2))))
2: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(D2))))))(NOT(NOT(D2))))
3: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(D2))))))(NOT(NOT(D2))))
4: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(D2))))))(NOT(NOT(D2))))
5: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D1)))))))(NOT(NOT(D2))))
6: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D1)))))))(NOT(NOT(D2))))
7: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
8: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
9: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
10: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
11: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
12: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(OR(D2)(D2)))))))(NOT(NOT(D2))))
13: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(OR(D1)(AND(D2)(D2))))))(NOT(NOT(D2))))
14: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(OR(D1)(AND(D2)(D1))))))(NOT(NOT(D2))))
15: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(OR(D1)(AND(D2)(D2))))))(NOT(NOT(D2))))
16: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(NOT(NOT(D1))))))(NOT(NOT(D2))))
17: Fitness 3: (AND(D2)(NOT(NOT(AND(D2)(OR(NOT(D1))(AND(NOT(D2))(D1)))))))
18: Fitness 3: (AND(D2)(NOT(NOT(AND(D2)(OR(NOT(D1))(AND(NOT(D2))(D1)))))))
19: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(OR(D1)(AND(D2)(D1))))))(NOT(NOT(D2))))
20: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(AND(D1)(AND(D2)(D1))))))(NOT(NOT(D2))))
21: Fitness 3: (AND(AND(D2)(OR(NOT(D2))(NOT(AND(D1)(AND(D2)(D1))))))(NOT(NOT(D2))))
22: Fitness 3: (AND(AND(D2)(OR(NOT(D2))(NOT(AND(D1)(AND(D2)(D1))))))(NOT(NOT(D2))))
23: Fitness 3: (AND(AND(D2)(OR(NOT(D2))(NOT(AND(D1)(AND(D1)(D1))))))(NOT(NOT(D2))))
24: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(AND(D1)(AND(D1)(D1))))))(NOT(NOT(D2))))
25: Fitness 3: (AND(AND(D2)(OR(NOT(D1))(NOT(AND(D1)(OR(D1)(D1))))))(NOT(NOT(D2))))
26: Fitness 3: (OR(OR(OR(D1)(D1))(D1))(NOT(NOT(D2))))
27: Fitness 3: (OR(OR(OR(D1)(D1))(D1))(NOT(NOT(D2))))
28: Fitness 4: (AND(OR(OR(D2)(OR(D1)(D1)))(D1))(NOT(AND(D2)(D1))))


The Parents are:

(AND(OR(OR(D2)(OR(D1)(D1)))(D1))(NOT(AND(D1)(D1))))

(OR(D1)(OR(NOT(D2))(D2)))
```

Eine Lösung wurde also gefunden, und zwar nach 28 Generationen.

5.3.1 Zur Verfügung stehende Klassen, Funktionen und Variablen

Im folgenden werden die im Rahmen der Hauptschleife zur Verfügung stehenden für den Anwender relevanten Klassen, Funktionen und Variablen erläutert. Die CRC-Charts in Abbildung 13 geben einen Überblick über die benutzerrelevanten Klassen und Abbildung 14 zeigt die entsprechenden Zusammenhänge zwischen diesen Klassen.

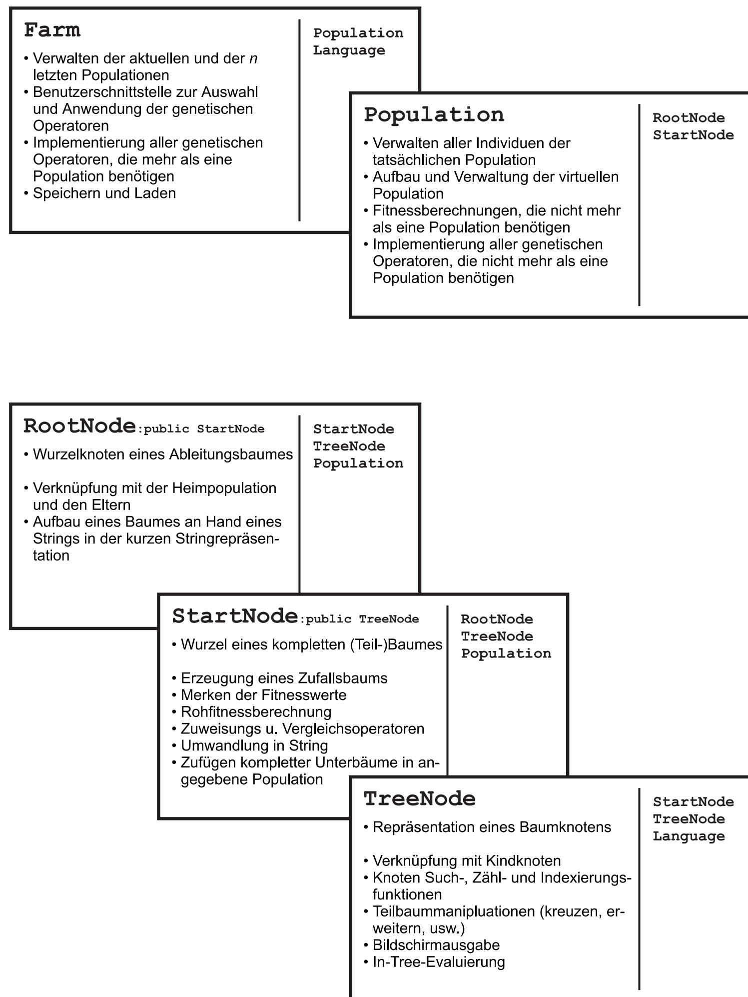


Abbildung 13: Die benutzerrelevanten Klassen

Die Klasse `Farm` enthält in ihrem Datenteil das Objekt `Language L`, welches die Grammatik repräsentiert, und einen Pointer `Population* Current`, der auf die aktuelle Population verweist. Die Populationen selber sind im `private`-Teil der Farm vor Zugriff geschützt.

Jedes Objekt der Klasse `Population` enthält im Datenteil die tatsächliche Population im Array `RootNode* Actual` und eine "virtuelle" Population im Array `StartNode** Virtual`, welche lediglich aus Pointern auf die Startknoten der tatsächlichen Population besteht. Durch diese Art der Implementierung ist sichergestellt, daß

die Aufnahme von Unterableitungsbäumen in den Selektionsmechanismus ohne großen Performanceverlust möglich ist. Programmintern laufen alle Zugriffe stets über die virtuelle Population ab. Selbst wenn also keine Unterableitungsbäume in den Selektionsvorgang aufgenommen werden, muß es dennoch eine virtuelle Population geben, die dann eben gleich groß ist wie die tatsächliche Population und lediglich Pointer auf die `RootNode`-Knoten beinhaltet. Hinzu kommen folgende, für den Anwender relevanten, Variablen:

- `int ActualPopSize`: Größe der tatsächlichen Population
- `int VirtualPopSize`: Größe der virtuellen Population
- `float AdjustedFitnessSum`: Summe der adjustierten Fitnesswerte aller Individuen der virtuellen Population (diese muß nicht unbedingt eins betragen, siehe Kapitel 4.2)
- `StartNode* BestIndividual`: Pointer auf das beste Individuum der virtuellen Population
- `StartNode* WorstIndividual`: Pointer auf das schlechteste Individuum der virtuellen Population
- `int ParentsAvailable`: Ist 1, wenn die Elterngeneration dieser Generation noch in der Farm verfügbar ist

Die `StartNode`-Klasse enthält im Datenteil folgende für den Anwender relevanten Variablen:

- `float RawFitness`: Rohfitnesswert des Individuums
- `float StandardizedFitness`: Standardisierte Fitness des Individuums
- `float AdjustedFitness`: Adjustierte Fitness des Individuums
- `float NormalizedFitness`: Normalisierte Fitness des Individuums
- `float TargSamplRate`: Auswahlwahrscheinlichkeit (tsr) für dieses Individuum

Aus dem `TreeNode`-Anteil der `StartNode:public TreeNode`-Klasse kommen noch folgende wichtige Variablen hinzu:

- `Language* TreeLanguage`: Zeiger auf die Sprachrepräsentation `Language L` in `Farm`.

- `SymbTabEntry* Symbol`: Token (i.e.: Zeiger auf einen Eintrag im Symbol-Table von `Language L` in `Farm`) des Symbols, welches durch diesen Knoten repräsentiert wird.
- `unsigned int DerivationDepth`: Ableitungstiefe dieses Individuums
- `RootNode* Root`: Pointer auf den Wurzelknoten dieses Ableitungsbaumes

Die Klasse `RootNode` schließlich enthält neben den Variablen der Klassen `StartNode` und `TreeNode` noch die folgenden Variablen:

- `Population* Home`: Zeiger auf die Population, der dieser Ableitungsbaum zugehörig ist.
- `StartNode* Parent1`: Zeiger auf den ersten Elternteil dieses Individuums (kann auch `NULL` sein, falls die Eltern nicht verfügbar sind, z.B. in der initialen Population). Vor Verwendung ist zu überprüfen, ob nicht das Flag `Home->ParentsAvailable` auf null steht. In diesem Fall sind die Eltern dieses Individuums nicht mehr verfügbar.
- `StartNode* Parent2`: Zeiger auf den zweiten Elternteil dieses Individuums. Kann ebenfalls `NULL` sein, wenn entweder keine Eltern verfügbar sind, oder es nur einen Elternteil gibt.
- `int Mutated`: Ist 1, falls dieses Individuum im letzten Generationsschritt einer Mutation unterworfen wurde, ansonsten 0.

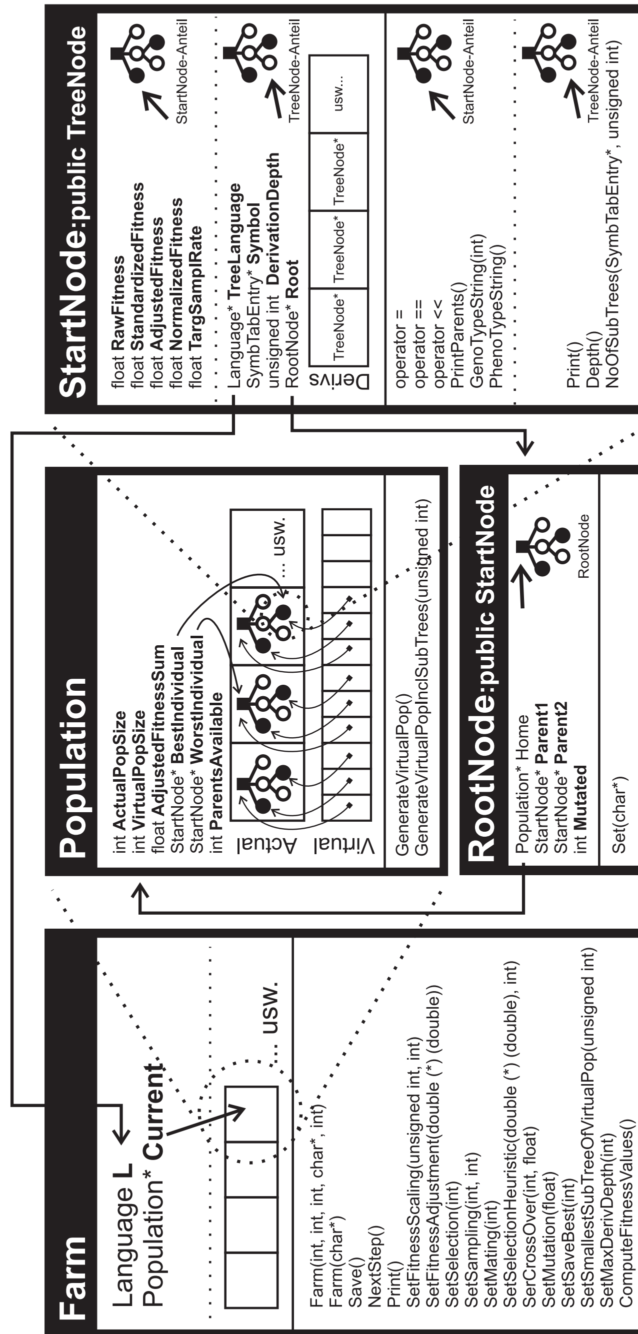


Abbildung 14: Die Zusammenhänge zwischen den benutzerrelevanten Klassen

5.3.2 Initialisierung

Die Initialisierung der Farm erfolgt durch Aufruf des Konstruktors

```
Farm(int psize, int hist, int uniform, char* filename, int mdd)
```

Die einzelnen Parameter haben folgende Bedeutung:

- `int psize`: Steht für die gewünschte initiale Populationsgröße und muß mindestens eins betragen). Die Populationsgröße kann im Laufe der Simulation variiert werden.
- `int hist`: Steht für die Anzahl der zurückverfolgbaren Populationen. Ein Wert von 5 bedeutet beispielsweise, daß für jedes Individuum über 5 Generationen die Ahnen zurückverfolgt werden können, und alle Berechnungen, die Werte von mehreren Generation bennötigen, höchstens 5 Generationen zurückzurechnen imstande sind. Dieser Wert ist nach der Initialisierung fix.
- `uniform`: Steht für die Art der Initialisierung. Erlaubte Werte sind:
  - `NONUNIF` oder `0`: Nicht gleichverteilte Ausgangspopulation
  - `SIZEUNIF` oder `1`: Nach Ableitungstiefe der Individuen gleichverteilte Ausgangspopulation (siehe Kapitel 4.1.2).
  - `EXACTUNIF` oder `2`: nach Ableitungstiefe und Form der Individuen gleichverteilte Ausgangspopulation (siehe Kapitel 4.1.3). Diese Art der Initialisierung ist noch nicht implementiert. Im entsprechenden Fall gibt das Programm bei der Initialisierung eine Warnung aus, und initialisiert nach der vorherigen Methode.
- `char* filename`: Steht für den Namen des Language-Definition Files (siehe Kapitel 5.1)
- `mdd`: Steht für die maximal zulässige Ableitungstiefe. Die Ableitungstiefe ist definiert als die Anzahl der Nichtterminalknoten im Ableitungsbaum [Geyer-Schulz, 1994, S. 280].

Zum Laden einer mit der Funktion `Farm::Save(char*)` gespeicherten Population steht ein eigener Konstruktor `Farm(char*)` zur Verfügung, der im Kapitel 5.3.6 erläutert wird. Eine Initialisierung mit dem Leerkonstruktor `Farm()` ist zwar möglich, jedoch nicht sinnvoll.

5.3.3 Festlegen der genetischen Operatoren und die `NextStep()`-Methode

Es können die anzuwendenden genetischen Operatoren zur Laufzeit frei gewählt und jederzeit geändert werden. Dazu stehen in der Klasse `Farm` eine Reihe von Funktionen zur Verfügung, die alle mit `Set...` beginnen.

- `SetFitnessScaling(unsigned int, int)`
  Diese Funktion definiert die Art der Fitness-Skalierung (siehe Kapitel 4.2.1), also die Art der Umrechnung von der Rohfitness zur standardisierten Fitness. Der erste Parameter gibt an, über wieviele Vorgängergenerationen die Fitness-Standardisierung erfolgen soll:
  - `0` ... Nur die aktuelle Generation wird berücksichtigt (Defaultwert)
  - `1` ... Die aktuelle und die Vorgängergeneration werden berücksichtigt
  - `2` ... Die letzen zwei Generationen werden berücksichtigt
  - usw.

  Stehen weniger Generationen im Speicher zur Verfügung als in diesem Parameter angegeben werden, so wird mit der maximal zur Verfügung stehenden Zahl von Generationen gerechnet. Dies kann zu Beginn der Simulation der Fall sein, oder falls der `hist`-Parameter bei der Initialisierung der Farm zu klein gewählt wurde. Der zweite Parameter definiert, ob die Rohfitness maximiert oder minimiert werden soll:
  - `MAXIMIZE` oder `1` ... Rohfitnessmaximierung (Defaultwert)
  - `MINIMIZE` oder `0` ... Rohfitnessminimierung

- `SetFitnessAdjustment(double (*) (double))`
  Diese Funktion definiert die Art der Fitnessadjustierung (siehe Kapitel 4.2.2), also die Umrechnung der standardisierten Fitness in die adjustierte Fitness. Als Parameter können angegeben werden:
  - `NULL`, falls keine Adjustierung gewünscht wird.
  - Die vorbereitete Funktion `double One_DividedBy_OnePusX(double)` ($\frac{1}{1+x}$) (Defaultwert).
  - Jede beliebige Funktion der Art `double` *function*`(double)`, die die in Kapitel 4.2.2 beschriebenen Bedingungen erfüllt.

- `SetSelection(int, float)`
  Legt die Selektionsmethode, also die Art der Berechnung der Auswahlwahrscheinlichkeit (*target sampling rate*), fest (siehe Kapitel 4.3). Gültige Parameterwerte des ersten Parameters sind

  - `PROPORTIONAL` oder `0` ... proportionale Selektion (Defaultwert).
  - `LINEAR_RANK` oder `1` ... Linear-Rank-Methode.

  Im Falle der Linar-Rank-Selektionsmethode kann ein zweiter Parameter angegeben werden, der den Wert von a_{min}, wie definiert in Kapitel 4.3.2, festlegt. Kleine Werte führen zu hohem, große Werte zu niedrigem Selektionsdruck. Wird der Wert des zweiten Parameters nicht definiert, dann wird ein Default-Wert von 0,2 angenommen.

- `SetSampling(int, int)`
  Legt das Samplingverfahren, also die Art der Auswahl von Individuen an Hand der Auswahlwahrscheinlichkeit (*target sampling rate*) und die Anzahl der zu selektierenden Individuen (somit die Anzahl der Nachkommen) fest (siehe Kapitel 4.4). Gültige Werte für den ersten Parameter sind:

  - `STOCHASTIC_UNIVERSAL` oder `0` ... die Stochastic-Universal-Sampling-Methode (Defaultwert).
  - `STOCH_SAMPL_WITH_REPL` oder `1` ... proportionale Selektion.

  Der zweite Parameter (Anzahl der zu selektierenden Individuen) muß mindestens eins betragen.

- `SetMating(int)`
  Legt das Matingverfahren, also die Art der Zuordnung von Partnern für die selektierten Individuen fest (siehe Kapitel 4.5). Gültige Parameterwerte sind:

  - `RANDOM_PERMUTATION` oder `0` ... Random-Permutation-Mating (Defaultwert)
  - `RANDOM` oder `1` ... Random-Mating

- `SetSelectionHeuristic(double (*) (double), int)`
  Legt die Selektionsheuristik für die Gewichtung der Auswahl von Unterableitungsbäumen bei Kreuzung und Mutation fest (siehe Kapitel 4.6.1). Der erste Parameter gibt die Funktion an, mit der gewichtet werden soll. Er kann folgende Werte annehmen:

  - `NULL`, falls keine Selektionsheuristik gewünscht wird (Defaultwert).
  - Die vorbereitete Funktion `pow2(double)` (x^2)
  - Die vorbereitete Funktion `pow3(double)` (x^3)
  - Die vorbereitete Funktion `pow4(double)` (x^4)
  - Jede beliebige Funktion der Art `double` *function*`(double)`.

  Der zweite Parameter gibt an, auf welcher Basis die Gewichtung der Auswahl erfolgt, und kann folgende Werte annehmen:

  - `TREESIZE` oder `0` ... Gewichtung auf Basis der Teilbaumgröße (Defaultwert)
  - `SEARCHSPACESIZE` oder `1` ... Gewichtung auf Grund der Suchraum größe des zum Startknoten des Teilbaums gehörenden Symbols (siehe Kapitel 4.1.1).

  Die Angabe beider Parameter ist optional, für den fehlenden Parameter wird der Defaultwert eingesetzt.

- `SetCrossOver(int, float)`
  Legt fest, ob eine 1-Punkt oder n-Punkt Kreuzung durchzuführen ist, und mit welcher Wahrscheinlichkeit es durchgeführt wird (siehe Kapitel 4.6). Der erste Parameter bestimmt dabei die Anzahl der Kreuzungspunkte (Defaultwert ist eins), der zweite die Durchführungswahrscheinlichkeit (Defaultwert ist 0,85).

- `SetMutation(float)`
  Legt fest, mit welcher Wahrscheinlichkeit eine Mutation durchzuführen ist (Defaultwert ist 0,05).

- `SetSaveBest(int)`
  Legt fest, ob nach Anwendung des Mutationsoperators auf die Population, das beste Individuum der Vorgängerpopulation in die neue Population übernommen werden soll (Elitismusstrategie, siehe Kapitel 4.8). Gültige Parameterwerte sind:

  - `TRUE` oder `1` ... Elitismus
  - `FALSE` oder `0` ... kein Elitismus

- `SetSmallestSubTreeOfVirtualPop(unsigned int)`
  Legt fest, ob Unterableitungsbäume in die virtuelle Population und somit in die Auslese mit aufgenommen werden sollen, oder nicht. Der Parameter definiert, wie groß Unterbäume mindestens sein müssen, um in die virtuelle Population aufgenommen zu werden. Die Größe eines Baumes ist hierbei definiert durch die Anzahl seiner Nichtterminalsymbole [Geyer-Schulz, 1994, S. 280]. Ein Spezialfall ist die Angabe von null als Parameter: In diesem Fall wird kein einziger Unterableitungsbaum in die virtuelle Population aufgenommen (Defaultwert).

- `Set MaxDerivDepth(int)`
  Legt die maximal zulässige Größe der den Individuen zugrundeliegenden Ableitungsbäumen fest. Jeder positive, ganzzahlige Wert größer null ist gültig. Der im Konstruktor `Farm(int psize, int hist, int uniform, char* filename, int mdd)` angegebene Wert `mdd` kann somit jederzeit verändert werden.

Wird (wie im Beispiel-Listing auf Seite 44) keine nähere Festlegung über die genetischen Operatoren getroffen, so arbeitet das Programm mit den Defaultoperatoren. Die jeweils nächste Generation wird einfach durch Aufruf der `Farm::NextStep()`-Methode erzeugt. An dieser Stelle noch beispielhaft das Listing einer Hauptschleife, in der etliche der eben erläuterten Funktionen angewendet werden:

```
    #include "ga.h"
    #include "xor.cc"
    #define MAXDERIVDEPTH 40
    #define HISTORY 5
 5  #define LANGUAGEDEF "xor.def"

    /*----------------------------------
       main()
      ----------------------------------
10  */
    int main()
    {
            int i;
            int NewPopSize;
15          Farm MainFarm(120, HISTORY, SIZEUNIF, LANGUAGEDEF, MAXDERIVDEPTH);

            MainFarm.SetFitnessScaling(1,MAXIMIZE);
            MainFarm.SetSelection(PROPORTIONAL);
            MainFarm.SetMating(RANDOM);
20          MainFarm.SetSelectionHeuristic(pow2,TREESIZE);
            MainFarm.SetCrossOver(1,0.9);
            MainFarm.SetMutation(0.004);
            MainFarm.SetSmallestSubTreeOfVirtualPop(30);

25          i = 0;
            while(i<100)
            {
                    i++;

30                  cout << endl << i << ": ";
                    MainFarm.Current->BestIndividual->Print();

                    if(MainFarm.Current->BestIndividual->RawFitness==4)
                            break;
35
                    NewPopSize = 20+(100-i);
                    MainFarm.SetSampling(STOCH_SAMPL_WITH_REPL, NewPopSize);
                    MainFarm.NextStep();
            }
```

Im Gegensatz zum Listing auf Seite 44 werden hier in den Zeilen 17-23 von den Defaultwerten abweichende Angaben über die zu verwendenden genetischen Operatoren gemacht, z.B. Fitness-Skalierung unter Berücksichtigung der Vorgängergeneration (Zeile 17); als Selektionsheuristik zur Auswahlwahrscheinlichkeit von Bäumen bei Kreuzung und Mutation wird x^2 festgelegt, wobei x die Größe des jeweiligen Baumes darstelle (Zeile 20), Unterbäume mit Mindestgröße 30 werden in die virtuelle Population mit aufgenommen (Zeile 23); usw.

In den Zeilen 36 und 37 wird gezeigt, wie die Populationsgröße dynamisch zur Laufzeit angepaßt werden kann. In jedem Durchlauf nimmt die Anzahl der zu selektierenden Individuen (somit die Populationsgröße) von 120 beginnend um eins ab. Der unten abgebildete Screen-Shot zeigt den von dieser Main-Loop gelieferten Output:

```
Computing search space size up to n = 40...
done.
Generating population...
done...
```

```
1: (AND(D2)(NOT(AND(NOT(AND(NOT(OR(D1)(NOT(NOT(D2)))))(NOT(D2))))(D1))))
2: (OR(AND(OR(NOT(D1))(NOT(NOT(NOT(NOT(OR(D1)(D1)))))))(D1))(OR(D2)(D2)))
3: (AND(D2)(NOT(AND(NOT(NOT(OR(NOT(NOT(D1)))(NOT(NOT(D2))))))(D1))))
4: (AND(OR(NOT(D1))(NOT(OR(D2)(AND(NOT(D1))(NOT(NOT(NOT(NOT(D1)))))))))(D1))
5: (OR(D2)(OR(AND(NOT(OR(NOT(D2))(D1)))(NOT(D1)))(D1)))
6: (OR(D2)(OR(OR(D2)(OR(AND(D1)(OR(D1)(NOT(AND(D1)(D1)))))(D1)))(D1)))
7: (OR(OR(D1)(AND(D1)(NOT(OR(OR(D1)(OR(D1)(AND(NOT(D2))(D1))))(D1)))))(D2))
8: (NOT(OR(NOT(NOT(NOT(OR(D1)(OR(D1)(OR(D2)(AND(NOT(D2))(D1))))))))(D2)))
9: (OR(D2)(OR(D1)(OR(D1)(OR(D2)(AND(OR(AND(NOT(D2))(NOT(D2)))(D1))(D1))))))
10: (NOT(OR(D1)(OR(NOT(NOT(OR(D1)(OR(AND(NOT(D2))(NOT(D2)))(D1)))))(D1))))
11: (OR(OR(D2)(OR(D2)(OR(D1)(OR(D1)(OR(D2)(AND(NOT(D2))(D1)))))))(D1))
12: (OR(D2)(OR(D2)(OR(D1)(OR(D1)(OR(D2)(AND(OR(AND(NOT(D2))(NOT(D2)))(D1))(D1)))))))
13: (AND(OR(D1)(NOT(OR(AND(NOT(D2))(NOT(D2)))(D1))))(OR(NOT(D2))(NOT(AND(D1)(D2)))))
```

Eine korrekte Lösung wird also bereits nach 13 Generationen gefunden.

5.3.4 Bestimmen der Herkunft und der Eltern eines Individuums

Die einfachste Möglichkeit, etwas über die Herkunft eines bestimmten Individuums zu erfahren, besteht im Aufuf der `StartNode::PrintParents()`-Methode (siehe Hauptschleifen-Listing auf Seite 44). Es ist jedoch auch möglich, direkt auf die Eltern eines bestimmten Individuums zuzugreifen, sowie Information darüber zu gewinnen, ob ein bestimmtes Individuum einer Mutation unterworfen wurde. Zur Bestimmung der Herkunft eines Individuums ist folgende Vorgehensweise einzuhalten:

- Zunächst kann geprüft werden, ob das Individuum (stets dargestellt durch ein Objekt der Klasse `StartNode`) untergeordneter Teilbaum eines größeren Ableitungsbaumes ist. Dies kann im Kontext von `StartNode` durch die Variable `Root` abgefragt werden. `Root` ist im Startknoten `StartNode` ein Pointer, der auf den Wurzelknoten `RootNode` verweist. Verweist nun `Root` auf das Knotenobjekt, welches geprüft werden soll ("zeigt auf sich selber"), dann ist dieser Knoten sein eigener Wurzelknoten. Oder einfacher: Es ist der Wurzelknoten. Das zu prüfende Individuum ist also nicht Teil eines anderen. Andernfalls, also wenn `Root` auf einen anderen Knoten verweist, dann handelt es sich um einen Teilableitungsbaum.

- Außerdem ist stets abfragbar, ob das Individuum einer Mutation unterworfen wurde. Dies ist im Kontext von `StartNode` dem Flag `Root->Mutated` zu entnehmen. Ist es `TRUE` oder `1`, dann ist gerade eine Mutation passiert.

- Im nächsten Schritt ist zu überprüfen, ob potentielle Eltern überhaupt noch im Speicher verfügbar sind. Im Kontext von `StartNode` erfolgt dies durch Abfrage der Variablen `Root->Home->ParentsAvailable` (siehe Kapitel 5.3.1). `Home` ist ein Pointer, der vom Wurzelknoten `RootNode` auf die Population verweist, der das Individuum angehört. In der Population schließlich ist in der boolschen Variablen `ParentsAvailable` vermerkt, ob die Eltern noch verfügbar sind. Falls hier `FALSE` oder `0` eingetragen ist, muß an dieser Stelle abgebrochen werden.

- Dann ist zu prüfen, ob für das spezielle Individuum Eltern verfügbar sind (z.B. ist dies bei "coaching", also beim Einbringen von Individuen durch den Anwender (siehe Kapitel 5.3.5), nicht der Fall). Sollten keine Eltern verfügbar sein, so ist im Kontext von `RootNode` die Variable `Root->Parent1` gleich `NULL`. In diesem Fall muß hier abgebrochen werden. Andernfalls verweist `Root->Parent1` auf den ersten Elternteil.

- Die Variable `Root->Parent2` im Kontext `StartNode` enthält einen Pointer auf den zweiten Elternteil, soferne existent. Ist kein zweiter Elternteil vorhanden, dann enthält sie `NULL`. Dies ist z.B. der Fall, wenn keine Kreuzung durchgeführt wurde.

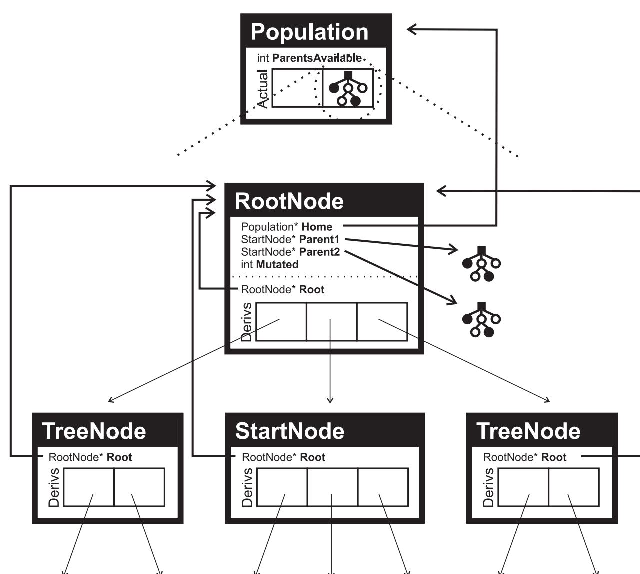


Abbildung 15: Bestimmung der Eltern

Die entprechenden Zusammenhänge sind in Abbildung 15 dargestellt. Der folgende Listingauszug, bei dem das eben vorgestellte Verfahren zur Anwendung kommt, zeigt einen Ersatz für den `PrintParents()`-Befehl, wie er z.B. in Zeile 32 des Listings von Seite 44 vorkommt.

```
    StartNode* Best = MainFarm.Current->BestIndividual;
    if(!Best->Root->Home->ParentsAvailable || Best->Root->Parent1==NULL)
            cout << "No parents available." << endl << endl;
    else
 5  {
            if(Best->Root!=Best)
            {
                    cout << "Individual is part of the following:";
                    cout << endl << endl;
10                  Best->Root->Print();
                    cout << endl << endl;
             }
             if(Best->Root->Parent2==NULL)
             {
```

```
15                  cout << "This individual has been cloned without ";
                    cout << "any crossover from:";
                    cout << endl << endl;
                    Best->Root->Parent1->Print();
                    cout << endl << endl;
20          }
            else
            {
                   cout << "The Parents are:";
                   cout << endl << endl;
25                 Best->Root->Parent1->Print();
                   cout << endl << endl;
                   Best->Root->Parent2->Print();
                   cout << endl << endl;
            }
30   }

     if(Best->Root->Mutated)
            cout << "There was a mutation." << endl << endl;
```

5.3.5 Coachen der Population

Unter "Coachen" versteht man das gezielte Einbringen von Individuen in die Population durch den Anwender. Dies muß im vorliegenden Programm in drei Schritten erfolgen:

1. Überschreiben bestimmter Individuen der tatsächlichen Population mit Hilfe der Funktion `RootNode::Set(char*)`
2. Neuaufbau der virtuellen Population mit Hilfe der Funktion `Population::GenerateVirtualPop()` oder mit Hilfe der Funktion `Population::GenerateVirtualPopIncSubTrees(unsigned int)`
3. Aktualisieren der Fitnesswerte durch Aufruf von `Farm::ComputeFitnessValues`

Die Funktion `RootNode::Set(char*)` erwartet als Parameter den Ableitungsbaum einer bestimmten Grammatik in einer ganz bestimmten Notation, die im Rahmen dieser Arbeit "kurze Stringrepräsentation" genannt werden soll. Für ein bestimmtes, bereits exisitierendes Individuum, ist sie mit Hilfe der Funktion `char* StartNode::PhenoTypeString()`, oder noch einfacher über den Operator `StartNode::operator <<` abzufragen. Der Syntax läßt sich am Besten durch ein kleines Beispiel veranschaulichen:

Gegeben sei wieder die Grammatik L_{XOR} aus Abbildung 3, sowie das Wort (NOT(D2)). Die entsprechende kurze Stringrepräsentation erhält man, indem man den Ableitungsbaum vom Wurzelknoten her nachvollzieht, und für jede Ableitung, die man macht, die entsprechende Nummer der Ableitung gefolgt von einem Punkt dem String hinzufügt

(Linksableitung). Dabei ist jeweils das am weitesten links befindliche, noch nicht aufgelöste, Nichtterminalsymbol als nächstes zu erweitern. Die Nummerierung der Ableitungen ergibt sich aus der Grammatik. Die erste Ableitung erhält die Nummer 1, die zweite die Nummer 2, usw. Im gegebenen Beispiel funktioniert das folgendermaßen:

- <fe> ⇒ (<f1><fe>)
  Der String lautet: “2.“
  (da es sich bei (<f1><fe>) um die zweite Ableitung von <fe> handelt)
- (<f1><fe>) ⇒ (NOT<fe>)
  Der String lautet: “2.1.“
  (da es sich bei NOT um die erste Ableitung von <f1> handelt)
- (NOT<fe>) ⇒ (NOT(<f0>))
  Der String lautet: “2.1.1.“
  (da es sich bei (<f0>) um die erste Ableitung von <fe> handelt)
- (NOT(<f0>)) ⇒ (NOT(D2))
  Der String lautet: “2.1.1.2.“
  (da es sich bei D2 um die zweite Ableitung von <fe> handelt)

Die kurze Stringrepräsentation von (NOT(D2)) lautet für die angegebene Grammatik also: “2.1.1.2.“. Das folgende Beispiel verdeutlicht die praktische Anwendung des Coaching:

```
     #include "ga.h"
     #include "xor.cc"
     #define MAXDERIVDEPTH 30
     #define POPSIZE 100
 5   #define HISTORY 5
     #define LANGUAGEDEF "xor.def"

     /*-----------------------------------
        main()
10    -----------------------------------
     */
     int main()
     {
             unsigned int i;
15           Farm MainFarm(POPSIZE, HISTORY, SIZEUNIF, "xor.def", MAXDERIVDEPTH);

             MainFarm.Current->Actual[0].Set("2.1.1.2.");
             MainFarm.Current->
             Actual[1].Set("3.1.3.2.2.1.1.2.1.1.3.2.2.1.2.1.1.2.2.1.1.1.");
20
             MainFarm.Current->GenerateVirtualPop();
             MainFarm.ComputeFitnessValues();


25           i = 0;
             while(i<500)
             {
                     i++;
```

```
30                  cout << endl << i << ": Fitness " <<
                    MainFarm.Current->BestIndividual->RawFitness << ": ";
                    MainFarm.Current->BestIndividual->Print();

                    if(MainFarm.Current->BestIndividual->RawFitness==4)
35                          break;

                    MainFarm.NextStep();
            }

40          cout << endl;
            MainFarm.Current->BestIndividual->PrintParents();
    }
```

Dieses Listing unterscheidet sich vom Listing auf Seite 44 lediglich durch die Zeilen 17-22. In diesen Zeilen werden das erste und zweite Individuum der tatsächlichen Population durch zwei andere, in ihrer kurzen Stringrepräsentation gegebenen, Individuen ersetzt und anschließend die virtuelle Population neu aufgebaut und die Fitnesswerte neu berechnet. Startet man das Programm, so ergibt sich folgender Output:

```
Computing search space size up to n = 30...
done.
Generating population...
done...

1: Fitness 4: (OR(AND(NOT(D2))(D1))(AND(NOT(NOT(D2)))(NOT(D1))))


No parents available.
```

Der Grund für die Kürze dieses Outputs liegt darin, daß das in den Zeilen 18 und 19 hinzugefügte Individuum bereits eine Lösung für das XOR-Problem darstellt.

"3.1.3.2.2.1.1.2.1.1.3.2.2.1.2.1.1.2.2.1.1.1." ist die kurze Stringrepräsentation von `(OR(AND(NOT(D2))(D1))(AND(NOT(NOT(D2)))(NOT(D1))))`. Das Coaching hat also geklappt.

5.3.6 Speichern und Laden: Anwendung und Datenformat

Vor allem bei längeren Simulationen, oder auch zum Zwecke der späteren Auswertung, kann das Abspeichern vom Populationen sehr hilfreich sein. Zu diesem Zwecke steht die Funktion `Farm::Save(char*` *filename*`)` zur Verfügung. Diese Funktion sichert alle zur Wiederherstellung einer Farm nötigen Daten in die Datei des angegebenen Namens. Gesichert werden:

- Die aktuelle und die zur Verfügung stehenden letzten tatsächlichen Generationen.
- Die Eltern-Kind-Beziehungen und das `RootNode::Mutated`-Flag.

- In jeder Population die Information über die minimale Größe der in die virtuelle Population aufzunehmenden Ableitungsbäume.
- Die Fitnesswerte und tsr der Individuen in der virtuellen Population.
- Die Grammatik.

Sobald ein Klassenobjekt des Typs `Farm` exisitiert, kann die Funktion jederzeit ohne Einschränkungen aufgerufen werden. Hier eine Beschreibung des Fileformats (alle Zahlenwerte werden als ASCII-Text dezimal codiert abgespeichert):

- **Fileheader**: Der Fileheader enthält die folgenden Werte. Jeder Wert wird durch ein Nummernsymbol (`#`) abgeschlossen. Der Fileheader selbst wird durch einen Zeilenwechsel abgeschlossen.
  - Nummer des Fileformats (momentan 1)
  - Anzahl der verfügbaren Vorgängergenerationen
  - Art der Initialisierung:
    - 0 ... nicht gleichverteilt
    - 1 ... gleichverteilt nach Größe
    - 2 ... gleichverteilt nach Form und Größe
  - maximal zulässige Ableitungstiefe
- **Population**: Dieser Teil folgt mehrere Male hintereinander in der Datei, je nach Anzahl der abgespeicherten Populationen
  - **Die tatsächliche Population**:
    - **Populationsheader**: Beginnt mit einem `P`, gefolgt von einem Zeilenwechsel und enthält die folgenden Werte, stets gefolgt von einem Nummernsymbol (`#`). Abgeschlossen wird der Populationsheader durch einen Zeilenwechsel.
      - Größe der tatsächlichen Population.
      - Minimale Ableitungstiefe, mit der Individuen noch in den virtuellen Teil dieser Population aufgenommen werden.
      - Das Flag, das angibt, ob die Elterngeneration dieser Population verfügbar ist.
      - Die Nummer des besten Individuums des virtuellen Teils der Population.
      - Die Nummer des schlechtesten Individuums des virtuellen Teils der Population.

- · Die Summe der adjustierten Fitnesswerte der im virtuellen Teil der Population referenzierten Individuen.

  * **Die Individuen**: Beginnen mit einem Rufzeichen und enden mit einem Zeilenwechsel. Die einzelnen Daten werden durch ein Nummernsymbol (`#`) abgeschlossen. Zu jedem Individuum werden folgende Werte gespeichert:
    - · Das Individuum in seiner kurzen Stringrepräsentation.
    - · Nummer des ersten Elternteils (null für unbekannt).
    - · Nummer des zweiten Elternteiles (null für unbekannt).
    - · Mutated-Flag: 1 bei Mutation, 0 bei keiner Mutation.

- **Die virtuelle Population**
  * **Header virtuelle Poulation**: Besteht aus einem `V`, gefolgt von einem Zeilenwechsel.
  * **Fitnesswerte für jedes Mitglied der virtuellen Population**: Beginnen mit einem Rufzeichen und enden mit einem Zeilenwechsel. Die einzelnen Daten werden durch ein Nummernsymbol (`#`) abgeschlossen. Zu jedem Individuum der virtuellen Population sind die folgende Werte gespeichert. Jeder dieser Fitnessvektoren wird mit einem Zeilenwechsel abgeschlossen:
    - · Rohfitness
    - · standardisierte Fitness
    - · adjustierte Fitness
    - · normalisierte Fitness
    - · target sampling rate

- **Die Grammatik**: Die Grammatik in ihrer Backus-Naur-Form.

Das Laden einer derart abgespeicherten Datei erfolgt mit Hilfe des Konstruktors `Farm(char*` *filename*`)`. Es ist dabei zu beachten, daß nur die zuvor im Dateiformat beschriebenen Werte wiederhergestellt werden, nicht aber die Informationen über die für den nächsten Generationenschritt anzuwendenden genetischen Operatoren, Selektionsheuristiken, etc. Das bedeutet, daß diese Angaben mit Hilfe der `Farm::Set...()`-Befehle wiederhergestellt werden müssen.

Es ist außerdem wichtig zu wissen, daß zwar die Fitnesswerte der geladenen Vorgängergenerationen unverändert wiederhergestellt werden, die Fitnesswerte der aktuellen Population jedoch bereits aufgrund der ausgewählten Fitnessberechnungsmethoden erfolgt. Dies bedeutet, daß z.B. unmittelbar nach Aufruf des Konstruktors `Farm(char*` *filename*`)` die Fitnesswerte der aktuellen Population anhand der Defaulteinstellungen ermittelt werden und von den gespeicherten abweichen können. Ändert man jedoch diese Einstellungen mit Hilfe der `Farm::Set...()`-Befehle auf die Einstellungen, die

zum Zeitpunkt der Speicherung gültig waren, kann man z.B. eine unterbrochene Simulation fortsetzen, als wäre sie nie unterbrochen worden. Eine Erweiterung auch für alle anderen (speicherbaren) Parameter eines Laufes ist geplant.

5.3.7 Sonstige relevante Funktionen

In Abbildung 14 sind noch ein paar für den Anwender relevante Funktionsaufrufe genannt, die bisher noch keine Erwähnung gefunden haben, oder nur nebenbei abgehandelt wurden. Diese Funktionen werden nun in diesem Kapitel erläutert.

5.3.7.1 `Farm::Print()`

Diese Funktion druckt die gesamte aktuelle Population aus. Hier ein Beispielausdruck einer Population der Größe 4:

```
Population size: 4
Virtual population size: 13

POPULATION:

Individual 1 raw-fitness: 1 normalized fitness: 0.0434783
(NOT(NOT(NOT(OR(D2)(AND(NOT(AND(NOT(D1))(D2)))(AND(D1)(D1)))))))

       Subtree 1 raw-fitness: 3 normalized fitness: 0.130435
       (NOT(NOT(OR(D2)(AND(NOT(AND(NOT(D1))(D2)))(AND(D1)(D1))))))

       Subtree 2 raw-fitness: 1 normalized fitness: 0.0434783
       (NOT(OR(D2)(AND(NOT(AND(NOT(D1))(D2)))(AND(D1)(D1)))))

       Subtree 3 raw-fitness: 3 normalized fitness: 0.130435
       (OR(D2)(AND(NOT(AND(NOT(D1))(D2)))(AND(D1)(D1))))

Individual 2 raw-fitness: 2 normalized fitness: 0.0652174
(AND(D1)(AND(D1)(NOT(NOT(OR(OR(NOT(D1))(AND(D2)(D1)))(NOT(D2)))))))

       Subtree 1 raw-fitness: 2 normalized fitness: 0.0652174
       (AND(D1)(NOT(NOT(OR(OR(NOT(D1))(AND(D2)(D1)))(NOT(D2))))))

       Subtree 2 raw-fitness: 2 normalized fitness: 0.0652174
       (NOT(NOT(OR(OR(NOT(D1))(AND(D2)(D1)))(NOT(D2)))))

       Subtree 3 raw-fitness: 2 normalized fitness: 0.0652174
       (NOT(OR(OR(NOT(D1))(AND(D2)(D1)))(NOT(D2))))

Individual 3 raw-fitness: 3 normalized fitness: 0.130435
(NOT(AND(AND(D2)(AND(OR(D1)(NOT(D1)))(D1)))(NOT(NOT(NOT(NOT(D1)))))))

       Subtree 1 raw-fitness: 1 normalized fitness: 0.0434783
       (AND(AND(D2)(AND(OR(D1)(NOT(D1)))(D1)))(NOT(NOT(NOT(NOT(D1))))))

Individual 4 raw-fitness: 3 normalized fitness: 0.130435
(NOT(AND(AND(AND(D1)(D1))(AND(OR(NOT(NOT(D1)))(NOT(D1)))(D2)))(D1)))

       Subtree 1 raw-fitness: 1 normalized fitness: 0.0434783
       (AND(AND(AND(D1)(D1))(AND(OR(NOT(NOT(D1)))(NOT(D1)))(D2)))(D1))

       Subtree 2 raw-fitness: 1 normalized fitness: 0.0434783
       (AND(AND(D1)(D1))(AND(OR(NOT(NOT(D1)))(NOT(D1)))(D2)))
```

Wie man erkennt, werden die Mitglieder der virtuellen Population, die keine eigenständigen Bäume darstellen, jeweils nach dem Individuum, von dem sie einen Teilbaum darstellen, eingerückt ausgedruckt.

5.3.7.2 `StartNode::operator =()`

Es handelt sich hierbei um einen Zuweisungsoperator, mit dem man z.B. ein Individuum der tatsächlichen Population auf ein temporäres Objekt der Klasse `StartNode` umkopieren könnte. Vorsicht geboten ist jedoch bei der direkten Zuweisung von Individuen in die tatsächliche Population. Zunächst gilt sinngemäß das Gleiche, wie beim Coachen (siehe Kapitel 5.3.5) einer Population: nach erfolgter Zuweisung müssen die virtuelle Population und die Fitnesswerte neu berechnet werden. Hinzu kommt jedoch, daß die im Individuum gespeicherten Elternbezüge ihre Konsistenz verlieren können. Vom Gebrauch dieses Operators durch den Anwender innerhalb der Population ist also eher abzuraten.

5.3.7.3 `StartNode::operator ==`

Hierbei handelt es sich um einen Vergleichsoperator, mit dessen Hilfe überprüft werden kann, ob zwei Individuen identische Ableitungsbäume haben. Aber Achtung! Bei manchen Grammatiken können unterschiedliche Ableitungsbäume (Genotypen) zu gleichen Wörtern (Phänotypen) führen. Es kann daher passieren, daß dieser Operator eine Ungleichheit zweier Individuen ergibt, die jedoch bei Anwendung der `TreeNode::Print()`-Funktion dasselbe Wort ausgeben! Will man die Phänotypen vergleichen, so kann man das mit Hilfe des Strings, der von der folgenden Funktion geliefert wird.

5.3.7.4 `StartNode::PhenoTypeString()`

Diese Funktion liefert in einem String den Phänotypen des Individuums, wie er auch mit `TreeNode::Print()` ausgegeben würde. Die Lebensdauer des Strings ist begrenzt auf die Lebensdauer des Individuums. Der durch diesen String belegte Speicherplatz wird vom Individuum verwaltet, und sollte nicht "händisch" mit dem `delete`-Befehl freigegeben werden. Zum Zwecke einer bloßen Bildschirmausgabe ist der `Print()`-Befehl zu bevorzugen, da er mit weniger Rechenaufwand verbunden ist als der Aufruf von `StartNode::PhenoTypeString()`.

5.3.7.5 `StartNode::GenoTypeString()`

Diese Funktion liefert in einem String den Genotypen des Individuums, wobei es zwei Formate gibt, die durch den Parameter bestimmt werden: `SHORT` oder `1` bewirkt eine Ausgabe in der bereits besprochenen kurzen Stringrepräsentation (siehe Kapitel 5.3.5).

Die Angabe von LONG oder 0 liefert einen, etwas besser leserlichen, String in der langen Stringrepräsentation. Die Erzeugung eines solchen Strings in der langen Stringrepräsentation läßt sich wieder sehr gut an Hand des Beispiels aus Kapitel 5.3.5 veranschaulichen:

Gegeben sei wieder die Grammatik L_{XOR} aus Abbildung 3, sowie das Wort (NOT(D2)). Die entsprechende lange Stringrepräsentation erhält man, indem man den Ableitungsbaum vom Wurzelknoten her nachvollzieht. In jedem Schritt sucht man das linkeste, bislang noch nicht aufgelöste Nichtterminalsymbol. Vor dieses schreibt man die Anzahl der Symbole in der Ableitung dieses Symbols, nach dem Symbol die Ableitung selbst. Dies ist so lange zu machen, bis alle Nichtterminalsymbole ersetzt sind. Im gegebenen Beispiel funktioniert das folgendermaßen:

- “<fe>“ ⇒ “4<fe>"("<f1><fe>")"“
  (da <fe> ⇒ "("<f1><fe>")", das sind vier Symbole)
- “4<fe>"("<f1><fe>")"“ ⇒ “ 4<fe>"("1<f1>"NOT"<fe>")"“
  (da <f1> ⇒ "NOT", das ist ein Symbol)
- “4<fe>"("1<f1>"NOT"<fe>")"“ ⇒
  “4<fe>"("1<f1>"NOT"3<fe>"("<f0>")"")"“
  (da <fe> ⇒ "("<f0>")", das sind drei Symbole)
- “4<fe>"("1<f1>"NOT"3<fe>"("<f0>")"")"“ ⇒
  “4<fe>"("1<f1>"NOT"3<fe>"("1<f0>"D2"")"")"“
  (da <f0> ⇒ "D2", das ist ein Symbol)

Die lange Stringrepräsentation von “(NOT(D2))“ lautet für die angegebene Grammatik also: 4<fe>"("1<f1>"NOT"3<fe>"("1<f0>"D2"")"")".

5.3.7.6 StartNode::operator <<

Dieser Operator liefert den Genotyp des Individuums in der kurzen Stringrepräsentation als ostream. Für Bildschirmausgaben des Genotypen in dieser Darstellung ist der <<-Operator gegenüber der Anweisung cout << GenoTypeString(SHORT) zu bevorzugen, da er weniger rechenaufwendig ist.

5.3.7.7 TreeNode::Depth()

Diese Funktion liefert die Höhe des Ableitungsbaumes, und wurde lediglich zu experimentellen Zwecken implementiert. Achtung! Die Höhe des Baumes ist ungleich der Ableitungstiefe! Die Ableitungstiefe ist definiert als die Anzahl der Nichtterminalsymbole. Die Höhe des Baumes dagegen beschreibt die Anzahl der nötigen Ableitungsschritte zum tiefsten Knoten (das Niveau des tiefsten Knotens) [Duden Informatik, 1993, S. 56].

5.3.7.8 `TreeNode::NoOfSubTrees(SymbTabEntry*, unsigned int)`

Diese Funktion liefert die Anzahl von Ableitungsunterbäumen, die mit einem gewissen Symbol beginnen und eine gewisse gegebene maximale Ableitungstiefe nicht überschreiten. Der erste Parameter definiert das Symbol in seiner Token-Darstellung, der zweite die maximal zulässige Ableitungstiefe. Das Token eines bestimmten Symbols kann man im Kontext von `TreeNode` durch `TreeLanguage->Symbole(char*)` abfragen, wobei als Parameter die Stringdarstellung des Symbols (ohne spitze Klammern oder Anführungszeichen) anzugeben ist.

5.3.7.9 `Language::Print()`

Zur Überprüfung, ob eine bestimmte Sprachdefinitionsdatei vom Grammatikinterpreter richtig interpretiert wurde, ist es möglich, die aktuelle Sprachdefinition auf dem Bildschirm auszugeben. Das folgende Programm veranschaulicht dies:

```
     #include "ga.h"
     #include "xor.cc"
     #define MAXDERIVDEPTH 30
     #define POPSIZE 100
 5   #define HISTORY 5
     #define LANGUAGEDEF "xor.def"

     /*----------------------------------
        main()
10     ----------------------------------
     */
     int main()
     {
            Farm MainFarm(POPSIZE, HISTORY, SIZEUNIF, LANGUAGEDEF, MAXDERIVDEPTH);
15          MainFarm.L.Print();
     }
```

Dieses Programm führt zu folgendem Output:

```
Computing search space size up to n = 30...
done.
Generating population...
done...

LANGUAGE DEFINITION:

fe
  Derivation 1: ( f0 )
  Derivation 2: ( f1 fe )
  Derivation 3: ( f2 fe fe )
f0
  Derivation 1: D1
  Derivation 2: D2
f1
  Derivation 1: NOT
f2
  Derivation 1: OR
  Derivation 2: AND
(
)
```

```
D1
D2
NOT
OR
AND

Starting symbol is <fe>.
```

5.3.7.10 `Language::operator <<`

Eine ähnliche Funktion wie der vorige Befehl erfüllt der `<<`-Operator der Klasse `Language`. Die aktuelle Grammatik kann hiermit in ihrer Backus-Naur-Form ausgegeben werden. Ersetzt man im vorigen Programmlisting die Zeile 14 durch die Befehlsfolge

```
cout << MainFarm.L;
```

so ergibt sich folgender Output:

```
Computing search space size up to n = 30...
done.
Generating population...
done...
S := <fe> ;
<fe> := "("<f0>")" | "("<f1><fe>")" | "("<f2><fe><fe>")" ;
<f0> := "D1" | "D2" ;
<f1> := "NOT" ;
<f2> := "OR" | "AND" ;
```

5.3.7.11 `Language::PrintArbSearchSpace()`

Mit Hilfe dieser Funktion kann die zuvor mit der Hilfe der Funktion `Language::ComputeArbSearchSpace(int)` berechnete Suchraumtabelle ausgegeben werden. Die Berechnung erfolgt mit `double float`-Genauigkeit. Das folgende Programm zeigt die Anwendung:

```
     #include "ga.h"
     #include "xor.cc"
     #define MAXDERIVDEPTH 30
     #define POPSIZE 100
 5   #define HISTORY 5
     #define LANGUAGEDEF "xor.def"

     /*-----------------------------------
        main()
10     ----------------------------------
     */
     int main()
     {
             Farm MainFarm(POPSIZE, HISTORY, SIZEUNIF, LANGUAGEDEF, MAXDERIVDEPTH);
15           MainFarm.L.ComputeArbSearchSpace(80);
             MainFarm.L.PrintArbSearchSpace();
     }
```

Dieses Programm führt zu folgendem Output:

```
Computing search space size up to n = 30...
done.
Generating population...
done...
Table of search space sizes in n derivations,
calculated with long double - accuracy.

2: 2e+0
4: 2e+0
6: 1e+1
8: 2.6e+1
10: 1.14e+2
12: 4.02e+2
14: 1.722e+3
16: 6.89e+3
18: 2.9794e+4
20: 1.26626e+5
22: 5.56778e+5
24: 2.44614e+6
26: 1.09301e+7
28: 4.90279e+7
30: 2.22074e+8
32: 1.0106e+9
34: 4.62869e+9
36: 2.1292e+10
38: 9.84059e+10
40: 4.56508e+11
42: 2.12557e+12
44: 9.92833e+12
46: 4.65137e+13
48: 2.185e+14
50: 1.02899e+15
52: 4.85698e+15
54: 2.29747e+16
56: 1.08891e+17
58: 5.1705e+17
60: 2.45936e+18
62: 1.17168e+19
64: 5.59051e+19
66: 2.67122e+20
68: 1.27805e+21
70: 6.12251e+21
72: 2.93649e+22
74: 1.40999e+23
76: 6.77743e+23
78: 3.26101e+24
80: 1.57057e+25
```

5.3.7.12 Language::PrintExactSearchSpace()

Mit Hilfe dieser Funktion kann die zuvor mit der Hilfe der Funktion `Language::ComputeExactSearchSpace(int)` berechnete Suchraumtabelle ausgegeben werden. Die Berechnung erfolgt stets mit Genauigkeit bis zur letzten Stelle. Das folgende Programm zeigt die Anwendung:

```
#include "ga.h"
#include "xor.cc"
#define MAXDERIVDEPTH 30
```

```
    #define POPSIZE 100
 5  #define HISTORY 5
    #define LANGUAGEDEF "xor.def

    /*-----------------------------------
       main()
10    -----------------------------------
    */
    int main()
    {
            Farm MainFarm(POPSIZE, HISTORY, SIZEUNIF, LANGUAGEDEF, MAXDERIVDEPTH);
15          MainFarm.L.ComputeExactSearchSpace(80);
            MainFarm.L.PrintExactSearchSpace();
    }
```

Dieses Programm führt zu folgendem Output:

```
Computing search space size up to n = 30...
done.
Generating population...
done...
Table of exact search space sizes in n derivations:

2: 2
4: 2
6: 10
8: 26
10: 114
12: 402
14: 1722
16: 6890
18: 29794
20: 126626
22: 556778
24: 2446138
26: 10930130
28: 49027890
30: 222074010
32: 1010597130
34: 4628686530
36: 21291965250
38: 98405888970
40: 456507632730
42: 2125567786290
44: 9928334983890
46: 46513705289850
48: 218499787732266
50: 1028989150918434
52: 4856976677053922
54: 22974667695652522
56: 108890607838342010
58: 517050168784158866
60: 2459355330657065074
62: 11716796830974689882
64: 55905093913548022602
66: 267121812670113301890
68: 1278045900782875487874
70: 6122512900714569252746
72: 29364914198648673398938
74: 140998894983592158310898
76: 677742840795889628429330
78: 3261014636429428731290170
80: 15705696964184925804308330
```

6 Programmdokumentation

Dieses Kapitel wendet sich an Programmierer, die die interne Funktionsweise des Programms verstehen wollen, oder die vor haben, Erweiterungen zu implementieren. Alle Klassenfunktionen inklusive aller privater Funktionen sind hier aufgelistet und im Detail erklärt.

In Kapitel 6.1 werden die Klassen `Language`, `SymbTabEntry` und `ProdTabEntry` erläutert, welche für die Verwaltung der Grammatik zur Laufzeit zuständig sind. Das Unterkapitel 6.1.4 setzt sich mit der Implementierung der Suchraumberechnung in diesen Klassen auseinander und erklärt die Funktionsweise der `SuperFloat`-Arithmetik.

Kapitel 6.2 beschreibt die Klassen `TreeNode`, `StartNode` und `RootNode`, welche die Knotenobjekte der Ableitungsbäume bilden (siehe Abbildung 13 auf Seite 47).

In den Kapiteln 6.3 und 6.4 schließlich werden die Klassen `Farm` und `Population` erläutert, welche die Individuen beherbergen, und die genetischen Operatoren zur Verfügung stellen (siehe Abbildung 14 auf Seite 50).

6.1 Grammatikrepräsentation

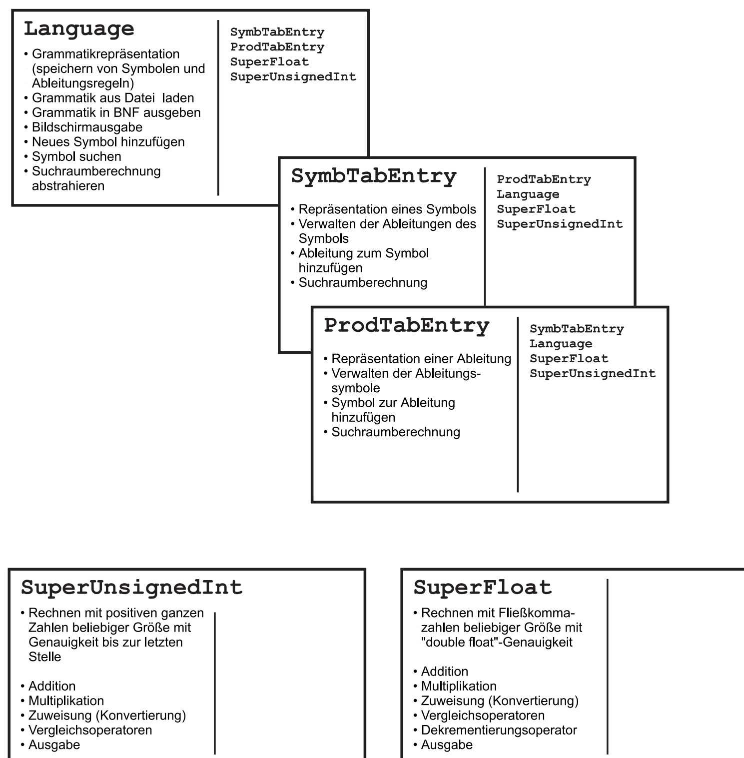


Abbildung 16: Die Klassen zur Grammatikrepräsentation

Wie in Abbildung 16 dargestellt, wird eine bestimmte Grammatik im Rahmen des vorliegenden Programmes mit Hilfe der Klassen `Language`, `SymbTabEntry` und `ProdTabEntry` sowie der in Abbildung 17 dargestellten Struktur `DerivSymbol` repräsentiert, wobei ein Objekt der Klasse `SymbTabEntry` ein bestimmtes Symbol der Grammatik darstellt, und ein Objekt der Klasse `ProdtabEntry` eine bestimmte Ableitung eines Symbols. Objekte der Struktur `DerivSymbol` stellen ein Symbol einer Ableitung dar. In Abbildung 17 ist der Zusammenhang zwischen der Klasse `Language` und der Klasse `SymbTabEntry` dargestellt. Ein Objekt der Klasse `Language` beinhaltet eine verkettete

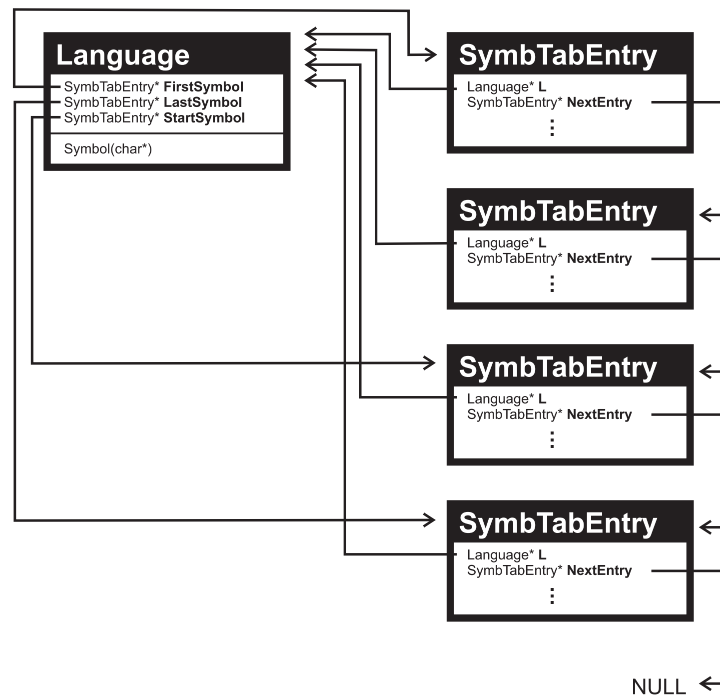


Abbildung 17: Zusammenhang zwischen den Klassen `Language` und `SymbTabEntry`

Liste von `SymbTabEntry`-Objekten, welche alle Symbole der Grammatik repräsentieren. Der Pointer `Language::FirstSymbol` zeigt auf den ersten Eintrag der verketteten Liste, der Pointer `Language::LastSymbol` auf den letzten. Der Pointer `StartSymbol` zeigt auf das das Startsymbol repräsentierende `SymbTabEntry`-Objekt der Grammatik. Die Verkettung erfolgt über die `SymbTabEntry::NextEntry`-Pointer, wobei der letzte Pointer auf `NULL` zeigt. Abbildung 17 zeigt also beispielsweise eine vier Symbole beinhaltende Grammatik, wobei das dritte Symbol der Liste das Startsymbol darstellt. Jedes Objekt der Klasse `SymbTabEntry` verfügt außerdem über einen Pointer `L`, der auf das zugehörige `Language`-Objekt verweist.

Die Methode `SymbTabEntry* Symbol(char*)` der Klasse `Language` erfüllt zwei Aufgaben:

- Einerseits kann mit Hilfe dieser Funktion ein zusätzliches Symbol der Grammatik angelegt werden, i.e.: an die verkettete `SymbTabEntry`-Liste wird ein entsprechen-

der Eintrag angefügt. Als Parameter muß die Textrepräsentation des Zeichens angegeben werden (ohne spitze Klammern oder Anführungszeichen). Zurückgegeben wird ein Pointer auf den neu angelegten Eintrag, der gewissermaßen als Token des Symbols betrachtet werden kann.

- Andererseits kann ein Pointer auf einen bestehenden Eintrag abgefragt werden, indem als Parameter die Textrepräsentation eines bereits angelegten Symbols der Grammatik angegeben wird. In diesem Fall wird kein Eintrag vorgenommen.

Abbildung 18 zeigt den Zusammenhang zwischen der Klasse `SymbTabEntry` und der Klasse `ProdTabEntry`. Man erkennt in dieser Abbildung, daß jedes Objekt der Klas-

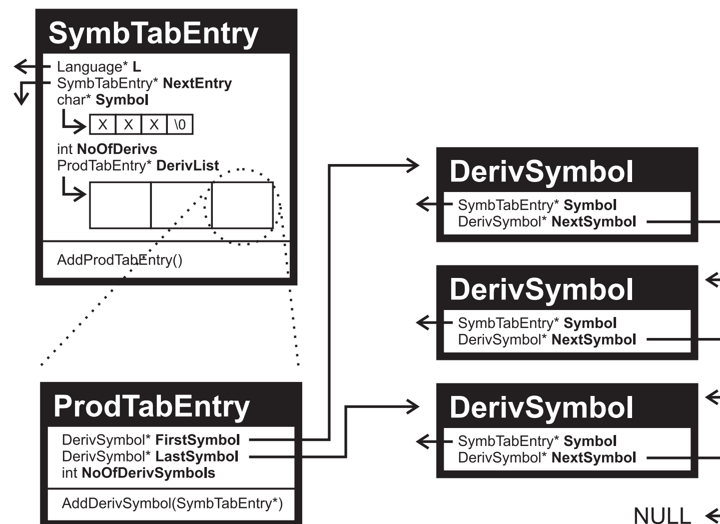


Abbildung 18: Die Klassen `SymbTabEntry`, `ProdTabEntry` und `DerivSymbol`

se `SymbTabEntry` neben den Pointern `L` und `NextEntry` auch ein Array `Symbol` mit der Textdarstellung des durch dieses Objekt repräsentierten Symbols beinhaltet. Jedem Symbol sind außerdem, so es sich um ein Nichtterminalsymbol handelt, ein oder mehrere mögliche Ableitungen zugeordnet. Die Anzahl der möglichen Ableitungen ist in der Variablen `NoOfDerivs` vermerkt, die Ableitungen selber im Array `DerivList`. Dieses Array beinhaltet für jede Ableitung ein Objekt der Klasse `ProdTabEntry`. Bei Terminalsymbolen ist `NoOfDerivs` gleich null. Mit Hilfe der Funktion `SymbTabEntry::AdProdTabEntry` ist es möglich, dem Array `DerivList` einen neuen Eintrag hinzuzufügen. Die Größe des Arrays wird dabei dynamisch verwaltet.

Ein Objekt der Klasse `ProdTabEntry` wiederum enthält eine verkettete Liste von Objekten der Struktur `DerivSymbol`. Jedes `DerivSymbol`-Objekt steht dabei für ein Symbol der Ableitung. Abbildung 18 zeigt also beispielsweise eine aus drei Symbolen bestehende Ableitung. Der Pointer `FirstSymbol` der Klasse `ProdTabEntry` zeigt dabei auf den ersten Eintrag der verketteten Liste, der `LastSymbol`-Pointer auf den letzten Eintrag. In `ProdTabEntry::NoOfDerivSymbols` ist vermerkt, aus wievielen Symbolen

die Ableitung besteht. Die Verkettung selbst erfolgt mit Hilfe des Pointers `NextSymbol` der Struktur `DerivSymbol`. Der letzte Pointer zeigt stets auf `NULL`. Das eigentliche Ableitungssymbol wird durch den Pointer `DerivSymbol::Symbol` repräsentiert. Mit Hilfe der Methode `AddDerivSymbol(SymbTabEntry*)` der Klasse `ProdTabEntry` ist es möglich, einer bestehenden Ableitung ein weiteres Symbol hinzuzufügen.

6.1.1 Die Language-Klasse

Die Klasse `Language` dient zur Repräsentation einer bestimmten Grammatik. Sie verwaltet alle Symbole und Ableitungen und bietet die Möglichkeit, eine in einem File zur Verfügung stehende Grammatik zu laden, bzw. die momentane Grammatik in der BNF auszugeben. Zudem bietet sie die oberste Schnittstelle zur Suchraumtabellenberechnung. Die Einbettung der Klasse `Language` in die Umgebung der anderen Klasen ist in Abbildung 16 auf Seite 70 dargestellt.

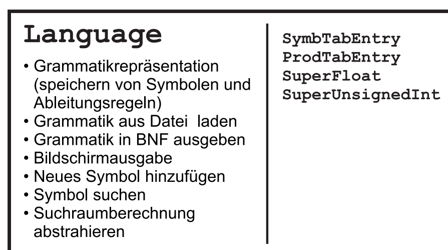


Abbildung 19: Die Klasse `Language`

6.1.1.1 Der Konstruktor Language()

```
   /*-------------------------------------------------------
     Language::Language()

     Constructor: Creating an empty grammar with no symbols.
 5   -------------------------------------------------------
   */
   Language::Language()
   {
           FirstSymbol = NULL;
10         LastSymbol = NULL;
           StartSymbol = NULL;
           ArbCard = NULL;
           ExactCard = NULL;
           ArbCardSize = -1;
15         ExactCardSize = -1;
   }
```

Der Konstruktor erzeugt eine leere Grammatikrepräsentation. Die in den Zeilen 12-15 deklarierten Variablen sind für die Suchraumberechnung wesentlich. `ArbCard` referenziert ein Array, welches eine Suchraumtabelle des Startsymbols der

Grammatik (siehe Kapitel 4.1.1), beinhalten kann. ArbCardSize gibt an, bis zu welcher Tiefe die Suchraumtabelle im Array `ArbCard` zur Verfgüng steht. Ist `ArbCard` gleich -1, dann ist bislang keine `ArbCard`-Suchraumtabelle berechnet worden. Das eben gesagte gilt auch für die Variablen `ExactCard` und `ExactCardSize`. Der Unterschied zwischen der Suchraumtabelle in `ArbCard` und der in `ExactCard` besteht darin, daß die Zahlen in ArbCard vom Typ `SuperFloat` sind, wohingegen das Array `ExactCard` Zahlen vom Typ `SuperUnsignedInt` enthält.

`SuperUnsignedInt` ist eine Klasse zum Rechnen mit beliebig langen, vorzeichenlosen Integerzahlen mit Genauigkeit bis zur letzten Stelle. `SuperFloat` kann ebenfalls beinahe beliebig große Zahlen verwalten, rechnet jedoch nur mit begrenzter („`long double`")-Genauigkeit. Die Suchraumberechnung wird im Kapitel 6.1.4 näher behandelt.

6.1.1.2 Die Funktion `SymbTabEntry* Symbol(char*)`

Diese Funktion prüft zuerst, ob es sich bei dem in seiner Textrepräsentation übergebenen Symbol um ein bereits existentes handelt. Ist dies der Fall, wird lediglich ein Pointer auf das entsprechende `SymbTabEntry`-Objekt zurückgegeben. Ansonsten wird ein neues `SymbTabEntry`-Objekt der Symbolliste hinzugefügt und ein Pointer auf dieses, neu angelegte Objekt zurückgegeben.

```
    /*-------------------------------------------------------
      SymbTabEntry* Language::Symbol(char* NewSymbol)

      Adding the NewSymbol given in its text-representation
 5    to the symbol-table and returning the token of symbol
      (i.e.: Pointer to Symbol-Table-Entry)

      If the NewSymbol already exists, and therefore is
      no new symbol at all, nothing is added, just the
10    token (pointer to Symbol-Table-Entry) is returned.
      -------------------------------------------------------
    */
    SymbTabEntry* Language::Symbol(char* NewSymbol)
    {
15          int SymbolAlreadyExists = NO;
            SymbTabEntry* CurrentSymbol = NULL;

            if(FirstSymbol!=NULL)
            {
                    CurrentSymbol = FirstSymbol;
20
                    while(strcmp(CurrentSymbol->Symbol,NewSymbol)!=EQUALSTRINGS &&
                    CurrentSymbol!=LastSymbol)
                            CurrentSymbol = CurrentSymbol->NextEntry;

25                  SymbolAlreadyExists=
                    (strcmp(CurrentSymbol->Symbol,NewSymbol)==EQUALSTRINGS);
            }

            if(!SymbolAlreadyExists)
30          {
                    SymbTabEntry* BeforeLast = LastSymbol;
                    LastSymbol = new SymbTabEntry(NewSymbol, this);

                if(FirstSymbol==NULL)
```

```
35                      FirstSymbol = LastSymbol;
                  else
                        BeforeLast->NextEntry = LastSymbol;

                  CurrentSymbol = LastSymbol;
40          }

            return(CurrentSymbol);
    }
```

In den Zeilen 18-27 wird geprüft, ob es sich bei dem Symbol `NewSymbol` tatsächlich um ein neues, oder um ein bereits bestehendes handelt: Wenn es überhaupt schon ein Symbol in der Liste gibt (Zeile 18), wird die Liste solange durchsucht, solange das Symbol noch nicht gefunden wurde (Zeile 21) und die Liste noch nicht zu Ende ist (Zeile 22). Das aktuelle Symbol ist also nach Durchlaufen dieser Schleife entweder das gesuchte, oder das letzte. In Zeile 25 wird dies festgestellt und das Flag `SymbolAlreadyExists` entsprechend gesetzt.

Handelt es sich um ein bereits bestehendes Symbol, werden die Befehle ab Zeile 30 nicht mehr ausgeführt, und der Pointer auf den ensprechenden `SymbTabEntry` wird zurückgegeben. Wurde das Symbol jedoch nicht gefunden, dann wird es in den Zeilen 30-39 der Liste hinzugefügt und ein Zeiger auf den neu angelegten Eintrag zurückgegeben. In Zeile 32 wird ein neues `SymbTabEntry`-Objekt erzeugt und als letztes Symbol der Liste deklariert. Handelt es sich um das erste Listenobjekt, wird der Pointer `Language::FirstSymbol` inititalisiert (Zeile 35), ansonsten wird der Verkettungszeiger `NextEntry` des bisher letzten Listenobjekts aktualisiert (Zeile 37).

6.1.1.3 Die Funktion `Load(char*)`

Mit Hilfe dieser Funktion kann eine Grammatik, die in einem File in ihrer Backus-Naur-Form (siehe Kapitel 5.1) zur Verfügung steht, in ein Objekt der Klasse `Language` geladen werden (siehe Kapitel 5.1). Eine eventuell bereits zuvor geladene Grammatik wird dabei gelöscht.

```
    /*---------------------------------------------------------
      int Language::Load(char* filename)

      Loading the language definition, as defined in the
 5    language definition file in the BNF-representation.
      Returning FALSE in case of an error.
      ---------------------------------------------------------
    */
    int Language::Load(char* filename)
10  {
            ProdTabEntry* Derivation = NULL;
            SymbTabEntry* CurrentSymbol = NULL;
            ifstream LangDefFile;
            char CurrentChar;
15          char* Buffer;
            int i;
            int warning;
            int line;
```

```
            int start;
20
            Clear();
            Buffer = new char[CHRBUFSIZE];
            line = 1;
            start = NO;
25
            LangDefFile.open(filename);
            if(!LangDefFile)
                    return(FALSE);

30          while(!LangDefFile.eof())
            {
                    CurrentChar = LangDefFile.get();
                    switch(CurrentChar)
                    {
35
                            case '\n':
                                    line++;
                                    break;

40                          case 'S':
                                    start = YES;
                                    break;

                            case ';':
45                                  start = NO;
                                    Derivation = NULL;
                                    break;

                            case'"':
50                                  CurrentChar = LangDefFile.get();
                                    i = 0;
                                    warning = NO;

                                    while(
55                                  CurrentChar!='"' &&
                                    CurrentChar!=';' &&
                                    CurrentChar!='>' &&
                                    !LangDefFile.eof())
                                    {
60                                          if(i<CHRBUFSIZE-1)
                                            {
                                                    Buffer[i] = CurrentChar;
                                                    i++;
                                            }
65                                          else
                                                    warning = YES;

                                            CurrentChar = LangDefFile.get();

70                                          if(CurrentChar=='\n')
                                                    line++;
                                    }

                                    if(warning)
75                                  {
                                            cerr << "Line " << line << ": ";
                                            cerr << "Warning! Symbol length ";
                                            cerr << "exceeds " << CHRBUFSIZE-1;
                                            cerr << " characters!" << endl;
80                                  }

                                    Buffer[i] = '\0';

                                    if(Derivation==NULL)
85                                  {
```

```
                                        CurrentSymbol = Symbol(Buffer);
                                        cerr << "Line " << line << ": ";
                                        cerr << "Warning! Non-terminal symbol " ;
                                        cerr << Buffer << " occurs as terminal ";
 90                                     cerr << "symbol!" << endl;
                                        cerr << "        <" << Buffer << "> ";
                                        cerr << "expected instead of ";
                                        cerr << '"' << Buffer << '"' << "." << endl;
                                }
 95                             else
                                        Derivation->AddDerivSymbol(Symbol(Buffer));

                                if(CurrentChar!='"')
                                {
100                                     start = NO;
                                        Derivation = NULL;
                                        cerr << "Line " << line << ": ";
                                        cerr << "Warning! " << '"' ;
                                        cerr << " expected!" << endl;
105                             }
                                break;

                        case '<':
                                CurrentChar = LangDefFile.get();
110                             i = 0;
                                warning = NO;

                                while(
                                CurrentChar!= '>' &&
115                             CurrentChar!= ';' &&
                                CurrentChar!= '"' &&
                                !LangDefFile.eof())
                                {
                                        if(i<CHRBUFSIZE-1)
120                                     {
                                                Buffer[i] = CurrentChar;
                                                i++;
                                        }
                                        else
125                                             warning=YES;

                                        CurrentChar = LangDefFile.get();

                                        if(CurrentChar=='\n')
130                                             line++;
                                }

                                if(warning)
                                {
135                                     cerr << "Line " << line << ": ";
                                        cerr << "Warning! Symbol length ";
                                        cerr << "exceeds " << CHRBUFSIZE-1;
                                        cerr << " characters!" << endl;
                                }
140
                                Buffer[i] = '\0';

                                if(Derivation==NULL)
                                {
145                                     if(start)
                                        {
                                                StartSymbol = Symbol(Buffer);
                                                start=NO;
                                        }
150                                     else
                                                CurrentSymbol = Symbol(Buffer);
                                }
```

```
                                else
                                        Derivation->AddDerivSymbol(Symbol(Buffer));
155
                                if(CurrentChar!='>')
                                {
                                        if(CurrentChar!='"')
                                        {
160                                             start = NO;
                                                Derivation = NULL;
                                        }
                                        cerr << "Line " << line << " ";
                                        cerr << "Warning! '>' expected!";
165                                     cerr << endl;
                                }
                                break;

                        case ':':
170                             CurrentChar = LangDefFile.get();
                                if(CurrentChar=='\n')
                                        line++;
                                if(CurrentChar==';')
                                {
175                                     Derivation = NULL;
                                        start = NO;
                                }
                                if( CurrentChar!='=' || (CurrentChar=='=' && start) )
                                      break;
180
                        case '|':
                                if(CurrentSymbol==NULL)
                                {
                                        cerr << "Line " << line << ": ";
190                                     cerr << "Error! Derivation can not ";
                                        cerr << "be related with any symbol!";
                                        cerr << endl;
                                }
                                else
195                                     Derivation =
                                        CurrentSymbol->AddProdTabEntry();

                                break;
                }
200     }

        if(StartSymbol==NULL && FirstSymbol!=NULL)
        {
                StartSymbol = FirstSymbol;
205             while(TRUE)
                {
                        if(StartSymbol->NextEntry==NULL||
                           StartSymbol->NoOfDerivs!=0)
                                break;
210                     else
                                StartSymbol = StartSymbol->NextEntry;
                }

                cerr << "Warning! No start-symbol defined." << endl;
215             cerr << "I declare the first non-terminal symbol <";
                cerr << StartSymbol->Symbol;
                cerr << "> as starting symbol." << endl;
        }

220     delete [] Buffer;

        return(FirstSymbol!=NULL);
    }
```

Die oben abgebildete `Load(char*)`-Funktion liest die Zeichen der im Parameter angegebenen Sprachdefinitionsdatei der Reihe nach ein, und ändert abhängig vom gelesenen Zeichen den Zustand, ähnlich einem Zustands-Automaten. Folgende Variablen (definiert in den Zeilen 11-19) sind zustandsdefinierend:

- `CurrentChar`: Momentan gelesenes Zeichen der Datei
- `CurrentSymbol`: Tokenrepräsentation des Nichtterminalsymbols im linken Teil der momentan behandelten Ableitungsregel
- `Derivation`: Pointer auf die momentan behandelte Ableitung des aktuellen, auf der linken Seite der Ableitungsregel stehenden Nichtterminalsymbols
- `start`: Ist `TRUE`, falls in dieser Zeile das Startsymbolzeichen `S` gefunden wurde.
- `line`: Hält die aktuelle Zeilennummer (wichtig für die Ausgabe von Fehlermeldungen)

Nach dem Löschen einer eventuell bereits geladenen Grammatik in Zeile 21, Reservierung eines Buffers für die zu ladenden Textrepräsentationen der Symbole in Zeile 22 und weiteren Inititalisierungen wird in Zeile 26 versucht, die angegebene Datei zu öffnen. Mißlingt dies, wird `FALSE` zurückgegeben und die Funktion abgebrochen (Zeile 28). Die eigentliche Einleseschleife beginnt in Zeile 30, und wird bis zum Ende des Files ausgeführt. Folgende Zeichen sind zustandsverändernd:

- **Zeilenwechsel** (Zeichen "`\n`"): Zeilen 36-38
  Bei einem Zeilenwechsel wird einfach die in `line` vermerkte Zeilenanzahl um eins erhöht.
- **Startsymbol** (Zeichen "`S`": Zeilen 40-42
  Bei einem Zeichen `S`, welches den Beginn einer Startsymboldefintion anzeigt, wird der Merker `start` auf `YES` gesetzt. Dieser Merker ist wesentlich für die Behandlung der nachfolgenden Zeichen dieser Zeile.
- **Strichpunkt** (Zeichen "`;`"): Zeilen 44-47
  Ein Strichpunkt schließt eine Regel der BNF ab. Der Merker `start` wird auf `NO` gesetzt, da eine eventuelle Startsymboldefinition hier endet. Ebenso wird die Variable `Derivation` auf NULL gesetzt, da ein eventuell folgendes Symbol kein weiteres Symbol der momentanen Ableitung darstellt.
- **Anführungszeichen** (Zeichen "`"`"): Zeilen 50-106
  Wird ein Anführungszeichen gefunden, so werden alle folgenden Zeichen bis zum nächsten Anführungszeichen, Strichpunkt oder bis zur "spitzen Klammer zu", höchstens jedoch `CHRBUFSIZE-1` Zeichen in den Buffer eingelesen (Zeilen 54-71).

CHRBUFSIZE wurde im Headerfile mit 256 definiert. Im Falle eines Bufferüberlaufes wird der Merker warning gesetzt, und in den Zeilen 74-81 eine entsprechende Warnung ausgegeben. In Zeile 85 wird geprüft, ob es bereits eine aktuelle Ableitung gibt, der das eingelesene Symbol hinzugefügt werden kann. Ist dies nicht der Fall, wird das Symbol in Zeile 87, falls neu, der Liste der Symbole hinzugefügt und, unter Ausgabe einer entsprechenden Warnung, zum aktuellen, auf der linken Seite der Ableitung stehenden, Nichtterminalsymbol erklärt. Gibt es aber bereits ein aktuelles Nichtterminalsybol mit einer in Derivation referenzierten Ableitung (Normalfall), dann wird das neue Symbol dieser Ableitung hinzugefügt (Zeile 96). In den Zeile 98-105 schließlich wird noch geprüft, ob das aktuelle, zum Abschluß des Symbols geführt habende Zeichen ungleich einem Anführungszeichen ist (auch eine "spitze Klammer zu" oder ein Strichpunkt schließen ja ein Symbol ab). In diesem Fall wird eine entsprechende Warnung ausgegeben.

- **"spitze Klammer auf"** (Zeichen "<"): Zeilen 108-167
  Wird eine "spitze Klammer auf" gefunden, so werden alle folgenden Zeichen bis zur nächsten "spitzen Klammer zu" (>), bis zum nächsten Anführungszeichen oder bis zum nächsten Strichpunkt, höchstens jedoch CHRBUFSIZE-1 Zeichen in den Buffer eingelesen (Zeilen 113-131). Ein Bufferüberlauf wird wie im vorigen Fall behandelt. Ist das nun gefundene Nichtterminalsymbol nicht der rechte Teil einer momentan behandelten Ableitungsregel (Zeile 143), so wird es, falls neu, der Liste der Symbole hinzugefügt, und zum aktuellen, auf der linken Seite einer Ableitungsregel stehenden Symbol erklärt (Zeile 150). Wenn zuvor ein S eingelesen wurde und das Flag start daher gesetzt ist (Zeile 145), handelt es sich jedoch um einen Ausnahmefall. Dann ist das eingelesene Symbol nicht Teil einer Ableitung, sondern das Startsymbol (Zeile 146). Für den Fall, daß das eingelesene Symbol auf der rechten Seite einer Ableitungsregel steht, gilt, daß es der momentanen Ableitung hinzugefügt wird (Zeile 154). In den Zeilen 157-167 schließlich wird noch geprüft, ob das aktuelle, letzte Zeichen ungleich einer "spitzen Klammer zu" ist (auch ein Anführungszeichen oder ein Strichpunkt schließen ja ein Symbol ab). In diesem Fall wird eine entsprechende Warnung ausgegeben.

- **Doppelpunkt** (Zeichen ":"): Zeilen 169-179
  Wird ein Doppelpunkt gefunden, so wird das nächste Zeichen eingelesen. In der Zeile 173 wird geprüft, ob es sich um einen Strichpunkt handelt. Ein Strichpunkt schließt eine Regel der BNF ab. Der Merker start wird auf NO gesetzt, da eine eventuelle Startsymboldefinition hier endet. Ebenso wird die Variable Derivation auf NULL gesetzt, da ein eventuell folgendes Symbol kein weiteres Symbol der momentanen Ableitung darstellt. Ist das eingelesene Zeichen ungleich einem Gleichheitszeichen (Zeichenfolge :=) oder ist zwar es ein Gleichheitszeichen, aber es handelt sich um eine Startsymboldefinition (Zeichenfolge "S:=", Flag start ist gesetzt), so wird abgebrochen (Zeilen 179-180). Ansonsten wird mit den Anweisungen für den "senkrechten Strich" weitergemacht, da nun die Symbolfolge einer neuen Ableitung für das aktuelle, auf der linken Seite der Ableitungsregel stehende Nichtterminalsymbol zu erwarten ist:

- **senkrechter Strich** (Zeichen "|"): Zeilen 181-198
  Wird ein "senkrechter Strich" gefunden (oder das Programm durch Übergang von der Zeichenfolge "`:=`" hier fortgesetzt), so wird zunächst in Zeile 182 geprüft, ob es ein aktuelles, auf der linken Seite der Ableitungsregel stehendes Nichtterminalsymbol gibt. Ist dies nicht der Fall, so handelt es sich um eine unvollständige Ableitungsregel ohne linke Seite. Eine entsprechende Fehlermeldung wird in den Zeilen 189-192 ausgegeben. Ist alles in Ordnung, wird eine neue Ableitung zum aktuellen, auf der linken Seite der Ableitungsregel stehenden Nichtterminalsymbol, angelegt (Zeilen 195-196).

In Zeile 202 wird geprüft, ob zwar Symbole eingelesen wurden, jedoch keine Startsymboldefinition gefunden wurde. Ist dies der Fall, so wird in den Zeile 205-212 die Symbolliste bis zum ersten Nichtterminalsymbol durchsucht, und dieses Symbol, unter Ausgabe einer Warnung (Zeilen 214-217), zum Startsymbol erklärt. Gibt es zumindest ein Symbol in der Symbolliste, wird die Funktion unter Rückgabe von `TRUE` abgeschlossen (Zeile 222).

6.1.1.4 Die Funktion `Print()`

Die Funktion `Print()` der Klasse `Language` druckt die geladene Grammatik auf dem Bildschirm aus (siehe Beispiel auf Seite 66). Es werden zuerst alle Nichtterminalsymbole und ihre Ableitungen, anschließend die Terminalsymbole ausgedruckt.

```
    /*-------------------------------------------------------
      void Language::Print()

      Printing out the language definition -
 5    first the non-terminal symbols and its derivations,
      followed by the terminal symbols.
      -------------------------------------------------------
    */
    void Language::Print()
10  {
            int i;
            int j;
            DerivSymbol* CurrentDerivSymbol;

15          cout << endl << "LANGUAGE DEFINITION:" << endl << endl;
            if(FirstSymbol==NULL)
                    cout << "Error: empty Language!" << endl;
            else
            {
20                  for(j=0;j<=1;j++)
                    {
                            SymbTabEntry* CurrentSymbol = FirstSymbol;
                            while(TRUE)
                            {
25                                  if(
                                    (j==0&&CurrentSymbol->NoOfDerivs>0) ||
                                    (j==1&&CurrentSymbol->NoOfDerivs==0))
                                    {
                                            cout << CurrentSymbol->Symbol << endl;
30                                          for(i=0;i<CurrentSymbol->NoOfDerivs;i++)
```

```
                                        {
                                                cout << "  Derivation " << i+1 << ": ";

                                                if((CurrentSymbol->DerivList+i)->
35                                              FirstSymbol==NULL)
                                                        cout << "empty derivation." <<
                                                        endl;
                                                else
                                                {
40                                                      CurrentDerivSymbol =
                                                        (CurrentSymbol->DerivList+i)->
                                                        FirstSymbol;

                                                        while(TRUE)
45                                                      {
                                                                cout <<
                                                                CurrentDerivSymbol->
                                                                Symbol->Symbol << " ";

50                                                              if(
                                                                CurrentDerivSymbol->
                                                                NextSymbol==NULL)
                                                                        break;
                                                                else
55                                                                      CurrentDerivSymbol =
                                                                        CurrentDerivSymbol->
                                                                        NextSymbol;
                                                        }
                                                        cout << endl;
60                                              }
                                        }
                                }

                                if(CurrentSymbol->NextEntry==NULL)
65                                      break;
                                else
                                        CurrentSymbol = CurrentSymbol->NextEntry;
                        }
                }
70
                if(StartSymbol==NULL)
                        cout << endl << "No start-symbol defined!" << endl;
                else
                {
75                      cout << endl << "Starting symbol is <";
                        cout << StartSymbol->Symbol << ">." << endl << endl;
                }
        }
}
```

In Zeile 16 wird zunächst geprüft, ob es überhaupt auszudruckende Symbole gibt. Ist dies nicht der Fall, wird eine entsprechende Fehlermeldung ausgegeben. Die Schleife beginnend in Zeile 20 bewirkt, daß die Liste aller Symbole zweimal durchlaufen wird. Einmal zur Suche nach Nichtterminalsymbolen, einmal zur Suche nach Terminalsymbolen. Die Bedingung in den Zeilen 25-27 stellt sicher, daß beim ersten Durchlauf auch tatsächlich nur Nichtterminalsymbole, beim zweiten Durchlauf nur Terminalsymbole zum Ausdruck kommen. In der in Zeile 23 beginnenden Schleife wird die verkettete Liste der vorhandenen Symbole (`SymTabEntry`-Objekte) solange durchgegangen, bis in Zeile 64 der letzte Listeneintrag gefunden wird. In Zeile 29 wird das momentane Symbol ausgegeben. Die in Zeile 30 beginnende Schleife wird so oft durchlaufen, wie

es für dieses Symbol Ableitungen gibt. In Zeile 35 wird geprüft, ob es sich womöglich um eine Ableitung ohne ein einziges Symbol handelt, was ein fehlerhafter Zustand wäre. In diesem Fall wird eine entsprechende Fehlermeldung auf dem Bildschirm ausgegeben (Zeilen 36-37). Andernfalls wird in den Zeilen 44-58 die verkettete Liste der Ableitungssymbole solange durchgegegangen und die Symbole ausgegeben, bis der `DerivSymbol::NextSymbol`-Pointer auf NULL zeigt (letztes Symbol der Ableitung). In Zeile 71 wird geprüft, ob ein Startsymbol definiert ist. Sollte dies nicht der Fall sein (fehlerhafter Zustand), so wird eine entsprechende Fehler meldung ausgegeben (Zeile 72). Ansonsten wird das Startsymbol ausgegeben (Zeilen 75-76).

6.1.1.5 Der Operator <<

Der Operator `<<` gibt die Gammatik in ihrer BNF als `ostream` zurück. Das Beispiel auf Seite 67 demonstriert die Anwendung.

```
    /*-------------------------------------------------------
      ostream& operator << (ostream& s, Language& TheGrammar)

      Friend of Language
 5
      Returning the Language in its BNF-representation
      as ostream.
      -------------------------------------------------------
    */
10  ostream& operator <<(ostream& s, Language& TheGrammar)
    {
            int i;
            int j;
            int k;
15          SymbTabEntry* CurrentSymbol;
            DerivSymbol* CurrentDerivSymbol;

            if(TheGrammar.StartSymbol!=NULL)
            {
20                  s << 'S' << ' ' << ':' << '=' << ' ' << '<';
                    for(i=0;i<strlen(TheGrammar.StartSymbol->Symbol);i++)
                            s << TheGrammar.StartSymbol->Symbol[i];
                    s << '>' << ' ' << ';' << endl;

25                  CurrentSymbol = TheGrammar.FirstSymbol;
                    while(TRUE)
                    {
                            if(CurrentSymbol->NoOfDerivs)
                            {
30                                  s << '<';
                                    for(i=0;i<strlen(CurrentSymbol->Symbol);i++)
                                            s << CurrentSymbol->Symbol[i];
                                    s << '>' << ' ' << ':' << '=' << ' ';

35                                  for(j=0;j<CurrentSymbol->NoOfDerivs;j++)
                                    {
                                            CurrentDerivSymbol =
                                            CurrentSymbol->DerivList[j].FirstSymbol;

40                                          for(k=0;
                                            k<CurrentSymbol- >DerivList[j].NoOfDerivSymbols;
                                            k++)
```

```
                          {
                              if(CurrentDerivSymbol->Symbol->
45                            NoOfDerivs==0)
                                  s << '"';
                              else
                                  s << '<';

50                            for(i=0;
                              i<strlen(CurrentDerivSymbol->
                              Symbol->Symbol);
                              i++)
                                  s << CurrentDerivSymbol->
55                                Symbol->Symbol[i];

                              if(CurrentDerivSymbol->
                              Symbol->NoOfDerivs==0)
                                  s << '"';
60                            else
                                  s << '>';

                              CurrentDerivSymbol =
                              CurrentDerivSymbol->NextSymbol;
65                        }

                          if(j+1==CurrentSymbol->NoOfDerivs)
                              s << ' ' << ';' << endl;
                          else
70                            s << ' ' << '|' << ' ';
                      }
                  }

                  if(CurrentSymbol==TheGrammar.LastSymbol)
75                    break;

                  CurrentSymbol = CurrentSymbol->NextEntry;
              }
          }
80
          return(s);
      }
```

In Zeile 18 wird zunächst geprüft, ob es überhaupt ein Startsymbol gibt. Nur wenn dies der Fall ist, wird das Programm fortgesetzt. In den Zeilen 20-23 wird dann die BNF-konforme Startsymboldeklaration (“`S :=` <*startsymbol*>“) ausgegeben. In der in Zeile 26 beginnenden Schleife wird die verkettete Liste der vorhandenen Symbole (`SymbTabEntry`-Objekte) solange durchlaufen, bis in Zeile 74 der letzte Listeneintrag gefunden wurde. In Zeile 28 wird geprüft, ob es sich um ein Nichtterminalsymbol handelt. Nur in diesem Fall wird mit der Ausgabe einer Ableitungsregel begonnen. In den Zeilen 30-33 wird die linke Seite der Ableitungsregel BNF-konform ausgegeben (“<*symbol*>`:=`“). Die in Zeile 35 beginnende Schleife wird hierauf so oft durchlaufen, wie es Ableitungen für dieses Symbol gibt. Für jede Ableitung wird die Schleife beginnend ab Zeile 40 durchlaufen, welche die Ableitung ausgibt, indem sie die verkettete Liste der Ableitungssymbole (`DerivSymbol`-Objekte) so oft durchläuft, wie es Ableitungssymbole gibt. In Zeile 67 wird geprüft, ob es sich um die letzte Ableitung des auf der linken Seite der Ableitungsregel stehenden Nichtterminalsymbols gehandelt hat, oder nicht. Im ersten Fall wird die Regel mit einem Strichpunkt und einem Zeilenwechsel abgeschlossen (Zeile 68), im zweiten Fall wird zur Trennung der Ableitungen ein senkrechter Strich ausgegeben (Zeile 70).

6.1.1.6 Der Language()-Destruktor und die Funktion Clear()

Der Destruktor ~Language() greift, wie das folgende Listing zeigt, auf die Funktion Clear() zurück.

```
    /*-----------------------------------------------------
      Language::~Language()

      Destructor: Deleting the whole grammar by using Clear()
 5    -----------------------------------------------------
    */
    Language::~Language()
    {
            Clear();
10  }
```

Die Funktion Clear():

```
    /*-----------------------------------------------------
      void Language::Clear()

      Deleting the whole Grammar
 5    -----------------------------------------------------
    */
    void Language::Clear()
    {
            if(FirstSymbol!=NULL)
10          {
                    SymbTabEntry* CurrentSymbol = FirstSymbol;
                    SymbTabEntry* NextSymbol;

                    while(CurrentSymbol!=LastSymbol)
15                  {
                            NextSymbol = CurrentSymbol->NextEntry;
                            delete CurrentSymbol;
                            CurrentSymbol = NextSymbol;
                    }
20                  delete CurrentSymbol;
            }
    }
```

In Zeile 9 wird zunächst geprüft, ob es überhaupt zu löschende Symbole (SymbTabEntry-Objekte) gibt. Ist dies der Fall, so wird in der in Zeile 14 beginnenden Schleife die verkettete Liste der Grammatiksymbole durchlaufen, und jedes Symbol einzeln gelöscht.

6.1.2 Die SymbTabEntry-Klasse

Die Klasse SymbTabEntry dient zur Repräsentation eines bestimmten Symbols der Grammatik. Sie verwaltet alle Ableitungen des Symbols und bietet die Möglichkeit, weitere Ableitungen hinzuzufügen. Zudem bietet sie Funktionen zur Suchraumtabellenberechnung. Die Einbettung der Klasse SymbTabEntry in die Umgebung der anderen Klasen ist in Abbildung 16 auf Seite 70 dargestellt.

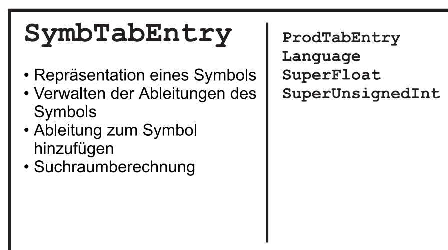


Abbildung 20: Die Klasse `SymbTabEntry`

6.1.2.1 Der Leerkonstruktor `SymbTabEntry()`

Der Leerkonstruktor der Klasse `SymbTabEntry` wird im vorliegenden Programm nie aufgerufen, und wurde nur vollständigkeitshalber implemetiert. Er definiert, daß das neu erzeugte `SymbTabEntry`-Objekt keiner Grammatik zugeordnet ist (Zeile 12), keine Ableitungen besitzt (Zeilen 11 und 13) und als schriftliche Symbolrepräsentation den String "empty symbol" zugeordnet bekommt (Zeile 10). Für die in den Zeilen 15-18 initialisierten Variablen gilt sinngemäß das in Kapitel 6.1.1.1 auf Seite 73 gesagte.

```
    /*-------------------------------------------------------
      SymbTabEntry::SymbTabEntry()

      Constructor: Creating an empty entry of the Symbol-Table
 5    -------------------------------------------------------
    */

    SymbTabEntry::SymbTabEntry()
    {
10          Symbol="empty symbol";
            DerivList=NULL;
            L = NULL;
            NoOfDerivs=0;
            NextEntry=NULL;
15          ExactCard = NULL;
            ExactCardSize = -1;
            ArbCard = NULL;
            ArbCardSize = -1;
    }
```

6.1.2.2 Konstruktor `SymbTabEntry(char*, Language*)`

Dieser Konstruktor sorgt dafür, daß das neu erzeugte `SymbTabEntry`-Objekt, die durch den `SymbolLanguage`-Pointer referenzierte Grammatik zugeordnet bekommt (Zeile 15), keine Ableitungen besitzt (Zeilen 17 und 18) und als schriftliche Symbolrepräsentation den String `NewSymbol` erhält (Zeile 14). Für die in den Zeilen 20-23 initialisierten Variablen gilt sinngemäß das in Kapitel 6.1.1.1 auf Seite 73 gesagte.

```
    /*-------------------------------------------------------
      SymbTabEntry::SymbTabEntry(char* NewSymbol , Language* SymbolLanguage)

      Constructor: Creating an entry of the Symbol-Table
 5    with the written representation of the Symbol,
```

```
      achieved in NewSymbol. No derivations of this symbol
      are created by this constructor, and the NextEntry-
      Pointer is set to NULL.
      -------------------------------------------------------
10  */

    SymbTabEntry::SymbTabEntry(char* NewSymbol, Language* SymbolLanguage)
    {
            Symbol=new char[strlen(NewSymbol)+1];
15          L = SymbolLanguage;
            strcpy(Symbol,NewSymbol);
            DerivList=NULL;
            NoOfDerivs=0;
            NextEntry=NULL;
20          ExactCard = NULL;
            ExactCardSize = -1;
            ArbCard = NULL;
            ArbCardSize = -1;
    }
```

6.1.2.3 Die Funktion `AddProdTabEntry()`

Diese Funktion fügt dem durch das `SymbTabEntry`-Objekt dargestellten Symbol eine Ableitung hinzu. Ist das zur Verfügung stehende Array `DerivList` noch groß genug, so geschieht dies einfach durch Inkrementierung von `NoOfDerivs`. Ist jedoch der Platz im zur Verfügung stehenden Array erschöpft, so wird zuvor ein neues, größeres Array erzeugt, das alte Array in das neue umkopiert und das alte Array anschließend vernichtet. Durch diese Strategie ist sichergestellt, daß zur Laufzeit alle Ableitungen des Symbols hintereinander in einem Array zur Verfügung stehen. Ein schneller wahlfreier Zugriff auf eine bestimmte Ableitung ist somit gewährleistet.

```
    /*-------------------------------------------------------
      ProdTabEntry* SymbTabEntry::AddProdTabEntry()

      Adding an entry to the production-table. All entries
 5    are held in a line in an array, rather than a chained list.

      If the allocated space runs out, a lager array is
      allocated, the old entries are copied to the new
      space, and the old one is deleted.
10
      This enables a very fast access to all derivations
      during run-time.
      -------------------------------------------------------
    */
15
    ProdTabEntry* SymbTabEntry::AddProdTabEntry()
    {
            ProdTabEntry* NewProdTab;
            int i;
20
            if(NoOfDerivs%PTBUFF == 0)
            {
                    NewProdTab = new ProdTabEntry[NoOfDerivs+PTBUFF]();

25                  for(i=0;i<NoOfDerivs;i++)
                            NewProdTab[i] = DerivList[i];

                    if(NoOfDerivs!=0)
```

```
                        delete [] DerivList;
30
                DerivList = NewProdTab;
        }

        NoOfDerivs++;
35      return(DerivList+NoOfDerivs-1);
}
```

In Zeile 21 wird zunächst geprüft, ob die Anzahl der bereits gespeicherten Ableitungen ein vielfaches von `PTBUFF` beträgt (`PTBUFF` ist im vorliegenden Programm mit 4 definiert). Ist dies der Fall, dann ist das zur Verfügung stehende `DerivList`-Array voll. Beim ersten Aufruf gibt es noch kein `DerivList`-Array, die Anzahl der gespeicherten Ableitungen ist dann genau null mal `PTBUFF`, was zur Folge hat, daß die folgenden Befehle ebenfalls ausgeführt werden. In Zeile 23 wird ein neues, um `PTBUFF` größeres Array als das bisherige erzeugt. In den Zeilen 25 und 26 wird der Inhalt des alten Arrays (so es eines gab) in das neue Array umkopiert. In Zeile 28 und 29 wird geprüft, ob es nun ein altes Array gab, und dieses wird gegebenfalls vernichtet. Schließlich wird in Zeile 34 der Inhalt der Variable `NoOfDerivs`, welche die Anzahl der Ableitungen beschreibt, um eins erhöht.

6.1.2.4 Der Destruktor von `SymbTabEntry()`

Der Destruktor der Klasse `SymbTabEntry` löscht zunächst die Textrepräsentation des Symbols (Zeile 14). Dann werden die im Array `DerivList` gehaltenen Ableitungen gelöscht, wobei hierzu eine spezielle Vorgehensweise nötig ist. Damit nicht beim Zerstören des alten `DerivList`-Arrays in der Funktion `AddProdTabEntry` die mit den `ProdTabEntry`-Objekten verknüpften `DerivSymbol`-Objekte mitgelöscht werden, wurde ein Teil des Destruktors der Klasse `ProdTabEntry` in eine eigene Funktion `ProdTabEntry::Release()` ausgelagert. Diese muß vor einer beabsichtigten Vernichtung eines Objekts der Klasse `ProdTabEntry` aufgerufen werden (Zeilen 16 und 17). In den Zeilen 21-25 werden noch die eventuell vorhandenen Suchraumtabellen vernichtet.

```
/*-------------------------------------------------------
  SymbTabEntry::~SymbTabEntry()

  Destructor: Deleting the symbol-string, the
5 ProductionTable-entries, and, before that,
  the derivations related with the ProductionTable-
  Entries.
  -------------------------------------------------------
*/
10
SymbTabEntry::~SymbTabEntry()
{
        int i;
        delete [] Symbol;
15
        for(i=0;i<NoOfDerivs;i++)
                DerivList[i].Release();

        delete [] DerivList;
```

```
20
            if(ExactCardSize>-1)
                    delete [] ExactCard;

            if(ArbCardSize>-1)
25                  delete [] ArbCard;
    }
```

6.1.3 Die ProdTabEntry-Klasse

Die Klasse **ProdTabEntry** dient zur Repräsentation einer bestimmten Ableitung eines Symbols. Sie verwaltet alle Symbole der Ableitung und bietet die Möglichkeit, weitere Symbole hinzuzufügen. Zudem bietet sie Funktionen zur Suchraumtabellenberechnung. Die Einbettung der Klasse **ProdTabEntry** in die Umgebung der anderen Klasen ist in Abbildung 16 auf Seite 70 dargestellt.

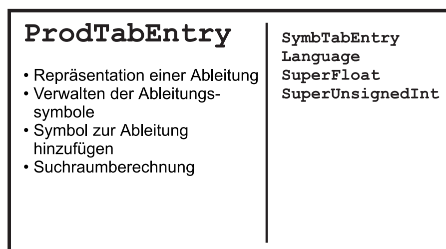


Abbildung 21: Die Klasse **ProdTabEntry**

6.1.3.1 Der Konstruktor ProdTabEntry()

Dieser Konstruktor legt fest, daß ein neues Objekt der Klasse **ProdTabEntry** eine "leere Ableitung" beinhaltet: Die Anzahl der Ableitungssymbole ist null (Zeile 13), und die verkettete Liste von Ableitungssymbolen (**DerivSymbol**-Objekten) ist leer (Zeilen 11 und 12). Für die in den Zeilen 15-18 initialisierten Variablen gilt sinngemäß das Kapitel 6.1.1.1 auf Seite 73 gesagte.

```
    /*-------------------------------------------------------
      ProdTabEntry::ProdTabEntry()

      Constructor: Creating an empty production-table
 5    entry
      -------------------------------------------------------
    */

    ProdTabEntry::ProdTabEntry()
10  {
            FirstSymbol = NULL;
            LastSymbol = NULL;
            NoOfDerivSymbols = 0;
            ExactCard = NULL;
15          ExactCardSize = -1;
            ArbCard = NULL;
            ArbCardSize = -1;
    }
```

6.1.3.2 AddDerivSymbol(SymbTabEntry* NewSymbol)

Diese Funktion fügt einer bestehenden Ableitung (dargestellt durch das Objekt der Klasse **ProdTabEntry**) das Ableitungssymbol **NewSymbol** hinzu. Dies geschieht durch Erstellung eines neuen Objekts der Klasse **DerivSymbol**, welches der verketteten Liste von Ableitungssymbolen hinzugefügt wird.

```
    /*------------------------------------------------------
      DerivSymbol* ProdTabEntry::AddDerivSymbol(SymbTabEntry* NewSymbol)

      Adding a symbol given in its token-representation
 5    (i.e.: pointer to the symbol-table) to this derivation
      ------------------------------------------------------
    */
    DerivSymbol* ProdTabEntry::AddDerivSymbol(SymbTabEntry* NewSymbol)
    {
10          DerivSymbol* BeforeLast;

            NoOfDerivSymbols++;
            BeforeLast = LastSymbol;
            LastSymbol = new DerivSymbol;
15          LastSymbol->Symbol = NewSymbol;
            LastSymbol->NextSymbol = NULL;

            if(FirstSymbol==NULL)
                    FirstSymbol = LastSymbol;
20          else
                    BeforeLast->NextSymbol = LastSymbol;

            if(NewSymbol->L!=NULL)
            {
25                  NewSymbol->L->ArbCard = NULL;
                    NewSymbol->L->ArbCardSize = -1;
                    NewSymbol->L->ExactCard = NULL;
                    NewSymbol->L->ExactCardSize = -1;
             }
30
            return(LastSymbol);
    }
```

In Zeile 12 wird zunächst der **NoOfDerivSymbols**-Zähler, der die Anzahl der Symbole in der Ableitung anzeigt, um eins erhöht. In den Zeilen 14-16 wird ein neues Objekt der Struktur **DerivSymbol** erzeugt, welches einen Pointer auf das neue Symbol **NewSymbol** zugewiesen bekommt und dessen Verkettungszeiger **NextSymbol** auf NULL (letztes Listenelement) gesetzt wird. In Zeile 18 wird geprüft, ob es sich um das erste Symbol der Ableitung handelt. In diesem Fall wird der Zeiger auf das erste Listenelement **FirstSymbol** entsprechend gesetzt (Zeile 19). Andernfalls wird der Verkettungszeiger des bislang letzten Listenelements auf das neue, nunmehr letzte Listenelement gesetzt (Zeile 21). In Zeile 23 wird geprüft, ob das neu hinzugekommene Symbol der Ableitung einer Grammatik, also einem Objekt der Klasse **Language** zugeordnet ist (Regelfall). Ist dies so, dann werden die eventuell vorhandenen Suchraumtabellen dieser Grammatik gelöscht, da nach dem Hinzufügen des neuen Ableitungssymbols auf sie kein Verlaß mehr ist (Zeilen 25-28).

6.1.3.3 Der Destruktor von ProdTabEntry() und die Funktion Release()

Damit nicht beim Zerstören des alten DerivList-Arrays in der Funktion SymbTabEntry::AddProdTabEntry die mit den ProdTabEntry-Objekten verknüpften DerivSymbol-Objekte mitgelöscht werden, wurde ein Teil des Destruktors der Klasse ProdTabEntry in eine eigene Funktion ProdTabEntry::Release() ausgelagert. Diese muß vor einer beabsichtigten Vernichtung eines Objekts der Klasse ProdTabEntry aufgerufen werden. Der eigentliche Destruktor vernichtet lediglich eventuell vorhandene Suchraumtabellen.

```
    /*-------------------------------------------------------
      ProdTabEntry::~ProdTabEntry()

      Destructor: Deleting cached search-space-size vectors.
 5    -------------------------------------------------------
    */
    ProdTabEntry::~ProdTabEntry()
    {
            if(ExactCardSize>-1)
10                  delete [] ExactCard;

            if(ArbCardSize>-1)
                    delete [] ArbCard;
    }
```

Die Funktion Release löscht darüber hinaus auch alle Symbole der Ableitung, die durch die verkettete Liste, bestehend aus Objekten der Klasse DerivSymbol, dargestellt werden.

```
    /*-------------------------------------------------------
      void ProdTabEntry::Release()

      Kind of destructor: Deletes the whole derivation.
 5    To be called before destruction, if wished.
      Not implemented as a regular destructor, because
      of the SymbTabEntry::AddProdTabEntry()-construction.

      In this methode, an array of ProdTabEntries is copied
10    to another array, before erasing the old one. The
      related derivations, however, shall not be killed,
      and have to be killed manually by using this method,
      if wished.
      -------------------------------------------------------
15  */
    void ProdTabEntry::Release()
    {
            DerivSymbol* CurrentSymbol;
            DerivSymbol* NextSymbol;
20
            if(FirstSymbol!=NULL)
            {
                    CurrentSymbol = FirstSymbol;

25                  while(CurrentSymbol!=LastSymbol)
                    {
                            NextSymbol = CurrentSymbol->NextSymbol;
                            delete CurrentSymbol;
                            CurrentSymbol = NextSymbol;
30                  }
                    delete CurrentSymbol;
            }
    }
```

In Zeile 21 wird zunächst geprüft, ob es überhaupt Ableitungssymbole gibt. Ist dies der Fall, dann wird die Lösch-Schleife beginnend in Zeile 25 so lange durchlaufen, bis das letzte Ableitungssymbol der Liste erreicht wurde. Das letzte Symbol wird durch die Anweisung in Zeile 31 gelöscht.

6.1.4 Die Suchraumberechnung

Da bei der Berechnug von Suchraumgrößen ziemlich schnell sehr große Zahlen auftreten, wurde eine eigene Klasse `SuperFloat` implementiert, welche eine rudimentäre Fließkommarithmetik für beinahe beliebig große Zahlen darstellt. Auf dieser Klasse aufbauend wurde die Funktion `ComputeArbSearchSpace(int)` in den Klassen `SymbTabEntry` und `ProdTabEntry` eingebaut. Diese Zweiteilung hat ihren Grund in der in Kapitel 4.1.1 vorgestellten Berechnungsmethode, die sich ebenfalls in die Suchraumberechnung für ein Symbol und die Berechnung für eine Ableitung gliedert. Hinzu kommt eine aufrufende Instanz der Funktion `ComputeArbSearchSpace(int)` in der Klasse `Language`. Der Parameter der Funktion `ComputeArbSearchSpace(int)` gibt an, bis zu welcher Tiefe der Suchraum berechnet werden soll.

6.1.4.1 Die Klasse `SuperFloat`

Intern zerlegt die Klasse `SuperFloat` jede Zahl in Mantisse und Exponent. Die Mantisse `Mant` ist eine "`long double`"-Variable und hält das Dezimalkomma stets nach der ersten signifikanten Stelle. Bei der Darstellung der Zahl Null sei definiert, daß diese durch den Wert null in der Mantisse und den Wert eins im Exponenten repräsentiert wird. Diese Art der Darstellung mit fixiertem Komma in der Mantisse soll in weiterer Folge "normierte Darstellung" genannt werden. Aus der Zahl 123 wird so z.B. $1,23 \times 10^2$. Der Exponent `Exp` ist eine `long int`-Variable. Der Wertebereich der Klasse `SuperFloat` ergibt sich somit aus dem Wertebereich von `long int`-Variablen im jeweiligen System. Und da die Mantisse als `long double`-Zahl gehalten wird, wird die Rechengenauigkeit der `SuperFloat`-Klasse durch die Genauigkeit von `long double`-Variablen im jeweiligen Sytem limitiert. Abbildung 22 zeigt die Funktionalität der `SuperFloat`-Klasse, die Addition, Multiplikation und Vergleichsoperatoren beinhaltet.

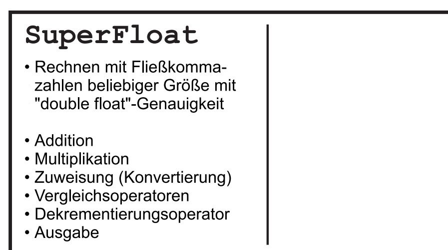


Abbildung 22: Die Klasse `SuperFloat`

- **Der Konstruktor** `SuperUnsignedInt()`

```
     /*------------------------------------------------------
       SuperFloat::SuperFloat()

       Constructor: Creating a SuperFloat-number, containing zero
5      ------------------------------------------------------
     */
     SuperFloat::SuperFloat()
     {
             Mant = 0;
10           Exp = 1;
     }
```

Der Konstruktor initialisert die neu erzeugte SuperFloat-Variable mit null.

- **Der Integer-Zuweisungsoperator** `operator =(int)`

Dieser Operator ermöglicht die Zuweisung von `int` zu `SuperFloat`.

```
     /*------------------------------------------------------
       int SuperFloat::operator =(int);

       Assignment from int to SuperFloat
 5     ------------------------------------------------------
     */
     int SuperFloat::operator =(int TheOtherNumber)
     {
             if(TheOtherNumber==0)
10           {
                     Mant = 0;
                     Exp = 1;
             }
             else
15           {
                     if(TheOtherNumber<0)
                             Exp = int(log10(-TheOtherNumber));
                     else
                             Exp = int(log10(TheOtherNumber));
20
                     Mant = (TheOtherNumber+0.0)/pow(10,Exp);
             }

             return(TRUE);
25   }
```

Ist die zuzuweisende Zahl null, so werden Exponent und Mantisse null gesetzt (Zeilen 11 und 12). Andernfalls wird in den Zeilen 16-19 der entsprechende Exponent, und in Zeile 21 die auf eine signifikante Stelle vor dem Komma normierte Mantisse errechnet. Aus 123 wird so z.B.: $1,23 \times 10^2$. Nach erfolgter Zuweisung wird `TRUE` zurückgegeben, da dies dem Verhalten der C++ - Standardzuweisungsoperatoren entspricht.

- **Der Float-Zuweisungsoperator** `operator =(float)`

Dieser Operator ermöglicht die Zuweisung von `float` zu `SuperFloat`

```
    /*-------------------------------------------------------
      int SuperFloat::operator =(float TheOtherNumber);

      Assignment from float to SuperFloat
 5    -------------------------------------------------------
    */
    int SuperFloat::operator =(float TheOtherNumber)
    {
            if(TheOtherNumber==0)
10          {
                    Mant = 0;
                    Exp = 1;
            }
            else
15          {
                    if(TheOtherNumber<0)
                            Exp = int(log10(-TheOtherNumber));
                    else
                            Exp = int(log10(TheOtherNumber));
20
                    Mant = (TheOtherNumber+0.0)/pow(10,Exp);
                    if((Mant>0&&Mant<1)||(Mant<0&&Mant>(-1)))
                    {
                            Mant = Mant*10;
25                          Exp--;
                    }
            }
            return(TRUE);
    }
```

Die Funktionsweise entspricht der des vorigen Zuweisungsoperators, lediglich die Zeilen 22-26 sind hinzugekommen. Dies hat seinen Grund darin, daß die verwendeten Formeln zur Berechnung des Exponenten und der Mantisse bei Zahlen im Bereich $0 < |x| < 1$ die Mantisse so normieren, daß die erste signifikante Stelle nach dem Dezimalkomma zu liegen kommt. Dies wird durch die hinzugefügten Programmzeilen ausgeglichen.

- **Der Double-Zuweisungsoperator** `operator =(double)`

  Dieser Operator ermöglicht die Zuweisung von `double` zu `SuperFloat`

```
    /*-------------------------------------------------------
      int SuperFloat::operator =(double TheOtherNumber);

      Assignment from double to SuperFloat
 5    -------------------------------------------------------
    */
    int SuperFloat::operator =(double TheOtherNumber)
    {
            if(TheOtherNumber==0)
10          {
                    Mant = 0;
                    Exp = 1;
            }
            else
15          {
                    if(TheOtherNumber<0)
                            Exp = int(log10(-TheOtherNumber));
                    else
                            Exp = int(log10(TheOtherNumber));
20
```

```
                        Mant = (TheOtherNumber+0.0)/pow(10,Exp);
                        if((Mant>0&&Mant<1)||(Mant<0&&Mant>(-1)))
                        {
                                Mant = Mant*10;
25                              Exp--;
                        }
                }

                return(TRUE);
30      }
```

Dieser Zuweisungsoperator unterscheidet sich in seiner Funktionsweise nicht vom vorherigen.

- **Der Vergleichsoperator auf Ungleichheit** `operator !=(int)`

  Dieser Operator ermöglicht einen Vergleich auf Ungleichheit zwischen einer SuperFloat- auf der linken, und eine Integer-Zahl auf der rechten Seite.

```
        /*--------------------------------------------------------
          int SuperFloat::operator !=(int TheOtherNumber)

          Overloading the != operator
 5        --------------------------------------------------------
        */
        int SuperFloat::operator !=(int TheOtherNumber)
        {
                SuperFloat Temp;
10
                Temp = TheOtherNumber;

                if(Exp!=Temp.Exp)
                        return(TRUE);
15
                if(Mant!=Temp.Mant)
                        return(TRUE);

                return(FALSE);
20      }
```

In Zeile 11 wird die zu vergleichende Integer-Zahl mit Hilfe des zuvor erläuterten Zuweisungsoperators in eine SuperFloat-Zahl umkopiert. In Zeile 13 wird danach auf ungleiche Exponenten, in Zeile 16 auf ungleiche Mantisse geprüft. Nur wenn beide gleich sind, wird `FALSE` zurückgegeben.

- **Der Vergleichsoperator** `operator >(SuperFloat)`

  Dieser Operator ermöglicht einen "größer-als"-Vergleich zwischen zwei SuperFloat-Zahlen.

```
        /*--------------------------------------------------------
          int SuperFloat::operator >(SuperFloat TheOtherNumber)

          Overloading the > operator
 5        --------------------------------------------------------
        */
        int SuperFloat::operator >(SuperFloat TheOtherNumber)
        {
```

```
            if(Mant>=0 && TheOtherNumber.Mant<0)
10                  return(TRUE);

            if(Mant<0 && TheOtherNumber.Mant>=0)
                    return(FALSE);

15          if(Mant==0)
            {
                    if(TheOtherNumber.Mant<0)
                            return(TRUE);
                    else
20                          return(FALSE);
            }

            if(TheOtherNumber.Mant==0)
            {
25                  if(Mant>0)
                            return(TRUE);
                    else
                            return(FALSE);
            }
30
            if(Mant>0)
            {
                    if(Exp>TheOtherNumber.Exp)
                            return(TRUE);
35
                    if(Exp<TheOtherNumber.Exp)
                            return(FALSE);

                    if(Mant>TheOtherNumber.Mant)
40                          return(TRUE);

                    return(FALSE);
            }
            else
45          {
                    if(Exp<TheOtherNumber.Exp)
                            return(TRUE);

                    if(Exp>TheOtherNumber.Exp)
50                          return(FALSE);

                    if(Mant>TheOtherNumber.Mant)
                            return(TRUE);

55                  return(FALSE);
            }
    }
```

In Zeile 9 wird geprüft, ob die linke Zahl größer oder gleich null, und die rechte Zahl kleiner null ist. In diesem Fall ist die erste Zahl immer größer als die zweite (z.B.: $2 > -2$), es wird `TRUE` zurückgegeben (Zeile 10). In Zeile 12 wird umgekehrt geprüft, ob vielleicht die linke Zahl kleiner null ist, und die rechte größer oder gleich null. In diesem Fall ist die erste Zahl immer kleiner als die zweite (z.B.: $-2 < 2$), und es wird `FALSE` zurückgegeben. Ab Zeile 14 kann also von Zahlen mit gleichem Vorzeichen ausgegangen werden, allerdings kann eine von beiden noch immer gleich null sein.

In Zeile 15 wird geprüft, ob die linke Zahl gleich null ist. Ist dann die rechte negativ, ist sie immer kleiner als die linke (z.B.: $0 > -3$), und es wird TRUE

zurückgegeben, andernfalls `FALSE`. In Zeile 23 wird geprüft, ob vielleicht die rechte Zahl gleich null ist. Ist in diesem Fall die linke Zahl größer null, dann ist die linke Zahl immer größer als die rechte (z.B.: $3 > 0$), und es wird `TRUE` zurückgegeben, andernfalls `FALSE`. Ab Zeile 39 kann also von vorzeichengleichen Zahlen ungleich null ausgegeangen werden.

In Zeile 31 wird geprüft, ob die linke Zahl (somit auch die rechte) positiv ist. Ist dies der Fall, dann zeigt, da beide Zahlen normiert sind, der größere Exponent auch die größere Zahl an (z.B.: $3 \times 10^4 > 4 \times 10^2$). In den Zeilen 33-37 wird dies abgeprüft. Sind jedoch beide Exponenten gleich, entscheidet bei zwei positiven Zahlen die größere Mantisse über die größere Zahl (z.B.: $3,3 \times 10^3 > 3,2 \times 10^3$). Dies wird in den Zeilen 39-42 getestet.

Für die Zeilen 45-55 bleiben noch zwei negative Zahlen zum Vergleich. In diesem Fall entscheidet, da beide Zahlen normiert sind, zunächst der kleinere Exponent über die größere Zahl (z.B.: $-4 \times 10^2 > -4 \times 10^5$). In den Zeilen 46-50 wird dies getestet. Sind beide Exponenten gleich, so entscheidet wieder die größere Mantisse (z.B.: $-2 \times 10^3 > -4 \times 10^3$, Zeilen 52 und 53).

- **Der Vergleichsoperator** `operator >(int)`

  Dieser Operator ermöglicht einen "größer-als"-Vergleich zwischen einer SuperFloat-Zahl auf der linken, und einer Integer-Zahl auf der rechten Seite.

```
    /*-------------------------------------------------------
      int SuperFloat::operator >(int TheOtherNumber)

      Overloading the > operator
 5    -------------------------------------------------------
    */
    int SuperFloat::operator >(int TheOtherNumber)
    {
            SuperFloat Temp;
10          int Result;

            Temp = TheOtherNumber;
            Result = (*this>Temp);
            return(Result);
15  }
```

  In Zeile 12 wird die rechte, zu vergleichende Zahl mit Hilfe des bereits erläuterten Zuweisungsoperators in eine SuperFloat-Zahl umgewandelt. Somit liegen zwei SuperFloat-Zahlen vor, die mit Hilfe des zuletzt erläuterten Vergleichsoperators verglichen werden können (Zeile 13).

- **Der Dekrementierungsoperator** `operator --()`

  Dieser Operator dekrementiert eine SuperFloat-Zahl um eins und gibt den Wert der Zahl vor der Dekrementierung zurück.

```
    /*-------------------------------------------------------
      SuperFloat SuperFloat::operator--()

      Overloading the --() operator
 5    -------------------------------------------------------
    */
    SuperFloat SuperFloat::operator--()
    {
            SuperFloat Temp;
10          SuperFloat MinusOne;

            Temp = *this;

            MinusOne = -1;
15          *this = *this + MinusOne;

            return(Temp);
    }
```

In Zeile 12 wird zunächst der momentane Wert dieser Zahl gesichert. Danach wird mit dem Integer-Zuweisungsoperator eine SuperFloat-Zahl des Wertes minus eins erzeugt (Zeile 14). Dadurch kann eine Dekrementierung mit Hilfe des Additionsoperators erreicht werden (Zeile 15), ohne daß eigens ein Subtraktionsoperator ausprogrammiert werden müßte.

- **Der Additionsoperator** `operator +(SuperFloat)`

  Mit Hilfe dieses Operators ist es möglich, zwei SuperFloat-Zahlen zu addieren.

```
    /*-------------------------------------------------------
      SuperFloat SuperFloat::operator +(SuperFloat TheOtherNumber);

      Overloading the + operator
 5    -------------------------------------------------------
    */
    SuperFloat SuperFloat::operator +(SuperFloat TheOtherNumber)
    {
            SuperFloat ResultNumber;
10          SuperFloat* TheLargerOne;
            SuperFloat* TheSmallerOne;

            if(Exp-TheOtherNumber.Exp > ACC)
                    return(*this);
15
            if(Exp-TheOtherNumber.Exp < -ACC)
                    return(TheOtherNumber);

            if(Exp>TheOtherNumber.Exp)
20          {
                    TheLargerOne = this;
                    TheSmallerOne = &TheOtherNumber;
            }
            else
25          {
                    TheLargerOne = &TheOtherNumber;
                    TheSmallerOne = this;
            }

30          ResultNumber.Mant =
            TheLargerOne->Mant +
            TheSmallerOne->Mant/pow(10,TheLargerOne->Exp-TheSmallerOne->Exp);
```

```
            ResultNumber.Exp = TheLargerOne->Exp;
35
            if(ResultNumber.Mant>=10)
            {
                  ResultNumber.Mant = ResultNumber.Mant / 10;
                  ResultNumber.Exp++;
40          }

            if(ResultNumber.Mant==0)
                  ResultNumber.Exp = 1;

45          return(ResultNumber);
      }
```

In Zeile 13 wird zunächst geprüft, ob der Exponent der linken Zahl den der rechten um den in der Präprozessorvariablen `ACC` festgelegten Wert übersteigt. Ist dies der Fall, wird angenommen, daß eine Addition aufgrund der beschränkten Rechengenauigkeit genau die linke Zahl als Ergebnis liefern würde (Zeile 14). Genau umgekehrt ist es, wenn der Exponent der rechten Zahl den der linken um den Wert von `ACC` übersteigt (Zeilen 16 und 17). Sind die Exponenten weniger als `ACC` auseinander, so wird ab Zeile 18 weitergerechnet. Der Wert der Präprozesssorvariablen `ACC` ist der Rechengenauigkeit von `double float`-Zahlen des jeweiligen Systems anzupassen. In den Zeilen 19-28 wird festgestellt, welche der beiden Zahlen den größeren Exponenten und welche den kleineren Exponenten aufweist. Der Exponent der größeren Zahl wird als Wert des Exponenten der Lösung angenommen (Zeile 34), und die Mantissen der beiden Zahlen werden, entsprechend umgerechnet, addiert (Zeilen 30-32). In Zeile 36 wird anschließend getestet, ob durch die Addition die Mantisse der Lösungszahl größer oder gleich zehn geworden ist. In diesem Fall ist der per Definition bestimmte zulässige Wertebereich der Mantisse (eine signifikante Stelle vor dem Komma) überschritten, Mantisse und Exponent werden entsprechend normiert (Zeilen 38 und 39). In Zeile 45 wird die Lösung der Addition zurückgegeben.

- **Der Multiplikationsoperator `operator *(SuperFloat)`**

  Mit Hilfe dieses Operators ist es möglich, zwei SuperFloat-Zahlen zu multiplizieren.

```
      /*-------------------------------------------------------
        SuperFloat SuperFloat::operator *(SuperFloat TheOtherNumber);

        Overloading the * operator
 5      -------------------------------------------------------
      */
      SuperFloat SuperFloat::operator *(SuperFloat TheOtherNumber)
      {
            SuperFloat ResultNumber;
10
            ResultNumber.Mant = Mant * TheOtherNumber.Mant;

            if(ResultNumber.Mant==0)
                  ResultNumber.Exp = 1;
15          else
                  ResultNumber.Exp = Exp + TheOtherNumber.Exp;
```

```
            if(ResultNumber.Mant>=10||ResultNumber.Mant<=-10)
            {
20                  ResultNumber.Mant = ResultNumber.Mant / 10;
                    ResultNumber.Exp++;
            }
            else
            {
25                  if((ResultNumber.Mant>0&&ResultNumber.Mant<1)||
                       (ResultNumber.Mant<0&&ResultNumber.Mant>-1))
                    {
                            ResultNumber.Mant = ResultNumber.Mant*10;
                            ResultNumber.Exp--;
30                  }
            }

            return(ResultNumber);
    }
```

Die Multiplikation wird durchgeführt, indem die Mantissen der beiden Zahlen multipliziert (Zeile 11) und die Exponenten addiert werden (Zeile 16). Ausnahmsweise, wenn die errechnete Mantisse genau null ergibt, so wird der Exponent definitionsgemäß auf eins gesetzt (Zeilen 13 und 14). In Zeile 19 wird geprüft, ob die Mantisse der Lösung größer oder gleich zehn ist, und nun nicht mehr mit einer Stelle vor dem Komma normiert ist. In diesem Fall wird die Mantisse auf eine Stelle vor dem Komma normiert und der Exponent entsprechend korrigiert (Zeilen 20 und 21). Andernfalls wird getestet, ob der Wert der Mantisse durch die Multiplikation unter eins gefallen ist, und somit ebenfalls den definitorischen Wertebereich (Normierung auf eine signifikante Stelle vor dem Komma) verlassen hat. Auch in diesem Fall werden Mantisse und Exponent richtiggestellt (Zeilen 27 und 28). In Zeile 33 wird die Lösung der Multiplikation zurückgegeben.

- **Der Ausgabeoperator** `operator <<(ostream&, SuperFloat&)`

  Mit Hilfe dieses Operators ist es möglich, SuperFloat-Zahlen als `ostream` auszugeben.

```
    /*---------------------------------------------------------
      ostream& operator <<(ostream& s, SuperFloat& TheNumber)

      Friend of SuperFloat.
 5
      Returning the SuperFloat-number TheNumber as ostream
      -------------------------------------------------------
    */
    ostream& operator <<(ostream& s, SuperFloat& TheNumber)
10  {
            s << TheNumber.Mant << "e" ;

            if(TheNumber.Exp>=0)
                    s << "+";
15
            s << TheNumber.Exp;
            return(s);
    }
```

In Zeile 11 wird zunächst die Mantisse gefolgt von einem “e“ ausgegeben. Danach folgt das Vorzeichen des Exponenten, welches für den Fall, daß dieser negativ ist,

automatisch zusammen mit dem Exponenten ausgegeben wird (Zeile 16). Falls der Exponent positiv (oder null) ist, wird ein Pluszeichen durch die Zeilen 13 und 14 erzeugt.

6.1.4.2 Suchraumberechnung in SymbTabEntry

Diese Funktion repräsentiert die Suchraumberechnung eines Symbols, wie in Kapitel 4.1.1 beschrieben. Im zu Abbildung 7 gehörenden Beispiel wird auf genau die gleiche Weise der Suchraum des Nichtterminalsymbols $< \gamma >$ berechnet, wie dies auch die folgende Funktion macht.

```
    /*--------------------------------------------------------
      SuperFloat* SymbTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)

      Computing the search space for MaxDerivDepth derivations
 5    with limited (long double float) accuracy.
      --------------------------------------------------------
    */
    SuperFloat* SymbTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)
    {
10  // The following 2 lines may be included if necessary or for
    // experimental use to calculate search space sizes including
    // eight byte floating point numbers.
    //
    //      SuperFloat AchtBit;
15  //      AchtBit = 256;
            int i;
            int j;
            SuperFloat* NewCard;
            SuperFloat* TempCard;
20
            if(MaxDerivDepth<=ArbCardSize)
                    return(ArbCard);

            NewCard = new SuperFloat[MaxDerivDepth+1];
25          for(i=0;i<=MaxDerivDepth;i++)
                    NewCard[i] = 0;

            if(!NoOfDerivs)
            {
30
    // The following lines may be included if necessary or for
    // experimental use to calculate search space sizes including
    // eight byte floating point numbers.
    //
35  // In the BNF-file the eight byte floating point number
    // must be represented as terminal symbol
    // "FLOATING_POINT_NUMBER".

    /*
40                  if(strcmp(Symbol,"FLOATING_POINT_NUMBER")==EQUALSTRINGS)
                    {
                            NewCard[0] = AchtBit;
                            NewCard[0] = NewCard[0]*AchtBit;
                            NewCard[0] = NewCard[0]*AchtBit;
45                          NewCard[0] = NewCard[0]*AchtBit;
                            NewCard[0] = NewCard[0]*AchtBit;
                            NewCard[0] = NewCard[0]*AchtBit;
                            NewCard[0] = NewCard[0]*AchtBit;
```

```
                                NewCard[0] = NewCard[0]*AchtBit;
50                      }
                        else
    */

    // End optional lines of code.
55
                                NewCard[0] = 1;
                }
                else
                {
60                      if(MaxDerivDepth)
                        {
                                for(i=0;i<NoOfDerivs;i++)
                                {
                                        TempCard =
65                                      DerivList[i].ComputeArbSearchSpace(MaxDerivDepth-1);

                                        for(j=1;j<=MaxDerivDepth;j++)
                                                NewCard[j] =
                                                NewCard[j] + TempCard[j-1];
70                              }
                        }
                }

                delete [] ArbCard;
75              ArbCard = NewCard;
                ArbCardSize = MaxDerivDepth;

                return(ArbCard);
    }
```

Wie zu erkennen ist, sind im Listing mehrere Zeilen auskommentiert. Damit hat es folgende Bewandtnis: Ein Nichtterminalsymbol kann für gewöhnlich in genau null Ableitungsschritten genau einen vollständigen Ableitungsbaum erzeugen (man könnte auch etwas verständlicher sagen: Bei einem Terminalsymbol ist kein Ableitungsschritt mehr nötig, und auch keiner mehr möglich). In [Geyer-Schulz, 1994, S. 368] ist jedoch ein Beispiel, welches von einem "unechten" Terminalsymbol ausgeht, das für eine acht Byte Fließkommazahl steht. Dieses "unechte" Terminalsymbol kann also in null Ableitungsschritten nicht eine, sondern 2^{64} Ableitungen erzeugen. Um dieses Beispiel (und andere in [Geyer-Schulz, 1994]) nachzurechnen, wurden die im obigen Listing auskommentierten Zeilen zeitweise dem Programm hinzugefügt. Taucht in der Sprachdefinitionsdatei dann ein Terminalsymbol mit der Bezeichnung "`FLOATING_POINT_NUMBER`" auf, so wird dieses als acht Byte Fließkommazahl angenommen. Um den Anwender in die Lage zu versetzen, solche Rechnungen nachzuvollziehen, wurden die Zeilen in auskommentierter Form im Listing belassen.

Die Funktionsweise der Funktion ist mit dem Hintergrundwissen aus Kapitel 4.1.1 schnell erklärt: In Zeile 21 wird zunächst geprüft, ob für dieses Symbol bereits eine Suchraumberechnung mit mindestens der angeforderten Tiefe `ArbCardSize` durchgeführt wurde. Die Größe der größten bisher berechneten und damit abrufbereiten Suchraumtabelle ist in der Variablen `ArbCardSize` vermerkt. Die gespeicherte Suchraumtabelle selbst wird durch den Pointer `ArbCard` referenziert. Ist also keine Neuberechnung erforderlich, dann wird einfach der Pointer auf die beste-

hende Tabelle zurückgegeben (Zeile 22). Ansonsten wird ein neues `SuperFloat`-Array der entsprechenden Größe erzeugt und initialisiert (Zeilen 24-26). In der Zeile 28 wird getestet, ob dieses `SymbTabEntry`-Objekt ein Terminalsymbol darstellt. Dies ist daran zu erkennen, daß es keine möglichen Ableitungen gibt. Ist dies so, dann ist die Suchraumtabelle sehr einfach zu erstellen: In null Schritten kann eine Baum erzeugt werden (Zeile 56). In den auskommentierten Zeilen befindet sich, wie gesagt, der abweichende Code für eine acht Byte Fließkommazahl. Handelt es sich jedoch um kein Terminalsymbol, so wird in Zeile 60 zunächst geprüft, ob die geforderte Berechnungstiefe größer null ist. Sollte dies nicht der Fall sein, dann gibt es nichts zu berechnen - die Suchraumtabelle des Nichtterminalsymbols hat eine Zeile ($i = 0$), und die erhält einen Nulleintrag. Ansonsten jedoch werden in der in Zeile 62 beginnenden Schleife alle möglichen Ableitungen des Symbols, repräsentiert durch die entsprechenden Objekte der Klasse `SymbTabEntry` (siehe Abbildung 18), nach ihren Suchraumtabellen bis zur Tiefe `MaxDerivDepth-1` gefragt. Diese Tabellen werden, wie in Kapitel 4.1.1 gezeigt, in die Ergebnistabelle für dieses Symbol hineinaddiert. In Zeile 74 wird schließlich die alte gespeicherte Suchraumtabelle gelöscht, und die neue zur nunmehrig aktuellen erklärt (Zeilen 75 und 76). In Zeile 77 wird ein Pointer auf die errechnete Tabelle als Ergebnis zurückgegeben.

6.1.4.3 Suchraumberechnung in `ProdTabEntry`

Diese Funktion repräsentiert die Suchraumberechnung einer Ableitung, wie in Kapitel 4.1.1 beschrieben. Im zu Abbildung 8 gehörenden Beispiel wird auf genau die gleiche Weise der Suchraum der dargestellten Ableitung $< \alpha >< \beta >< \gamma >$ berechnet, wie dies auch die folgende Funktion macht.

```
    /*-------------------------------------------------------
      SuperFloat* ProdTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)

      Computing the search space for MaxDerivDepth derivations
 5    with limitied (long double float) accuracy.
      -------------------------------------------------------
    */
    SuperFloat* ProdTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)
    {
10          int i;
            int j;
            int k;
            SuperFloat* NewCard;
            SuperFloat* TempCard1;
15          SuperFloat* TempCard2;
            SuperFloat* XCard;
            DerivSymbol* CurrentDerivSymbol;

            if(MaxDerivDepth<=ArbCardSize)
20                  return(ArbCard);

            NewCard = new SuperFloat[MaxDerivDepth+1];
            TempCard2 = new SuperFloat[MaxDerivDepth+1];

25          NewCard[0] = 1;

            for(i=1;i<=MaxDerivDepth;i++)
```

```
                NewCard[i] = 0;

30          CurrentDerivSymbol = FirstSymbol;
            for(i=0;i<NoOfDerivSymbols;i++)
            {
                TempCard1 =
                CurrentDerivSymbol->Symbol->
35              ComputeArbSearchSpace(MaxDerivDepth);

                for(j=0;j<=MaxDerivDepth;j++)
                    TempCard2[j] = 0;

40              for(j=0;j<=MaxDerivDepth;j++)
                {
                    if(NewCard[j]!=0)
                    {
                        for(k=0;k<=(MaxDerivDepth-j);k++)
45                      {
                            TempCard2[j+k] =
                            TempCard2[j+k] +
                            NewCard[j] * TempCard1[k];
                        }
50                  }
                }

                XCard = NewCard;
                NewCard = TempCard2;
55              TempCard2 = XCard;

                CurrentDerivSymbol =
                CurrentDerivSymbol->NextSymbol;
            }
60
            delete [] TempCard2;
            delete [] ArbCard;
            ArbCard = NewCard;
            ArbCardSize = MaxDerivDepth;
65
            return(ArbCard);
        }
```

In Zeile 19 wird zunächst wieder geprüft, ob für diese - durch dieses Objekt der Klasse `ProdTabEntry` repräsentierte - Ableitung schon eine Suchraumtabelle mit ausreichender Tiefe berechnet wurde. Ist dies der Fall, so wird diese gepufferte Tabelle als Ergebnis zurückgegeben (Zeile 20). Anderenfalls werden zwei Arrays erzeugt. Das eine soll nach der Berechnung die Ergebnistabelle beinhalten, das andere soll Zwischenergebnisse der Art aufnehmen, wie z.B. die Suchraumtabelle für die Ableitung $< \alpha >< \beta >$ im Beispiel auf Seite 24. In den Zeilen 25-28 wird das neue Array `NewCard` wie bei einem Terminalsymbol initialisiert. Der Grund dafür liegt darin, daß die Berechnungsmethode stets die bisher berechnete Suchraumtabelle um das nächste Symbol der Ableitung erweitert (siehe Abbildung 8). Zu Beginn muß es daher eine Suchraumtabelle eines virtuellen ersten Symbols geben. Es kann sich hierbei nur um die Tabelle eines - an den Anfang der Ableitung gedachten - Terminalsymboles handeln, da nur ein Terminalsymbol die Suchraumtabelle nicht verändert. In der Schleife ab Zeile 31 wird dann die verkettete Liste der Ableitungssymbole (siehe Abbildung 18) durchgegangen. Zu Beginn der Suchraumberechnung eines jeden Ableitungssymboles wird die Suchraumtabelle diese Symboles errechnet, und die Ergebnistabelle mit `TempCard1` referenziert (Zeilen

33-35). Das Array `TempCard2` wird gelöscht. Es soll in weiterer Folge das Zwischenergebnis aufnehmen. Das bisherige Zwischenergebnis liegt an dieser Stelle (wie man noch sehen wird) in `NewCard` vor. Die in Zeile 40 beginnende Schleife geht alle gefragten Suchraumtiefen der zu berechnenden Tabelle durch. Die Anweisung in Zeile 42 hilft, sinnlose Multiplikationen mit null zu vermeiden. Und schließlich beginnt in Zeile 44 jene Schleife, die, wie im Beispiel auf Seite 24 gezeigt, alle möglichen Kombinationen bis `MaxDerivDepth` errechnet und die dazugehörigen Suchraumgrößen dem Zwischenergebnis an der jeweils entsprechenden Stelle additiv hinzufügt. In den Zeilen 53-55 werden die Pointer `NewCard` und `TempCard` getauscht. Damit ist sichergestellt, daß bei jedem weiteren Schleifendurchlauf der Inhalt von `TempCard2` wieder gelöscht werden kann, und in `NewCard`, wie erwartet, das Zwischenergebnis zu finden ist. Nach dem letzten Schleifendurchlauf referenziert `NewCard` das letzte Zwischenergebnis, und das ist gleichzeitig das Endergebnis. Nach dem Löschen der Zwischenergebnistabelle und der alten Suchraumtabelle für diese Ableitung in den Zeilen 61 und 62 wird das neu errechnete Ergebnis zurückgegeben.

6.1.4.4 `Language::ComputeArbSearchSpace(int)`

Durch Aufruf dieser Funktion werden alle eventuell vorhandenen gepufferten Suchraumtabellen der `SymbTabEntry` und `ProdTabEntry`-Objekte gelöscht, und anschließend die Suchraumberechnung, beginnend vom Startsymbol der Grammatik, gänzlich neu gestartetet. Damit ist sichergestellt, daß auch bei eventuellen Grammatikerweiterungen zur Laufzeit ein neuer Aufruf dieser Funktion eine korrekte Suchraumtabelle liefert.

```
     /*--------------------------------------------------------
       void Language::ComputeArbSearchSpace(int Depth);

       Computing the search space Size up to Depth
 5     derivations and storing the result in ArbCard.

       First deleting cached old values in the symbol-
       and production table.
       --------------------------------------------------------
10   */
     int Language::ComputeArbSearchSpace(int Depth)
     {
             int i;
             SymbTabEntry* CurrentSymbol;
15
             if(StartSymbol==NULL || Depth<0)
                     return(FALSE);

             CurrentSymbol = FirstSymbol;
20           while(TRUE)
             {
                     if(CurrentSymbol->ArbCardSize>=0)
                     {
                             CurrentSymbol->ArbCardSize = -1;
25                           delete [] CurrentSymbol->ArbCard;
                             CurrentSymbol->ArbCard = NULL;
                     }

                     for(i=0;i<CurrentSymbol->NoOfDerivs;i++)
30                   {
```

```
                    if(CurrentSymbol->DerivList[i].ArbCardSize>=0)
                    {
                            CurrentSymbol->DerivList[i].ArbCardSize = -1;
                            delete [] CurrentSymbol->DerivList[i].ArbCard;
35                          CurrentSymbol->DerivList[i].ArbCard = NULL;
                    }
            }

            if(CurrentSymbol==LastSymbol)
40                  break;

            CurrentSymbol = CurrentSymbol->NextEntry;
        }

45      ArbCard = StartSymbol->ComputeArbSearchSpace(Depth);
        ArbCardSize = Depth;
        return(TRUE);
    }
```

In Zeile 16 wird zunächst geprüft, ob fehlerhafterweise kein Startsymbol definiert ist, oder ein negativer Parameter übergeben wurde. In diesem Fall wird die Funktion unter Rückgabe von `FALSE` abgebrochen (Zeile 17). In der in Zeile 20 beginnende Schleife werden solange alle Symbole der Grammatik (dargestellt durch die Objekte der Klasse `SymbTabEntry`) durchlaufen, bis in Zeile 39 das Ende der verketteten Liste gefunden wurde. In Zeile 22 wird für das momentane Symbol geprüft, ob eine gepufferte Suchraumtabelle im Speicher liegt. In diesem Fall wird in den Zeilen 24-26 diese Tabelle gelöscht. In der in Zeile 29 beginnenden Schleife werden alle Ableitungen (`ProdTabEntry`-Objekte) des aktuellen Symbols durchlaufen, und auch bei diesen, soferne nötig, die Suchraumtabellen gelöscht. Anschließend wird in Zeile 45 die Suchraumtabelle bis zur angeforderten Tiefe `Depth` gänzlich neu berechnet und `TRUE` zurückgegeben.

6.1.4.5 `Language::PrintArbSearchSpace()`

Diese Funktion gibt die Suchraumtabelle des Startsymbols, die zuvor mit `ComputeArbSearchSpace(int)` berechnet worden sein muß, auf dem Bildschirm aus.

```
    /*-------------------------------------------------------
      void Language::PrintArbSearchSpace()

      Printing out the card-table of the search space
 5    depths calculateted by

      Language::ComputeExactSearchSpace(int)
      -------------------------------------------------------
    */
10  void Language::PrintArbSearchSpace()
    {
        int i;

        cout << "Table of search space sizes in n derivations," << endl;
15      cout << " calculated with long double - accuracy." << endl << endl;

        for(i=0;i<=ArbCardSize;i++)
        {
```

```
                    if(ArbCard[i]!=0)
20                          cout << i << ": " << ArbCard[i] << endl;
            }
    }
```

Die in Zeile 17 beginnende Schleife wird so oft durchlaufen, als die gespeicherte Suchraumtabelle tief ist. Wenn es sich nicht um einen Nulleintrag handelt (Zeile 19), dann wird die entsprechende Zeile der Suchraumtabelle ausgegeben (Zeile 20).

6.1.5 Suchraumberechnung mit Genauigkeit bis zur letzten Stelle

Zu experimentellen Zwecken wurde neben der Suchraumberechnung, basierend auf der `SuperFloat`-Arithmetik, auch eine Suchraumberechnung implementiert, die auf der Klasse `SuperUnsignedInt` beruht. `SuperUnsignedInt` ist der Prototyp einer rudimentären Arithmetik zum Rechnen mit positiven, ganzen Zahlen beliebiger Länge bis zur letzten Stelle. Als "Spiegelbild" der `ComputeArbSearchSpace()`-Funktionen wurden die Funktionen

- `Language::ComputeExactSearchSpace(int)`
- `SymbTabEntry::ComputeExactSearchSpace(int)`
- `ProdTabEntry::ComputeExactSearchSpace(int)`

sowie

- `Language::PrintExactSearchSpace()`

implementiert. Abgesehen von einigen kleinen Unterschieden, die auf eine unterschiedliche Handhabung der `SuperUnsignedInt`-Arithmetik zurückgehen, sind sie exakt gleich aufgebaut, wie die zuvor erläuterten `ComputeArbSearchSpace(int)`-Funktionen. Die Anzahl der bei einem Printout maximal auszugebenden Stellen wird durch die Präprozessorvariable `MAXDIGITS` festgelegt, welche im vorliegenden Programm auf 40 eingestellt ist. Da im vorliegenden Programm zur Inititialisierung der Population die `ComputeArbSearchSpace(int)`-Funktionen verwendet werden und die prinzipielle Suchraumberechnung völlig gleich funktioniert, wird auf eine nähere Dokumentation der `ComputeExactSearchSpace(int)`-Funktionen an dieser Stelle verzichtet. Die Implementierung kann jedoch dem Gesamtlisting im Anhangkapitel A.1.4 entnommen werden.

6.2 Ableitungsbäume

Wie bereits im Kapitel 5.3 kurz erwähnt, werden alle Genotypen im vorliegenden Programm als vollständige Ableitungsbäume mit Knoten-Objekten der Basisklasse `TreeNode` dargestellt. Davon abgeleitet ist die Klasse `StartNode`. Jeder Knoten, der ein Startsymbol repräsentiert, ist von dieser Klasse. Abgeleitet von der Klasse `StartNode` ist die Klasse `RootNode`. Der Wurzelknoten eines Ableitungsbaumes (der natürlich auch immer zugleich ein Startsymbol darstellt) ist immer ein Objekt der Klasse `RootNode`.

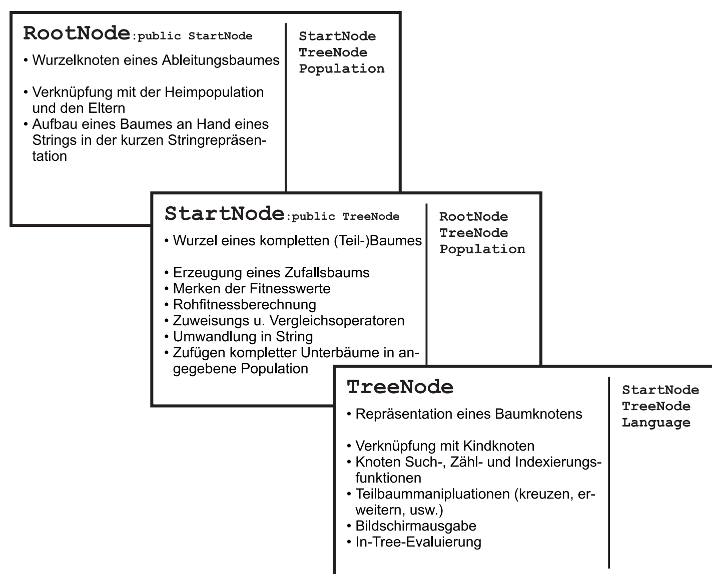


Abbildung 23: Die Klassen `TreeNode`, `StartNode` und `RootNode`

Die Variablen der Datenstruktur der Klasse `TreeNode`, die allen Objekten dieser Klassenhierarchie zu eigen sind, haben folgende Funktion:

- `SymbTabEntry* Symbol`: Beinhaltet das "Token" des durch diesen Knoten dargestellten Symbols (i.e.: ein Pointer auf das enstprechende `SymbTabEntry`-Objekt).
- `Language* TreeLanguage`: Zeigt auf die diesem Ableitungsbaum zugehörige Grammatik, dargestellt durch ein Objekt der Klasse `Language`
- `TreeNode* PredecessorNode`: Zeigt auf den Vorgängerknoten dieses Knotens. Bei einem Objekt der Klasse `RootNode` ist dieser Eintrag `NULL`
- `RootNode* Root`: Zeigt auf den Wurzelknoten `RootNode` dieses Baumes.
- `TreeNode** Derivs`: Zeigt auf ein Array von `TreeNode`-Pointern. Die in diesem Array gehaltenen Pointer zeigen auf die Kinder-Knoten dieses Knotens. Dieses Array referenziert also die konkrete Ableitung des durch diesen Knoten repräsentierten Nichtterminalsymbols. Im Falle eines Terminalsymboles ist `Derivs` gleich NULL.

- `int NoOfDerivSymbols`: Gibt an, wieviele Kinder-Knoten dieser Knoten hat (man könnte auch sagen: wieviele Symbole die Ableitung dieses Knotens hat). Bei einem Terminalsymbol ist der Inhalt dieser Variablen null.

- `int DerivationNo`: Gibt an, mit der wievielten der für dieses Symbol möglichen Ableitungen dieser Knoten erweitert wurde. Die Nummerierung bezieht sich auf die Nummerierung der Ableitungen in der durch `Treelanguage` repräsentierten Grammatik.

- `unsigned int DerivationDepth`: Enthält die Ableitungstiefe des an dieser Stelle beginnenden Unterbaumes. Die Ableitungstiefe ist gleich der Anzahl der in diesem Unterbaum enthaltenen Nichtterminalsymbole (einschließlich dieses Knotens).

- `unsigned int TempIndex`: Variable zur Aufnahme eines temporären Index zur schnelleren Durchsuchung des Baumens nach Knoten, die ein bestimmtes Symbol repräsentieren.

In der von `TreeNode` abgeleiteten Klasse `StartNode` kommen zusätzlich folgende Variablen hinzu:

- `float RawFitness`: Enthält die Rohfitness des Individuums, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

- `float StandardizedFitness`: Enthält die standardisierte Fitness des Individuums, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

- `float AdjustedFitness`: Enthält die adjustierte Fitness des Individuums, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

- `float NormalizedFitness`: Enthält die normalisierte Fitness des Individuums, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

- `float TargSamplRate`: Nimmt die Auswahlwahrscheinlichkeit (tsr) des Individuums, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird, auf.

- `char* PhenoString`: Kann auf einen String zeigen, der den Phänotypen des Individuums enthält, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

- `char* GenoString`: Kann auf einen String zeigen, der den Genonotypen (in seiner kurzen oder langen Darstellung) des Individuums enthält, das durch den an dieser Stelle beginnenden Ableitungsbaum repräsentiert wird.

In der von `StartNode` abgeleiteten Klasse `RootNode` schließlich, kommen noch die folgenden Variablen hinzu:

- `Population* Home`: Pointer auf die "Heim"-Population des an dieser Stelle beginnenden Ableitungsbaumes.
- `StartNode* Parent1`: Pointer auf den ersten Elternteil dieses Ableitungsbaumes. Gibt es keine Eltern (z.B. unmittelbar nach der Initialisierung), so zeigt dieser Pointer auf `NULL`.
- `StartNode* Parent2`: Pointer auf den zweiten Elternteil dieses Ableitungsbaumes. Gibt es keinen zweiten Elternteil (z.B. wenn keine Kreuzung durchgeführt wurde), so zeigt dieser Pointer auf `NULL`.
- `int Mutated`: Enthält `TRUE`, wenn der Ableitungsbaum durch eine Mutation verändert wurde.

6.2.1 Die Funktionen der Klasse `RootNode`

Die Klasse `RootNode` ist abgeleitet von der Klasse `StartNode`, welche ihrerseits von `TreeNode` abgeleitet ist. Ein Objekt der Klasse `RootNode` repräsentiert also einen Baumknoten (`TreeNode`-Eigenschaft), der ein Startsymbol beinhaltet (`StartNode`-Eigenschaft), und selbst keine Elternknoten mehr besitzt, also Wurzelknoten ist (`RootNode`-Eigenschaft). Neben der Funktionalität der Klassen `TreeNode` und `StartNode` beinhaltet diese Klasse zusätzlich Pointer, die eine Verknüpfung mit den Eltern des an dieser Stelle beginnenden Individuums herstellen können, sowie eine Funktion zur Erweiterung des an dieser Stelle beginnenden Ableitungsbaumes anhand eines Vorlage in kurzer Stringrepräsentation (siehe Kapitel 5.3.5).

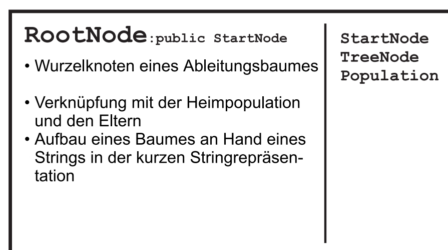


Abbildung 24: Die Klasse `RootNode`

6.2.1.1 `RootNode()`

Der Leerkonstruktor definiert, daß der neu erzeugte Wurzelknoten keiner Population zugeordnet ist (Zeile 13), keine Eltern hat (Zeilen 10 und 11), und nicht weiter erweitert wird. Ein `RootNode`-Pointer des neu erzeugten `RootNode`-Objekts wird auf sich selber gesetzt (Zeile 14). Vorgängerknoten gibt es naturgemäß keinen (Zeile 15).

```
    /*------------------------------------------------------
       RootNode::RootNode()

       Constructor: Creating an empty tree with no parents and
 5     no home-population
      ------------------------------------------------------
    */
    RootNode::RootNode()
    {
10          Parent1 = NULL;
            Parent2 = NULL;
            Mutated = NO;
            Home = NULL;
            Root = this;
15          PredecessorNode = NULL;
    }
```

6.2.1.2 RootNode(Language*, int, Population*)

Dieser Konstruktor erzeugt (ohne Berücksichtigung der Gleichverteilung) ausgehend von diesem Wurzelknoten einen zufälligen Baum, der der im ersten Parameter referenzierten Grammatik und im letzten Parameter referenzierten Population zugeordnet ist. Der zweite Parameter gibt die maximal zulässige Ableitungstiefe (definiert als die Anzahl der Nichtterminalsymbole im Baum) des Baumes an.

```
    /*------------------------------------------------------
      RootNode::RootNode(Language* L,
                         int MaxDerivDepth,
                         Population* HomePopulation)
 5
      Constructor: Creating a random-tree of the given
      language L with the given maxium derivation depth,
      part of the population, referred by the HomePopulation-
      pointer.
10    -----------------------------------------------------
    */
    RootNode::RootNode(Language* L, int MaxDerivDepth, Population* HomePopulation)
    {
            int i;
15          unsigned int NoOfNonTerminalNodes;
            Root = this;
            Parent1 = NULL;
            Parent2 = NULL;
            Mutated = NO;
20          Home = HomePopulation;
            PredecessorNode = NULL;
            DerivationDepth = 1;
            TempIndex = 1;
            TreeLanguage = L;
25          RawFitness = 0;
            StandardizedFitness = 0;
            AdjustedFitness = 0;
            NormalizedFitness = 0;
            TargSamplRate = 0;
30          PhenoString = NULL;
            GenoString = NULL;
            Symbol = L->StartSymbol;
            Derivs = 0;
            NoOfDerivSymbols = 0;
35          DerivationNo = 0;
```

```
        while(TRUE)
        {
40              NoOfNonTerminalNodes = 1;
                if(RandomEnhance(MaxDerivDepth, NoOfNonTerminalNodes))
                        break;
                else
                {
45                      for(i=0;i<NoOfDerivSymbols;i++)
                                delete Derivs[i];

                        delete [] Derivs;
                }
50      }
    }
```

In den Zeilen 16-36 wird zunächst die Datenstruktur initialisiert:

- Das neu erzeugte `RootNode`-Objekt "ist seine eigene Wurzel" (Zeile 16).
- Es gibt keine Eltern und zuletzt gab es keine Mutation (Zeilen 17-19).
- Die "Heimpopulation" wird vermerkt (Zeile 20).
- Das neu erzeugte Objekt der Klasse `RootNode` ist ein Wurzelknoten und hat keinen Vorgängerknoten (Zeile 21).
- Die Ableitungstiefe beträgt vorerst eins, da es im Moment nur diesen einen Knoten gibt (Zeile 22).
- Die diesem Baum zugeordnete Grammatik wird vermerkt (Zeile 24).
- Die Fitnesswerte und die tsr werden mit null initialisiert (Zeilen 25-29).
- Es gibt bislang noch keine Stringrepräsentation des Ableitungsbaumes (Zeilen 30 und 31).
- Das neu erzeugte Objekt ist ein Wurzelknoten, und daher dem Startsymbol der Grammatik zugeordnet (Zeile 32).
- Vorerst gibt es keine Ableitungen (Kinderknoten) (Zeilen 33 und 35).

Die in Zeile 38 beginnende Schleife wird schließlich so lange durchlaufen, bis der Knoten erfolgreich zu einem Zufallsbaum erweitert werden konnte (dazu sind unter Umständen wegen der festgelegten maximal zulässigen Ableitungstiefe mehrere Versuche notwendig). Die Erweiterung erfolgt mit Hilfe der Funktion `TreeNode::RandomEnhance (unsigned int,unsigned int)` in Zeile 41. Diese Funktion erwartet als ersten Parameter die maximal zulässige Ableitungstiefe und als zweiten Parameter eine zur Verfügung gestellte globale Variable, die die bislang erzeugten Nichtterminalknoten (und somit die Ableitungstiefe) zählt. Diese Variable wird in Zeile 40 mit eins initialisiert. Schlägt der Erweiterungsversuch fehl, so werden die erzeugten Unterbäume in den Zeilen 45 und 46 vernichtet.

6.2.1.3 operator =(StartNode&)

Dieser Operator ermöglicht die Zuweisung eines vollständigen Ableitungsbaumes, beginnend mit einem `StartNode` zu diesem Ableitungsbaum, der an dieser Stelle seinen Wurzelknoten `RootNode` hat. Alle eventuell bestehenden Kinder-Knoten (Ableitungen) werden, samt ihren kompletten Unterbäumen, zuvor gelöscht. Alle Pointer dieses `RootNode` werden entsprechend eingestellt, die Fitnesswerte werden mitübernommen. Der rechts vom Gleichheitszeichen stehende Baum wird als erster Elternteil vermerkt. Sollte die Zuweisung von einem temporären Baum erfolgen, so muß die Variable `Parent1` nach der Zuweisung entsprechend verändert werden.

```
    /*-------------------------------------------------------
      int RootNode::operator =(StartNode& TheOtherTree)

      New assignment operator =
 5
      Deleting, if necessary, the complete subtree,
      starting from this RootNode, and building up a copy of
      the right tree.
      This is done by using the methode
10    void TreeNode::EnhanceWith(TreeNode*)

      TheOtherTree is stored as parent. Must be altered after
      assignment if not OK. Home population is home population
      of TheOtherTree. Fitness values and tsr are copied also.
15    -------------------------------------------------------
    */
    int RootNode::operator =(StartNode& TheOtherTree)
    {
            int i;
20
            if(NoOfDerivSymbols>0)
            {
                    for(i=0;i<NoOfDerivSymbols;i++)
                            delete Derivs[i];
25
                    delete [] Derivs ;
            }

            Parent1 = &TheOtherTree;
30          Parent2 = NULL;
            Mutated = NO;
            Home = TheOtherTree.Root->Home;
            Root = this;
            DerivationDepth = TheOtherTree.DerivationDepth;
35          TempIndex = TheOtherTree.TempIndex;
            PredecessorNode = NULL;
            RawFitness = TheOtherTree.RawFitness;
            StandardizedFitness = TheOtherTree.StandardizedFitness;
            AdjustedFitness = TheOtherTree.AdjustedFitness;
40          NormalizedFitness = TheOtherTree.NormalizedFitness;
            TargSamplRate = TheOtherTree.TargSamplRate;
            TreeLanguage = TheOtherTree.TreeLanguage;
            Symbol = TheOtherTree.Symbol;

45          EnhanceWith(&TheOtherTree);

            return(TRUE);
    }
```

In den Zeilen 21-27 werden zunächst eventuell vorhandene Kinderknoten samt ihren Unterbäumen gelöscht. In Zeile 29 wird der andere Ableitungsbaum als erster Elternteil vermerkt. Die "Heimpopulation" des zuzuweisenden Baumes sei auch die "Heimpopulation" dieses Baumes (Zeile 32), ebenso die dem anderen Baum zugeordnete Grammatik (Zeile 42) und das dem anderen Baum zugeordnete Symbol (Zeile 43). Ebenfalls übernommen werden die Ableitungstiefe und der temporäre Index des anderen Baumes (Zeilen 34-35), sowie alle Fitnesswerte und die Auswahlwahrscheinlichkeit (*target sampling rate*) (Zeilen 37-41). Anschließend wird dieser Startknoten mit der Vorlage des zu kopierenden Baumes erweitert. Dies geschieht unter Verwendung der Funktion `TreeNode::EnhanceWith(TreeNode*)`.

6.2.1.4 Set(char*)

Die Coaching-Funktion ermöglicht es, diesen Wurzelknoten gemäß dem durch den Parameter referenzierten String zu erweitern. Der String muß den gewünschten Baum in der kurzen Stringrepräsentation enthalten (siehe Kapitel 5.3.5). Sollte die Erweiterung aufgrund eines syntaktischen Fehlers im String fehlschlagen, bleibt der von diesem Wurzelknoten beginnende Ableitungsbaum unberührt und es wird `FALSE` zurückgegeben. Anderenfalls wird er durch den neu erzeugten Ableitungsbaum ersetzt und der Rückgabewert dieser Funktion ist `TRUE`.

```
     /*-------------------------------------------------------
       int RootNode::Set(char* TheString)

       Assignment from genotype-string TheString
 5     (StartNode::GenoTypeString(SHORT)-representation)
       to this tree. Returning FALSE in case of an invalid
       string.
       -------------------------------------------------------
     */
10   int RootNode::Set(char* TheString)
     {
             int EnhancementOK;
             int Position;
             RootNode NewTree;
15
             if(TheString==NULL)
                     return(FALSE);

             NewTree.TreeLanguage = TreeLanguage;
20           NewTree.Home = Home;
             NewTree.Symbol = TreeLanguage->StartSymbol;

             Position = 0;
             EnhancementOK = NewTree.EnhanceWithString(TheString,Position);
25
             if(EnhancementOK)
             {
                     *this = NewTree;
                     Parent1 = NULL;
30           }
             else
                     cerr << "Genotype-string error at Position " << Position+1 << endl;

             return(EnhancementOK);
35   }
```

In Zeile 16 wird zunächst geprüft, ob überhaupt ein String übergeben wurde. Sollte dies nicht der Fall sein, so wird unter Rückgabe von `FALSE` abgebrochen. Andernfalls werden in den Zeilen 19-21 dem temporären Baum `NewTree` die entsprechende Grammatik, die richtige "HeimPopulation" und das korrekte Startsymbol zugewiesen. In Zeile 24 wird dann, unter Verwendung der Funktion `TreeNode::EnhanceWithString(char*,int&)`, der temporäre Baum unter Vorlage des Strings `TheString` erweitert. Der zweite Parameter ist eine globale Variable, welche die momentane Position im String zählt. Startposition ist null (Zeile 23). Hat es bei der Erweiterung keine Probleme gegeben, so wird in den Zeilen 28 und 29 der temporäre Baum mit Hilfe des Zuweisungsoperators auf den an dieser Stelle beginnenden Baum umkopiert, und festgelegt, daß keine Eltern verfügbar sind. Andernfalls wird eine Fehlermeldung ausgegeben (Zeile 32). In Zeile 34 wird schließlich im Rückgabewert der aufrufenden Instanz mitgeteilt, ob die Erweiterung funktioniert hat.

6.2.1.5 `CrossOver(CrossOverParameters)`

Diese Funktion löscht zunächst alle eventuell vorhandenen Kinder-Knoten dieses Wurzelknotens, samt den dazugehörigen Unterbäumen. Danach wird dieser Wurzelknoten zu einem neuen Baum erweitert, der sich aus den zwei im Parameter `Selected` übergebenen Teilbäumen, beginnend bei `TreeNode* Selected.FirstTree` und `TreeNode* Selected.SecondSubTree` wie folgt zusammensetzt: Begonnen wird die Erweiterung vom Baum `Selected.FirstTree`. Die Erweiterung wird solange durchgeführt, bis das Programm auf den in `Selected` referenzierten Kreuzungsknoten `Selected.CrossOverPoint` trifft. Der mit diesem Knoten beginnende Unterbaum wird unter Vorlage des durch `Selected.SecondSubTree` referenzierten Teilbaumes erweitert. Die durch `Selected.CrossOverPoint` und `Selected.SecondSubTree` referenzierten Knoten müssen dasselbe Symbol der Grammatik beinhalten.

```
    /*-------------------------------------------------------
      int RootNode::CrossOver(CrossOverParameters Selected)

      Deleting, if necessary, the complete existing tree,
 5    and building up a copy of Selected.FirstTree, until
      Selected.CrossOverPoint occurs. The copying from
      this point on is done from Selected.SecondTree.

      This is done by use of the methode
10    void TreeNode::XEnhanceWith(TreeNode*, TreeNode*, TreeNode*)

      Assignement of parents and home-population included.
      -------------------------------------------------------
    */
15  int RootNode::CrossOver(CrossOverParameters Selected)
    {
            int i;

            if(Selected.CrossOverPoint==NULL||Selected.SecondSubTree==NULL)
20          {
                    cerr << "ERROR while Tree::Crossover. No treepointer (NULL)!" << endl;
                    return(FALSE);
            }
            else
```

```
25          {
                    if(Selected.CrossOverPoint->Symbol!=Selected.SecondSubTree->Symbol)
                    {
                            cerr << "ERROR while StartNode::Crossover. ";
                            cerr << "Subtrees start with different symbols!" << endl;
30                          return(FALSE);
                    }
                    else
                    {
                            if(NoOfDerivSymbols>0)
35                          {
                                    for(i=0;i<NoOfDerivSymbols;i++)
                                            delete Derivs[i];

                                    delete [] Derivs;
40                          }

                            Parent1 = Selected.FirstTree;
                            Parent2 = Selected.SecondSubTree->Root;
                            Mutated = NO;
45                          Root = this;
                            PredecessorNode = NULL;
                            Home =  Selected.SecondSubTree->Root->Home;
                            RawFitness = 0;
                            StandardizedFitness = 0;
50                          AdjustedFitness = 0;
                            NormalizedFitness = 0;
                            TargSamplRate = 0;
                            TreeLanguage = Selected.FirstTree->TreeLanguage;
                            Symbol = TreeLanguage->StartSymbol;
55                          DerivationDepth = Selected.FirstTree->DerivationDepth;
                            TempIndex = 0;

                            XEnhanceWith(Selected.FirstTree,
                                         Selected.CrossOverPoint,
60                                       Selected.SecondSubTree)->DerivationDepthUpdate();
                    }
            }
            return(TRUE);
      }
```

In Zeile 19 wird zunächst geprüft, ob nicht `Selected.CrossOverPoint` oder `Selected.SecondSubTree` gleich NULL sind. Ist dies so, dann wird unter Ausgabe einer entsprechenden Fehlermeldung abgebrochen (Zeilen 21 und 22).

Andernfalls wird in Zeile 26 geprüft, ob die durch `Selected.CrossOverPoint` und `Selected.SecondSubTree` referenzierten Knoten das gleiche Symbol der Grammatik beinhalten. Sollte dies nicht der Fall sein, so wird in Zeilen 28 und 29 eine entsprechende Fehlermeldung ausgegeben und die Routine abgebrochen (Zeile 30).

In der in Zeile 34 beginnenden Schleife werden sodann alle Kinderknoten dieses Wurzelknotens, inklusive aller dazugehörigen Teilbäume, vernichtet.

In den Zeilen 41 und 42 werden die in `Selected` referenzierten Individuen als Eltern des neu zu erzeugenden Individuums eingetragen. Als "Heimpopulation" wird die Population des durch `Selected.SecondSubTree` referenzierten Teilbaumes eingetragen. Sollte dies nicht erwünscht sein, so ist der `Home`-Pointer nach der Kreuzung entsprechend richtig zu stellen. Da sich dem neu erzeugten Baum a priori keine Fitnesswerte

und keine tsr zuordnen lassen, werden diese Werte auf null gestellt (Zeilen 48-52).

In den Zeilen 53 und 54 wird dann als Grammatik dieses Baumes die Grammatik des durch `Selected.FirstTree` referenzierten Baumes eingestellt, und das Startsymbol zugewiesen.

Die eigentliche Erweiterung erfolgt in den Zeilen 58-60 unter Verwendung der Funktion `TreeNode::XenhanceWith(...)`. Diese Funktion liefert einen Pointer auf den neu erzeugten Knoten, bei dem die Kreuzung durchgeführt wurde, zurück.

Da die Funktion `XEnhanceWith(...)` die Werte von `TreeNode::DerivationDepth` mitkopiert, haben alle Knoten, die auf einem tieferen Niveau liegen als dieser Knoten, korrekte Ableitungstiefen in der Variablen `DerivationDepth` vermerkt. Alle Knoten im Baum, die über diesem Knoten liegen, haben aber nun falsche Ableitungstiefen gespeichert, da sich ja der an diesem Punkt beginnende Teilbaum - im Vergleich zum durch `Selected.CrossOverPoint` referenzierten Teilbaum - geändert hat.

Dieses Problem wurde durch den Aufruf der Funktion `->DerivationDepthUpdate()` gelöst (Zeile 60). Diese Funktion durchwandert den Baum, unter Verwendung der `PredecessorNode`-Pointer, rekursiv von dem Punkt an, von dem sie aufgerufen wurde, bis zum Wurzelknoten, und aktualisert dabei den Inhalt aller `DerivationDepth`-Variablen. Nach erfolgreicher Kreuzung wird in Zeile 63 die Funktion unter Rückgabe von `TRUE` beendet.

6.2.2 Die Funktionen der Klasse `StartNode`

Die Klasse `StartNode` ist von der Klasse `TreeNode` abgeleitet. Ein Objekt der Klasse `StartNode` repräsentiert also einen Baumknoten (`TreeNode`-Eigenschaft), der ein Startsymbol beinhaltet (`StartNode`-Eigenschaft), und somit die Wurzel eines vollständigen (Teil-)Baumes darstellt. Neben der Funktionalität der Klasse `TreeNode` beinhaltet diese Klasse zusätzlich Variablen zur Aufnahme von Fitnesswerten (siehe Kapitel 5.3.1), eine rekursive Funktion zum Aufbau eines Zufallsbaumes, Zuweisungs- und Vergleichsoperatoren, Funktionen zur Umwandlung des an dieser Stelle beginnenden Baumes in einen String, sowie die vom Anwender zu ergänzende Funktion `ObjFuncVal()` zur Rohfitnessberechnung. Hinzu kommt die Funktion `AddCompleteSubTreesTo(...)`, auf welche das Klassenobjekt `Population` zurückgreift. Diese Funktion fügt alle kompletten Unterbäume, die die angegebene Größe nicht überschreiten, der angegebenen Population hinzu. Die Einbettung der Klasse `StartNode` in die Umgebung der anderen Klassen ist in Abbildung 13 auf Seite 47 dargestellt.

6.2.2.1 `StartNode()`

Der Leerkonstruktor initialisiert in einem neu erzeugten Objekt der Klasse `StartNode` die Fitnesswerte und die Auswahlwahrscheinlichkeit (*target sampling rate* - tsr) mit null (Zeilen 9-13) und die Pointer, die die Stringrepräsentation des Individuums referenzieren, mit NULL (Zeilen 14 und 15).

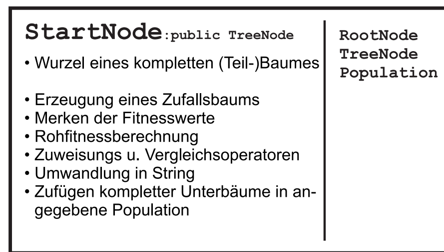


Abbildung 25: Die Klasse `StartNode`

```
    /*-------------------------------------------------------
       StartNode::StartNode()

       Constructor: Creating an empty StartNode
 5    -------------------------------------------------------
    */
    StartNode::StartNode()
    {
            RawFitness = 0;
10          StandardizedFitness = 0;
            AdjustedFitness = 0;
            NormalizedFitness = 0;
            TargSamplRate = 0;
            PhenoString = NULL;
15          GenoString = NULL;
    }
```

6.2.2.2 StartNode(Language*, TreeNode*, RootNode*)

Dieser Konstruktor definiert, daß das neu erzeugte Objekt der Klasse `StartNode` der im ersten Parameter referenzierten Grammatik (`Language`), dem im zweiten Parameter referenzierten Vorgängerknoten (`TreeNode`) und dem im letzten Parameter referenzierten Wurzelknoten (`RootNode`) zugeordnet ist.

```
    /*-------------------------------------------------------
       StartNode::StartNode(Language* L, TreeNode* Predecessor, RootNode* R)

       Constructor: Creating an empty start node
 5     of language L with a link to the predecessor node
       and to the root node.
      -------------------------------------------------------
    */
    StartNode::StartNode(Language* L, TreeNode* Predecessor, RootNode* R)
10  {
            PredecessorNode = Predecessor;
            DerivationDepth = 1;
            TempIndex = 0;
            Root = R;
15          RawFitness = 0;
            StandardizedFitness = 0;
            AdjustedFitness = 0;
            NormalizedFitness = 0;
            TargSamplRate = 0;
20          PhenoString = NULL;
            GenoString = NULL;
            Symbol = L->StartSymbol;
            Derivs = NULL;
```

```
            NoOfDerivSymbols = 0;
25          DerivationNo = 0;
            TreeLanguage = L;
    }
```

In Zeile 11 wird zunächst der Vorgängerknoten definiert. Die Ableitungstiefe in Zeile 12 wird eins gesetzt, da der Ableitungsbaum vorerst lediglich aus diesem einen Startsymbol besteht. In Zeile 14 wird der Wurzelknoten des Ableitungsbaumes vermerkt. In den Zeilen 15-21 werden die Fitnesswerte, die tsr, und die Stringpointer initialisiert. In Zeile 22 wird diesem Objekt der Klasse `StartNode` das Startsymbol der Grammatik zugeordnet. Die Zeilen 23-25 legen fest, daß dieser Startknoten vorerst keine Ableitung (keine Kinderknoten) besitzt. In Zeile 26 schließlich wird noch die zum Baum gehörige Grammatik vermerkt.

6.2.2.3 operator =(StartNode&)

Dieser Operator ermöglicht die Zuweisung eines vollständigen Ableitungsbaumes, beginnend mit einem `StartNode`, zu diesem Ableitungsbaum, der an dieser Stelle einen Startsymbolknoten `StartNode` hat. Alle eventuell bestehenden Kindernoten (Ableitungen) werden, samt ihren kompletten Unterbäumen, zuvor gelöscht. Alle Fitnesswerte werden übernommen und die vermerkten Ableitungstiefen jedes Baumknotens oberhalb dieses Knotens werden aktualisiert.

```
    /*--------------------------------------------------------
      virtual int StartNode::operator =(StartNode& TheOtherTree)

      New assignment operator =
 5
      Deleting, if necessary, the complete subtree,
      and building up a copy of the right tree. This
      is done by using the methode

10    void TreeNode::EnhanceWith(TreeNode*)
      --------------------------------------------------------
    */
    int StartNode::operator =(StartNode& TheOtherTree)
    {
15          int i;

            if(NoOfDerivSymbols>0)
            {
                    for(i=0;i<NoOfDerivSymbols;i++)
20                          delete Derivs[i];
                    delete [] Derivs ;
            }

            RawFitness = TheOtherTree.RawFitness;
25          StandardizedFitness = TheOtherTree.StandardizedFitness;
            AdjustedFitness = TheOtherTree.AdjustedFitness;
            NormalizedFitness = TheOtherTree.NormalizedFitness;
            TargSamplRate = TheOtherTree.TargSamplRate;
            TreeLanguage = TheOtherTree.TreeLanguage;
30          Symbol = TheOtherTree.Symbol;
            DerivationDepth = TheOtherTree.DerivationDepth;
            TempIndex = TheOtherTree.TempIndex;
```

```
            EnhanceWith(&TheOtherTree);
35
            if(PredecessorNode!=NULL)
                   PredecessorNode->DerivationDepthUpdate();

            return(TRUE);
40    }
```

In den Zeilen 21-22 werden zunächst eventuell vorhandene Kinderknoten samt ihren Unterbäumen gelöscht. Danach werden die Fitnesswerte und die tsr kopiert (Zeilen 24-28), die Grammatik, Startsymbol und die vermerkte Ableitungstiefe des Startknotens des zuzuweisenden Baumes übernommen (Zeilen 29-31). Anschließend wird dieser Startknoten mit der Vorlage des zu kopierenden Baumes erweitert. Dies geschieht unter Verwendung der Funktion `TreeNode::EnhanceWith(TreeNode*)`. Der an dieser Stelle beginnende Knoten ist kein Wurzelknoten, denn in diesem Fall wird der Zuweisungsoperator der Klasse `RootNode` aufgerufen. Alle Knoten des Baumes, die über diesem Knoten liegen, haben aber nun falsche Ableitungstiefen `DerivationDepth` vermerkt. Dieses Problem wurde durch den Aufruf der Funktion `DerivationDepthUpdate()` im Vorgängerknoten gelöst (Zeile 37). Diese durchwandert den Baum, unter Verwendung der `PredecessorNode`-Pointer, rekursiv von dem Punkt an, von dem sie aufgerufen wurde, bis zum Wurzelknoten, und aktualisiert dabei den Inhalt aller `DerivationDepth`-Variablen.

6.2.2.4 operator ==(StartNode&)

Hierbei handelt es sich um einen Vergleichsoperator, mit dessen Hilfe überprüft werden kann, ob der an dieser Stelle beginnende Ableitungsbaum identisch ist mit dem auf der rechten Seite des Operators stehenden Ableitungsbaum. Aber Achtung! Bei manchen Grammatiken können unterschiedliche Ableitungsbäume (Genotypen) zu gleichen Wörtern (Phänotypen) führen. Es kann daher passieren, daß dieser Operator eine Ungleichheit zweier Ableitungsbäume ergibt, die jedoch bei Anwendung der `TreeNode::Print()`-Funktion dasselbe Wort ausgeben! Will man die Phänotypen vergleichen, so kann man das mit Hilfe des Strings, der von der folgenden `PhenoTypeString()`-Funktion geliefert wird.

```
      /*---------------------------------------------------------
        int StartNode::operator ==(StartNode& TheOtherTree)

        New boolean operator ==
 5
        Comparing the complete tree, including all subtrees
        ---------------------------------------------------------
      */
      int StartNode::operator ==(StartNode& TheOtherTree)
10    {
            return(CompareWith(&TheOtherTree));
      }
```

In Zeile 11 greift der Vergleichsoperator auf die "protected"-Funktion `TreeNode::CompareWith(TreeNode*)` zurück, welche den Vergleich durchführt. Das Ergebnis wird an die aufrufende Instanz weitergereicht.

6.2.2.5 PrintParents()

Diese Funktion druckt folgende Informationen aus:

- Falls der an dieser Stelle beginnende Baum Teilbaum eines größeren Baumes ist, den Phänotypen dieses Baumes.
- Die Eltern des Baumes, von dem dieser Baum ein Teil ist, soferne verfügbar. Ist dies der Wurzelknoten, dann ist mit der Formulierung "der Baum, von dem dieser Baum ein Teil ist genau der an dieser Stelle beginnende Baum gemeint.
- Ob der Baum, von dem dieser Teilbaum ein Teil ist, einer Mutation unterzogen wurde.

Die Funktionsweise der Elternbestimmung wurde auch in Kapitel 5.3.4 bereits ausführlich behandelt.

```
    /*-------------------------------------------------------
      void StartNode::PrintParents();

      Printing out the parents of this tree and the mutation
 5    status.
      -------------------------------------------------------
    */
    void StartNode::PrintParents()
    {
10          cout << endl << endl;
            if(Root==NULL)
                    cout << "No parents available." << endl << endl;
            else
            {
15                  if(Root->Home==NULL)
                            cout << "No parents available." << endl << endl;
                    else
                    {
                            if(!Root->Home->ParentsAvailable || Root->Parent1==NULL)
20                             cout << "No parents available." << endl << endl;
                            else
                            {
                                    if(Root!=this)
                                    {
25                                          cout << "Individual is part of the ";
                                            cout << "following:";
                                            cout << endl << endl;
                                            Root->Print();
                                            cout << endl << endl;
30                                  }
                                    if(Root->Parent2==NULL)
                                    {
                                            cout << "This individual has ";
```

```
                                        cout << "been cloned without ";
35                                      cout << "any crossover from:";
                                        cout << endl << endl;
                                        Root->Parent1->Print();
                                        cout << endl << endl;
                                }
40                              else
                                {
                                        cout << "The Parents are:";
                                        cout << endl << endl;
                                        Root->Parent1->Print();
45                                      cout << endl << endl;
                                        Root->Parent2->Print();
                                        cout << endl << endl;
                                }
                        }
50              }

                if(Root->Mutated)
                {
                        cout << "There was a mutation.";
55                      cout << endl << endl;
                }
        }
}
```

In Zeile 11 wird zunächst geprüft, ob der Wurzelknoten dieses Startknotens bekannt ist. Sollte dies nicht der Fall sein (fehlerhafter Zustand), dann wird in der Zeile 12 die Meldung *"No parents available."* ausgegeben und nicht weitergemacht. Andernfalls wird in Zeile 15 getestet, ob im Wurzelknoten die "Heimpopulation" vermerkt ist. Sollte dies nicht der Fall sein (fehlerhafter Zustand), dann wird wiederum die Meldung *"No parents available.* ausgegeben, und die Elternbestimmung aufgegeben. Andernfalls wird in Zeile 19 geprüft, ob für diese Population im allgemeinen und für dieses Individuum im speziellen Eltern verfügbar sind. Sollte dies nicht der Fall sein, dann wird in Zeile 21 wiederum *"No parents available."* ausgegeben. Andernfalls wird in Zeile 23 getestet, ob dieser Teilbaum Teil eines größeren Baumes ist. Falls dies zutrifft, wird in den Zeilen 25-29 eine entsprechende Meldung ausgegeben. Danach wird in Zeile 31 geprüft, ob ein oder zwei Elternteile vorhanden sind. Ist nur ein Elternteil vorhanden, dann wird die Meldung in den Zeilen 33-38 ausgegeben, ansonsten die Meldung in den Zeilen 42-47. In Zeile 52 schließlich wird noch getestet, ob der Baum, von dem dieser Teilbaum ein Teil ist, einer Mutation unterzogen wurde. In diesem Fall wird eine entsprechende Meldung (Zeilen 54 und 55) ausgegeben.

6.2.2.6 `GenoTypeString(int)`

Diese Funktion liefert den an diesem Startknoten beginnenden Ableitungsbaum (den Genotypen) in seiner kurzen oder langen Stringrepräsentation (siehe Kapitel 5.3.5), je nachdem welcher Parameter übergeben wird (`SHORT` für die kurze, `LONG` für die lange Darstellung).

```
    /*---------------------------------------------------------
      char* StartNode::GenoTypeString(int Representation)

      Returning a char-string representation of the tree.
 5    There are two representations available:


      Long representation (human-readable)
      (representation==LONG)
10
        best explained by this example:

                           <fe>
                   "("   <f1>     <fe>    ")"
15                     "NOT"  "(" <f0> ")"
                                  "D2"

        is represented by the stream

20      4<fe>"("1<f1>"NOT"3<fe>"("1<f0>"D2"")"")"


      Short representation: (machine-readable)
      (representation==SHORT)
25
        Same as in the ostream&<< operator.
      -------------------------------------------------------
    */
    char* StartNode::GenoTypeString(int Representation)
30  {
            int Position;

            if(GenoString!=NULL)
                    delete [] GenoString;
35
            GenoString = new char[GenoTypeStringLength(Representation)+1];
            Position = 0;
            BuildUpGenoTypeStringIn(GenoString, Position, Representation);
            GenoString[Position] = '\0';
40          return(GenoString);
    }
```

In Zeile 33 wird zunächst geprüft, ob von dieser Funktion an dieser Stelle schon einmal ein String berechnet wurde. Dieser wird dann gegebenenfalls vernichtet (Zeile 34) und Platz für einen neuen String reserviert (Zeile 36). Die Länge des zu erstellenden Strings wird mit Hilfe der Funktion `TreeNode::GenoTypStringLength(int)` ermittelt. Die eigentliche Erzeugung des Strings erfolgt unter Zuhilfenahme der Funktion `TreeNode::BuildUpGenoTypeStringIn(char*, int&, int)`. Diese erhält als ersten Parameter die Adresse des Character-Arrays, in dem der String aufgebaut werden soll, als zweiten Parameter eine zur Verfügung gestellte globale Variable, die die momentane Position im String mitzählt und als dritten Parameter die Art der gewünschten Stringdarstellung (`LONG` oder `SHORT`. Der Positionszähler wird in Zeile 37 mit 0 initialisiert. In Zeile 39 wird das C++-konforme Stringende-Zeichen angehängt, und in Zeile 41 schließlich der Pointer auf den erzeugten String zurückgegeben.

6.2.2.7 PhenoTypeString()

Diese Funktion liefert den Phänotypen des an diesem Startknoten beginnenden Ableitungsbaumes, wie er auch von der Funktion `TreeNode::Print()` ausgegeben würde, als String.

```
    /*------------------------------------------------------
      char* StartNode::PhenoTypeString()

      Returning a char-string of the tree like a
 5    TreeNode::Print() - output.
      ------------------------------------------------------
    */
    char* StartNode::PhenoTypeString()
    {
10          int Position;

            if(PhenoString!=NULL)
                    delete [] PhenoString;

15          PhenoString = new char[PhenoTypeStringLength()+1];
            Position = 0;
            BuildUpPhenoTypeStringIn(PhenoString, Position);
            PhenoString[Position] = '\0';
            return(PhenoString);
20  }
```

Die Funktionsweise ist ganz analog der zuvor besprochenen Funktion: In Zeile 12 wird zunächst geprüft, ob von dieser Funktion an dieser Stelle schon einmal ein String berechnet wurde. Dieser wird dann gegebenenfalls vernichtet (Zeile 13) und Platz für einen neuen String reserviert (Zeile 15). Die Länge des zu erstellenden Strings wird mit Hilfe der Funktion `TreeNode::PhenoTypStringLength(int)` ermittelt. Die eigentliche Erzeugung des Strings erfolgt unter Zuhilfenahme der Funktion `TreeNode::BuildUpPhenoTypeStringIn(char*, int&)`. Diese erhält als ersten Parameter die Adresse des Character-Arrays, in dem der String aufgebaut werden soll und als zweiten Parameter eine zur Verfügung gestellte globale Variable, die die momentane Position im String mitzählt. Der Positionszähler wird in Zeile 16 mit 0 initialisiert. In Zeile 18 wird das C++ - konforme Stringende-Zeichen angehängt, und in Zeile 19 schließlich der Pointer auf den erzeugten String zurückgegeben.

6.2.2.8 SubTree(unsigned int)

Diese Funktion liefert einen Pointer auf den Teilbaum, der dem im Parameter angegebenen Knotenindex (Knotennummer) entspricht. Ein Teilbaum ist in diesem Zusammenhang ein mit einem Nichtterminalknoten beginnender Baum. Der Index jedes Knotens errechnet sich aus den unmittelbar zur Verfügung stehenden Ableitungstiefen. Durch diesen Trick können die gespeicherten Ableitungstiefen der Knoten zur schnellen indexierten Suche herangezogen werden. Die Nummerierung der Knoten funktioniert folgendermaßen:

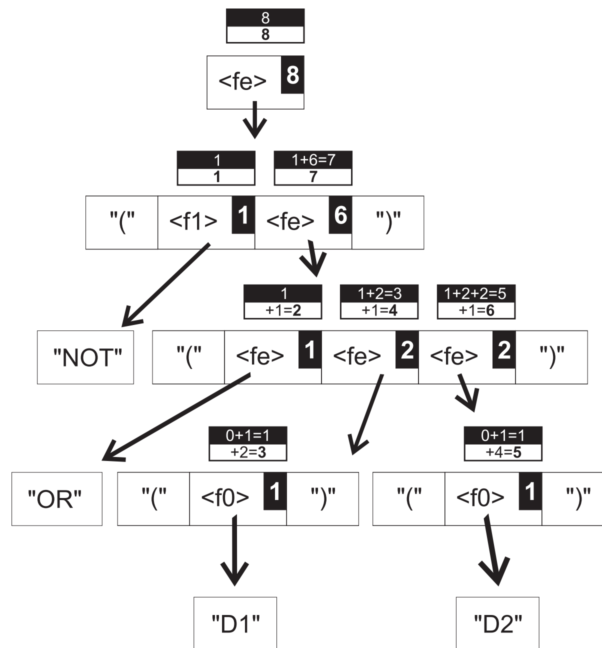


Abbildung 26: Knotenindexierung mit Hilfe der Ableitungstiefe

- Bei allen Nichtterminalknoten wird die Summe der Ableitungstiefen aller links in der gleichen Ableitung befindlichen Nichtterminalknoten zur Ableitungstiefe des betrachteten Knotens dazuaddiert und als Zwischenergebnis für diesen Knoten vorgemerkt.

- Gibt es keinen Knoten links vom betrachteten Knoten in der gleichen Ableitung, dann wird null hinzuaddiert.

- Das errechnete Ergebnis für die Wurzel des betrachteten Baumes ist also gleich seiner Ableitungstiefe und zugleich die gesuchte Knotennummer der Wurzel.

- Zu den anderen errechneten Zwischenergebnissen wird, von der zweiten Ebene des Baumes beginnend, die Nummer des links vom Mutterknoten in der gleichen Ableitung befindlichen Nichtterminalknotens addiert.

- Gibt es links vom betrachteten Mutterknoten in der gleichen Ableitung keinen Nichtterminalknoten, dann wird null hinzuaddiert.

Das Beispiel in Abbildung 26 verdeutlicht dies. In den schwarzen Kästchen in den Nichtterminalsymbolen ist die Ableitungstiefe eingetragen. Im oberen, schwarzen Teil der Rechenfelder wird das Zwischenergebnis errechnet, im weißen, unteren Teil der endgültige Index. Der im Parameter `TargetTreeNumber` übergebene Index ist gleich dem nach dem obigen Verfahren errechneten Index minus eins. Will man also z.B. den nach dem obigen Verfahren errechneten fünften Knoten finden, so muß man die folgende Funktion mit vier als Parameter aufrufen.

```
    /*-------------------------------------------------------
      TreeNode* StartNode::SubTree(unsigned int TargetTreeNumber)

      Returning a pointer to the starting TreeNode of the
 5    complete sub-tree referenced by TargetTreeNumber.

      0 <= TargetTreeNumber < NoOfSubTrees()

      In case the sub-tree numbered TargetTreeNumber
10    does not exist, this methode returns NULL.
      ------------------------------------------------------
    */
    TreeNode* StartNode::SubTree(unsigned int TargetTreeNumber)
    {
15          int i;
            unsigned int Offset;
            TreeNode* CurrentTreeNode;

            Offset=0;
20          CurrentTreeNode = this;
            TargetTreeNumber++;

            while(CurrentTreeNode->NoOfDerivSymbols>0)
            {
25                  if(Offset+CurrentTreeNode->DerivationDepth==TargetTreeNumber)
                            return(CurrentTreeNode);

                    i = 0;

30                  while(i<CurrentTreeNode->NoOfDerivSymbols)
                    {
                            if(Offset +
                            CurrentTreeNode->Derivs[i]->DerivationDepth>=
                            TargetTreeNumber)
35                                  break;

                            Offset +=
                            CurrentTreeNode->Derivs[i]->DerivationDepth;

40                          i++;
                    }

                    if(i==CurrentTreeNode->NoOfDerivSymbols)
                            return(NULL);
45                  else
                            CurrentTreeNode = CurrentTreeNode->Derivs[i];
            }
            return(NULL);
    }
```

Das obige Programm sucht nun nach folgender Strategie den angeforderten Knoten:

1. Setze `Offset` gleich 0 (Zeile 19).
2. Mache den Startknoten zum aktuellen Knoten (Zeile 20).
3. Addiere eins zur übergebenen Zielknotennummer, um den Unterschied zwischen der obigen Indexberechnungsmethode und der Parameterdefinition auszugleichen (Zeile 21).

4. Wenn der Offset plus der Ableitungstiefe des aktuellen Knotens gleich der gesuchten Zielknotennummer ist, ist der aktuelle Knoten der gesuchte Knoten. Das Programm wird abgebrochen, und der aktuelle Knoten zurückgegeben (Zeilen 25 und 26).

5. Falls dies nicht der Fall ist, gehe alle Symbole der Ableitung des aktuellen Knotens (alle Kindknoten) durch, und addiere die jeweilige Ableitungstiefe zum Offset (Zeile 37 und 38), bis Offset plus Ableitungstiefe nicht mehr kleiner sind als die Zielknotennummer (Zeilen 32-35)).

6. Mache den Ableitungsknoten, bei dem die Abbruchbedingung in Punkt 5 erreicht wurde, zum aktuellen Knoten (Zeile 46).

7. Wurde die Abbruchbedingung der Schleife von Punkt 5 auch beim letzten Symbol nicht erreicht, dann ist die Knotennummer ungültig. Brich unter Rückgabe von `NULL` ab (Zeile 44).

8. Andernfalls fahre fort mit Punkt 4.

6.2.2.9 SubTree(SymbTabEntry*, unsigned int, unsigned int)

Diese Funktion liefert einen Pointer auf den Startknoten des Unterbaumes, der das im ersten Parameter referenzierte Startsymbol beinhaltet, die im letzten Parameter angegebene Ableitungstiefe nicht überschreitet und durch die Baumnummer im zweiten Parameter referenziert wird. Die Nummerierung ist dabei nicht identisch mit der Nummerierung der Funktion `SubTree(unsigned int)`, da nur die Unterbäume, die die geforderte maximale Ableitungstiefe nicht überschreiten und deren Startknoten das gewählte Symbol beinhalten, gezählt werden. Der Indexierungsmechanismus funktioniert jedoch ganz analog, nur wird nicht auf den in `DerivationDepth` gespeicherten Wert zurückgegriffen, sondern auf `TempIndex`. Vor Aufruf dieser Funktion ist es also nötig, die Funktion `TreeNode::NoOfSubTrees(SymbTabEntry*, unsigned int)` mit den gleichen Parametern aufzurufen. Diese Funktion gibt die Anzahl der Teilbäume, deren Startknoten ein bestimtes Symbol beinhalten und die gewisse Ableitungstiefe nicht überschreiten, zurück. Dabei werden alle Baumknoten abgesucht und "nebenbei der temporäre Index `TempIndex` erstellt. So wie `DerivationDepth` für einen bestimmten Knoten angibt, wieviele Unterbäume - beginnend mit irgend einem Nichtterminalsymbol - sich unter ihm befinden, so gibt `TempIndex` an, wieviele Unterbäume - beginnend mit dem gewählten Symbol und die gewählte Ableitungstiefe nicht überschreitend - sich unterhalb dieses Knotens befinden.

```
    /*-------------------------------------------------------
      TreeNode* StartNode::SubTree(SymbTabEntry* SelectedSymbol,
                                   unsigned int TargetTreeNumber,
                                   unsigned int MaxDerivDepth)
5
      Returning a pointer to the starting TreeNode of the
      complete sub-tree referenced by SelectedSymbol,
```

```
       TargetTreeNumber and MaxDerivSymbol.

10     For example: SubTree(L.Symbol("f0),3,20) is returning
       the 3rd subtree starting with symbol "f0", counting
       only subtrees with maximum derivation depth 20.

       If the acquired subtree is non-existent, this
15     method is returning NULL.
       ------------------------------------------------------
     */
     TreeNode* StartNode::SubTree(SymbTabEntry* SelectedSymbol,
                                  unsigned int TargetTreeNumber,
20                                unsigned int MaxDerivDepth)
     {
             int i;
             unsigned int Offset;
             TreeNode* CurrentTreeNode;
25
             Offset=0;
             CurrentTreeNode = this;
             TargetTreeNumber++;

30           while(CurrentTreeNode->NoOfDerivSymbols>0)
             {
                     if(
                     Offset + CurrentTreeNode->TempIndex==TargetTreeNumber &&
                     CurrentTreeNode->Symbol==SelectedSymbol &&
35                   CurrentTreeNode->DerivationDepth<=MaxDerivDepth)
                             return(CurrentTreeNode);

                     i = 0;
                     while(i<CurrentTreeNode->NoOfDerivSymbols)
40                   {
                             if(
                             Offset+CurrentTreeNode->Derivs[i]->TempIndex>=
                             TargetTreeNumber)
                                     break;
45
                             Offset +=
                             CurrentTreeNode->Derivs[i]->TempIndex;

                             i++;
50                   }

                     if(i==CurrentTreeNode->NoOfDerivSymbols)
                             return(NULL);
                     else
55                           CurrentTreeNode = CurrentTreeNode->Derivs[i];
             }
             return(NULL);
     }
```

Die Funktion funktioniert, bis auf die Abweichung in Zeile 32-35 und die Verwendung von `TempIndex` statt `DerivationDepth`, analog zur zuletzt vorgestellten Funktion `SubTree(unsigned int)`:

1. Setze `Offset` gleich 0 (Zeile 26).
2. Mache den Startknoten zum aktuellen Knoten (Zeile 27).
3. Addiere eins zur übergebenen Zielknotennummer, um den Unterschied zwischen der obigen Indexberechnungsmethode und der Parameterdefinition auszu gleichen (Zeile 28).

4. Wenn die Summe des Offsets und des temporären Indexes gleich der gesuchten Zielknotennummer ist **und** der aktuelle Knoten das gewählte Symbol beinhaltet **und** der Baum unter dem aktuellen Knoten die gewählte maximale Ableitungstiefe nicht überschreitet, **dann** ist der aktuelle Knoten der gesuchte Knoten. Brich ab und gib den aktuellen Knoten zurück (Zeilen 30 bis 36).

5. Falls dies nicht der Fall ist, gehe alle Symbole der Ableitung des aktuellen Knotens (alle Kindknoten) durch, und addiere den jeweiligen temporären Index zum Offset hinzu (Zeile 37 und 38), bis Offset plus temporärer Index nicht mehr kleiner ist als die Zielknotennummer (Zeilen 32-35)).

6. Mache den Ableitungsknoten, bei dem die Abbruchbedingung in Punkt 5 erreicht wurde, zum aktuellen Knoten (Zeile 46).

7. Wurde die Abbruchbedingung der Schleife von Punkt 5 auch beim letzten Symbol nicht erreicht, dann ist die Knotennummer ungültig. Brich unter Rückgabe von `NULL` ab (Zeile 44 und 44).

8. Andernfalls fahre fort mit Punkt 4.

6.2.2.10 AddCompleteSubTreesTo(Population*, unsigned int)

Diese Funktion fügt diesen Knoten zum virtuellen Teil der im ersten Parameter referenzierten Population. Anschließend sucht sie rekursiv alle vollständigen Unterbäume, die die im zweiten Parameter angegebene minimale Ableitungstiefe nicht unterschreiten, und fügt diese ebenfalls zum virtuellen Teil der im ersten Parameter referenzierten Population hinzu. Vollständige Unterbäume sind in diesem Zusammenhang Teilbäume, die mit einem, ein Startsymbol beinhaltenden Knoten (also `StartNode` oder `RootNode`), beginnen. Der Knoten, in dem diese Funktion aufgerufen wurde, wird der Population in jedem Fall, ohne Prüfung der Ableitungstiefe, hinzugefügt.

```
    /*-------------------------------------------------------
      void StartNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                            unsigned int MinDepth)

 5    Adding all complete subtrees (trees starting with a
      StartNode) with a depth larger or equal MinDepth
      to the ThePopulation.
      -------------------------------------------------------
    */
10  void StartNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                          unsigned int MinDepth)
    {
            int i;

15          ThePopulation->AddTreeToVirtualPop(this);

            for(i=0;i<NoOfDerivSymbols;i++)
            {
                    if(Derivs[i]->DerivationDepth>=MinDepth)
20                          Derivs[i]->
                            AddCompleteSubTreesTo(ThePopulation,MinDepth);
            }
    }
```

In Zeile 15 wird zunächst dieser Knoten dem virtuellen Teil der angegebenen Population hinzugefügt, und zwar unter Verwendung der Funktion `Population::AddTreeToVirtualPop(StartNode)`. In der in Zeile 17 beginnenden Schleife werden alle Ableitungsknoten durchlaufen. Soferne die unter diesem Knoten liegenden Bäume die minimale Ableitungstiefe nicht unterschreiten, wird die Funktion `AddCompleteSubTreesTo(...)` auch rekursiv für diese Knoten aufgerufen (Zeilen 20 und 21). Bei Knoten der Klasse `TreeNode` handelt es sich dabei um die Funktion `TreeNode::AddCompleteSubTreesTo(...)`, welche zwar weitersucht, aber den betreffenden Knoten - da er kein Startknoten ist - der Population **nicht** hinzufügt.

6.2.2.11 EnhanceWith(StartNode*)

Diese Funktion erweitert den Startknoten `StartNode` mit der Vorlage des Baumes, der mit dem im Parameter referenzierten Startknoten beginnt. Dies geschieht unter rekursiver Verwendung der Funktionen `StartNode::EnhanceWith(StartNode*)` und `TreeNode::EnhanceWith(TreeNode*)`. Die Fitnesswerte und die Auswahlwahrscheinlichkeit (*target sampling rate*) werden mitkopiert.

```
    /*-------------------------------------------------------
      void StartNode::EnhanceWith(StartNode* TheOtherNode)

      Enhancing the StartNode with the derivation given
 5    by TheOtherNode.
      -------------------------------------------------------
    */
    void StartNode::EnhanceWith(StartNode* TheOtherNode)
    {
10
            int i;

            RawFitness = TheOtherNode->RawFitness;
            StandardizedFitness = TheOtherNode->StandardizedFitness;
15          AdjustedFitness = TheOtherNode->AdjustedFitness;
            NormalizedFitness = TheOtherNode->NormalizedFitness;
            TargSamplRate = TheOtherNode->TargSamplRate;
            NoOfDerivSymbols = TheOtherNode->NoOfDerivSymbols;
            DerivationNo = TheOtherNode->DerivationNo;
20          Derivs = new TreeNode*[NoOfDerivSymbols];

            for(i=0;i<NoOfDerivSymbols;i++)
            {
                    if(TheOtherNode->Derivs[i]->Symbol==TreeLanguage->StartSymbol)
25                          Derivs[i] =
                            new StartNode(TreeLanguage, this, Root);
                    else
                            Derivs[i] =
                            new TreeNode(TheOtherNode->Derivs[i]->Symbol,
30                                          TreeLanguage,this,Root);

                    Derivs[i]->DerivationDepth =
                    TheOtherNode->Derivs[i]->DerivationDepth;

35                  Derivs[i]->TempIndex = 0;

                    if(TheOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                            Derivs[i]->EnhanceWith(TheOtherNode->Derivs[i]);
            }
40  }
```

In den Zeilen 13-17 werden zunächst die Fitnesswerte und die Auswahlwahrscheinlichkeit (*target sampling rate* - tsr) des im Parameter referenzierten Knotens übernommen. In Zeile 18 wird die Anzahl der die Ableitungssymbole enthaltenden Kindknoten und in Zeile 19 die Nummer der Ableitung übernommen. In Zeile 20 wird ein entsprechend großes Array zur Aufnahme der Ableitungsknoten-Pointer geschaffen. In der in Zeile 22 beginnenden Schleife werden die Ableitungsknoten des zu kopierenden Knotens durchgegangen. In Zeile 24 wird geprüft, ob der momentane Ableitungsknoten ein `StartNode` ist. In diesem Fall wird der Ableitung dieses Knotens ebenfalls ein `StartNode` hinzugefügt (Zeilen 25 und 26). Andernfalls handelt es sich um einen `TreeNode` (Zeilen 28-30). In den Zeilen 32-35 werden die Ableitungstiefe und der temporäre Index übernommen. Schließlich wird in Zeile 37 geprüft, ob der soeben kopierte Ableitungsknoten selber Ableitungen besitzt. In diesem Fall wird die Funktion `EnhanceWith(...)` rekursiv aufgerufen.

6.2.2.12 `ObjFuncVal()`

Diese Funktion bildet die Schnittstelle zwischen dem genetischen Algorithmus und der Problemstellung. Sie muß vom Anwender ausprogrammiert werden, und soll den Rohfitnesswert des Individuums liefern, das dem an dieser Stelle beginnenden Ableitungsbaum entspricht. Die Art und Weise, wie dies geschehen kann, ist im Kapitel 5.2 bereits erläutert worden. In diesem Kapitel ist auch eine möglich Ausformulierung dieser Funktion für das XOR-Problem zu finden.

6.2.3 Die Funktionen der Klasse `TreeNode`

Ein Objekt der Klasse `TreeNode` repräsentiert einen Baumknoten eines Ableitungsbaumes. Es sorgt für die Verknüpfung mit den Kindknoten, bietet Such-, Zähl- und Indexierungsfunktionen, sowie Funktionen zur Teilbaummanipulation und rekursive Hilfsfunktionen für Vergleich, Erweiterung und Knotensuche, deren sich die Klasse `StartNode` bedient. Die Einbettung der Klasse `TreeNode` in die Umgebung der anderen Klassen ist in Abbildung 13 auf Seite 47 dargestellt.

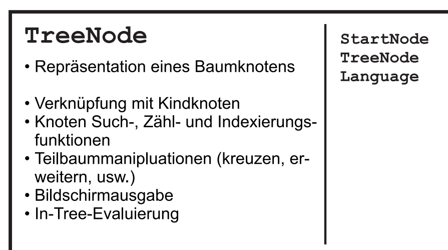


Abbildung 27: Die Klasse `TreeNode`

6.2.3.1 TreeNode()

Der Leerkonstruktor legt fest, daß der neu erzeugte Baumknoten der Klasse TreeNode vorerst keiner Grammatik, keinem Symbol und keinem Ableitungsbaum zugeordnet ist.

```
    /*------------------------------------------------------
      TreeNode::TreeNode()

      Constructor: Creating an empty TreeNode
 5    ------------------------------------------------------
    */
    TreeNode::TreeNode()
    {
            Symbol = NULL;
10          Derivs = NULL;
            NoOfDerivSymbols = 0;
            TreeLanguage = NULL;
            PredecessorNode = NULL;
            DerivationDepth = 0;
15          TempIndex = 0;
            Root = NULL;
            DerivationNo = 0;
    }
```

6.2.3.2 TreeNode(SymbTabEntry*, Language*, TreeNode*, RootNode*)

Dieser Konstruktor legt fest, daß der neu erzeugte Baumknoten der Klasse TreeNode dem im ersten Parameter referenzierten Symbol SymbTabEntry, der im zweiten Parameter referenzierten Grammatik Language, dem im dritten Parameter referenzierten Vorgängerknoten TreeNode und dem im vierten Parameter referenzierten Wurzelknoten RootNode zugeordnet ist.

```
    /*------------------------------------------------------
      TreeNode::TreeNode(SymbTabEntry* TheSymbol,
                         Language* L,
                         TreeNode* Predecessor,
 5                       RootNode* R)

      Constructor: Creating an TreeNode of Language L
      with Symbol TheSymbol and linking it to the
      predecessor and root node.
10    ------------------------------------------------------
    */
    TreeNode::TreeNode(SymbTabEntry* TheSymbol,
                       Language* L,
                       TreeNode* Predecessor,
15                     RootNode* R)
    {
            Symbol = TheSymbol;
            Derivs = NULL;
            NoOfDerivSymbols = 0;
20          TreeLanguage = L;
            PredecessorNode = Predecessor;
            DerivationDepth = 0;
            TempIndex = 0;
            Root = R;
25          DerivationNo = 0;
    }
```

In Zeile 17 wird zunächst das im ersten Parameter referenzierte Symbol zugeordnet. Danach wird in den Zeilen 18,19 und in Zeile 25 festgelegt, daß dieser Knoten vorerst keine Ableitung (keine Kinderknoten) besitzt. Anschließend wird in Zeile 20 die Grammatik und in Zeile 21 der Vogängerknoten zugewiesen. Und da dieser Knoten ja keine Ableitung hat, werden Ableitungstiefe und temporärer Index vorerst null gesetzt (Zeilen 22 und 23).

6.2.3.3 Der Destruktor von TreeNode()

Der Destruktor stellt (durch rekursiven Aufruf) sicher, daß alle im Baum unterhalb dieses Knotens befindlichen Knoten vernichtet werden.

```
   /*------------------------------------------------------
     TreeNode::~TreeNode()

     Destructor: Deleting the complete subtree.
 5   ------------------------------------------------------
   */
   TreeNode::~TreeNode()
   {
           int i;
10
           if(NoOfDerivSymbols>0)
           {
                   for(i=0;i<NoOfDerivSymbols;i++)
                           delete Derivs[i];
15
                   delete [] Derivs;
           }
   }
```

In Zeile 11 wird zunächst geprüft, ob es unterhalb dieses Knotens noch weitere Knoten im Ableitungsbaum gibt. Ist dies der Fall, so werden diese Ableitungsknoten in der in Zeile 13 beginnenden Schleife durchgegangen, und (samt ihrer eventuell vorhandenen Unterknoten) vernichtet (Zeile 14). Danach wird das Array `Derivs`, welches die Pointer auf die Kinderknoten beinhaltet hat, ebenfalls vernichtet.

6.2.3.4 Print()

Diese Funktion druckt, wenn dieser Knoten ein Terminalsymbol beinhaltet (i.e. es gibt keine Ableitungsknoten mehr), dieses Symbol auf den Bildschirm, ansonsten werden rekursiv die `Print()`-Funktionen der Ableitungsknoten aufgerufen. Dies führt zum Ausdruck des Phänotypen des durch diesen (Teil-)Baum repräsentierten Individuums.

```
   /*------------------------------------------------------
     void TreeNode::Print()

     Printing, if no further derivations available, the
 5   terminal symbol, otherwise the Print()-methods of
     the derivation-nodes are called.
```

```
      This leads to a printout of the phenotype-string.

10    cout << PhenoTypeString(); would have the same effect,
      but had the disadvantage of needing more CPU-time
      and memory.
      -------------------------------------------------------
    */
15  void TreeNode::Print()
    {
            int i;
            if(NoOfDerivSymbols==0)
                    cout << Symbol->Symbol;
20          else
            {
                    for(i=0;i<NoOfDerivSymbols;i++)
                            Derivs[i]->Print();
            }
25  }
```

In Zeile 18 wird zunächst geprüft, ob es sich um einen ein Terminalsymbol beinhaltenden Knoten handelt. Ist dies der Fall, wird die (im `SymbTabEntry` des durch den Pointer `Symbol` referenzierten Symbols gespeicherte) Textrepräsentation des Symbols ausgegeben. Andernfalls werden in der in Zeile 22 beginnenden Schleife die Ableitungsknoten durchgegangen, und rekursiv die `Print()`-Funktionen der Ableitungsknoten aufgerufen (Zeile 23).

6.2.3.5 operator <<(ostream&, TreeNode&)

Dieser Operator gibt, wenn dieser Knoten ein Nichtterminalsymbol beinhaltet (i.e.: es exisistieren Ableitungsknoten), die Nummer der Ableitung gefolgt von einem Punkt als `ostream` aus, und ruft sodann rekursiv die `<<`-Operatoren der Ableitungsknoten auf. Dies führt dazu, daß der an dieser Stelle beginnende (Teil-)Baum in seiner kurzen Stringrepräsentation ausgegeben wird (siehe Kapitel 5.3.5).

```
    /*-------------------------------------------------------
      ostream& operator <<(ostream& s, TreeNode& TheTreeNode)

      Friend of TreeNode.
 5    Returning the complete TreeNode, including the
      derivation, as ostream.

      This example explains the syntax. The Tree

10                             <fe>
                   "("    <f1>      <fe>     ")"
                       "NOT"   "(" <f0> ")"
                                   "D2"

15    is within the grammar

      S := <fe>;
      <fe> := "(" <f0> ")" |
              "(" <f1> <fe> ")" |
20            "(" <f2> <fe> <fe> ")" ;
      <f0> := "D1" | "D2" ;
      <f1> := "NOT" ;
```

```
     <f2> := "OR" | "AND" ;

25   represented by the stream

     2.1.1.2.

     because
30
     "("<f1><fe>")" is the 2nd derivation of <fe>,
     "NOT" is the 1st (and incidentally the only) derivation of <f1>,
     "("<f0>")" is the 1st derivation of <fe> in the second row,
     and "D2" ia the 2nd derivation of <f0>.
35   -------------------------------------------------
    */
    ostream& operator <<(ostream& s, TreeNode& TheTreeNode)
    {
            int i;
40
            if((&TheTreeNode)!=NULL && TheTreeNode.Symbol!=NULL)
            {
                    if(TheTreeNode.NoOfDerivSymbols>0)
                            s << TheTreeNode.DerivationNo+1 << '.' ;
45
                    for(i=0;i<TheTreeNode.NoOfDerivSymbols;i++)
                            s << *(TheTreeNode.Derivs[i]);
            }
            return(s);
50  }
```

In Zeile 41 wird geprüft, ob auch tatsächlich ein auszugebender `TreeNode` übergeben wurde, und ob diesem ein Symbol zugeordnet ist (Regelfall). Sodann wird in Zeile 43 getestet, ob es sich um einen ein Nichtterminalsymbol beinhaltenden Knoten handelt. In diesem Fall wird in Zeile 44 die in `DerivationNo` gespeicherte Ableitungsnummer plus eins (da die erste Ableitung in `DerivationNo` den Wert null erhält) ausgegeben. Danach werden in der in Zeile 46 beginnenden Schleife, so vorhanden, alle Ableitungsknoten durchwandert, und die `<<`-Operatoren dieser Knoten rekursiv aufgerufen.

6.2.3.6 `operator =(TreeNode&)`

Dieser Operator ermöglicht die Zuweisung eines Teilableitungsbaumes, beginnend mit einem `TreeNode`, zu dem an dieser Stelle beginnenden Teilableitungsbaum. Alle eventuell bestehenden Kinder-Knoten (Ableitungen) werden, samt ihren kompletten Unterbäumen, zuvor gelöscht. Alle vermerkten Ableitungstiefen werden mitübernommen und die vermerkten Ableitungstiefen jedes Baumknotens oberhalb dieses Knotens werden aktualisiert.

```
    /*-------------------------------------------------------
     int TreeNode::operator =(TreeNode& TheOtherTreeNode)

     New assignment operator =
 5
     Deleting, if necessary, the complete tree,
     and building up a copy of the right tree. This
     is done by using the method
```

```
10      void TreeNode::EnhanceWith(TreeNode*)
        ------------------------------------------------------
      */
      int TreeNode::operator =(TreeNode& TheOtherTreeNode)
      {
15            int i;

              if(NoOfDerivSymbols>0)
              {
                      for(i=0;i<NoOfDerivSymbols;i++)
20                            delete Derivs[i];
                      delete [] Derivs;
              }

              TreeLanguage = TheOtherTreeNode.TreeLanguage;
25            DerivationDepth = TheOtherTreeNode.DerivationDepth;
              TempIndex = TheOtherTreeNode.TempIndex;
              Symbol = TheOtherTreeNode.Symbol;
              EnhanceWith(&TheOtherTreeNode);

30            if(PredecessorNode!=NULL)
                      PredecessorNode->DerivationDepthUpdate();

              return(TRUE);
      }
```

In den Zeilen 17-22 werden zunächst eventuell vorhandene Kinderknoten samt ihren Unterbäumen gelöscht. Danach wird die Grammatik, das Symbol, der temporäre Index und die vermerkte Ableitungstiefe des zuzuweisenden Knotens übernommen (Zeilen 24-27). Anschließend wird dieser Knoten mit der Vorlage des zu kopierenden Baumes erweitert. Dies geschieht unter Verwendung der Funktion `EnhanceWith(TreeNode*)` (Zeile 28). Der an dieser Stelle beginnende Knoten ist kein Wurzelknoten, denn in diesem Fall wird der Zuweisungsoperator der Klasse `RootNode` aufgerufen. Alle Knoten des Baumes, die über diesem Knoten liegen, haben aber nun falsche Ableitungstiefen `DerivationDepth` vermerkt. Dieses Problem wird durch den Aufruf der Funktion `DerivationDepthUpdate()` im Vorgängerknoten gelöst (Zeile 31). Diese durchwandert den Baum, unter Verwendung der `PredecessorNode`- Pointer, rekursiv von dem Punkt an, von dem sie aufgerufen wurde, bis zum Wurzelknoten, und aktualisiert dabei den Inhalt aller `DerivationDepth`-Variablen. In Zeile 33 schließlich wird der Zuweisungsoperator, mit der C++ - konformen Rückgabe von `TRUE`, beendet.

6.2.3.7 `DerivationDepthUpdate()`

Diese Funktion durchwandert, wie zuvor gerade erwähnt, den Baum, unter Verwendung der `PredecessorNode`-Pointer, rekursiv von dem Punkt an, von dem sie aufgerufen wurde, bis zum Wurzelknoten, und aktualisert dabei den Inhalt aller `DerivationDepth`-Variablen. Dies ist z.B. nach dem Austausch eines Unterbaumes nötig. In diesem Fall muß diese Funktion im Mutterknoten des Startknotens des ausgetauschten Baumes aufgerufen werden.

```
    /*------------------------------------------------------
      int TreeNode::DerivationDepthUpdate()

      Updating of the derivation depth from this point
 5    on up to the RootNode.
      ------------------------------------------------------
    */
    void TreeNode::DerivationDepthUpdate()
    {
10          int i;

            DerivationDepth = (NoOfDerivSymbols>0);

            for(i=0;i<NoOfDerivSymbols;i++)
15                  DerivationDepth =
                    DerivationDepth + Derivs[i]->DerivationDepth;

            if(PredecessorNode!=NULL)
                    PredecessorNode->DerivationDepthUpdate();
20  }
```

In Zeile 12 wird zunächst geprüft, ob das Symbol dieses Knotens ein Nichtterminalsymbol ist. Je nachdem wird `DerivationDepth` mit eins oder null initialisiert. Sodann werden in der in Zeile 14 beginnenden Schleife, so vorhanden, alle Ableitungsknoten durchwandert und die vermerkte Ableitungstiefe dieser Knoten zur Ableitungstiefe dieses Knotens hinzuaddiert (Zeilen 15 und 16). Die Ableitungstiefe dieses Knotens ist somit aktualisiert. In Zeile 18 wird anschließend geprüft, ob es einen Vorgängerknoten (Mutterknoten) gibt, und gegebenenfalls wird die Funktion `DerivationDepthUpdate` dort rekursiv aufgerufen.

6.2.3.8 AddCompleteSubTreesTo(Population*, unsigned int)

Diese Funktion sucht rekursiv alle vollständigen Unterbäume, die die im zweiten Parameter angegebene minimale Ableitungstiefe nicht unterschreiten, und fügt diese zum virtuellen Teil der im ersten Parameter referenzierten Population hinzu. Vollständige Unterbäume sind in diesem Zusammenhang Teilbäume, die mit einem, ein Startsymbol beinhalteten Knoten (also `StartNode` oder `RootNode`), beginnen.

```
    /*------------------------------------------------------
      virtual void TreeNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                             unsigned int MinDepth)

 5    Adding all complete subtrees (trees starting with a
      StartNode) with a depth larger or equal MinDepth
      to the ThePopulation.
      -----------------------------------------------------
    */
10  void TreeNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                     unsigned int MinDepth)
    {
            int i;

15          for(i=0;i<NoOfDerivSymbols;i++)
            {
                    if(Derivs[i]->DerivationDepth>=MinDepth)
                            Derivs[i]->
                            AddCompleteSubTreesTo(ThePopulation,MinDepth);
20          }
    }
```

In der in Zeile 15 beginnenden Schleife werden alle Ableitungsknoten durchlaufen. Soferne die unter diesem Knoten liegenden Bäume die minimale Ableitungstiefe nicht unterschreiten (Zeile 17), wird die Funktion `AddCompleteSubTreesTo(...)` auch rekursiv für diese Knoten aufgerufen (Zeilen 18 und 19). Bei Knoten der Klasse `StartNode` handelt es sich dabei um die Funktion `StartNode::AddCompleteSubTreesTo(...)`, welche nicht nur weitersucht, sondern auch den betreffenden Knoten - da er ein Startknoten ist - der Population hinzufügt.

6.2.3.9 `NoOfSubTrees(SymbTabEntry*, unsigned int)`

Diese Funktion liefert die Anzahl der Unterbäume dieses Knotens, welche die im zweiten Parameter gegebene Ableitungstiefe nicht überschreiten, und deren Startknoten das im ersten Parameter referenzierte Symbol beinhalten. Zugleich wird entsprechend dieser Angaben der temporäre Index aufgebaut, mit dessen Hilfe die Funktion `StartNode::SubTree(SymbTabEntry, unsigned int, unsigned int)` einen bestimmten Startknoten dieser Unterbäume schnell auffinden kann (siehe Kapitel 6.2.2.9, Seite 127).

```
     /*-------------------------------------------------------
       unsigned int TreeNode::NoOfSubTrees(SymbTabEntry* SelectedSymbol,
                                           unsigned int MaxDerivDepth)

 5     Returning the number of subtrees, starting
       with the SelectedSymbol, not exceeding the
       given maximum derivation depth.

       Builds up a temporary index to be used
10     by the method SubTree(SelectedSymbol)
       -------------------------------------------------------
     */
     unsigned int TreeNode::NoOfSubTrees(SymbTabEntry* SelectedSymbol,
                                         unsigned int MaxDerivDepth)
15   {
             int i;

             TempIndex = 0;

20           for(i=0;i<NoOfDerivSymbols;i++)
                     if(Derivs[i]->NoOfDerivSymbols>0)
                             TempIndex = TempIndex +
                             Derivs[i]->NoOfSubTrees(SelectedSymbol, MaxDerivDepth);

25           if(Symbol==SelectedSymbol && DerivationDepth<=MaxDerivDepth)
                     TempIndex++;

             return(TempIndex);
     }
```

In Zeile 18 wird zunächst der temporäre Index dieses Knotens mit null initialisiert. In der in Zeile 20 beginnenden Schleife werden dann alle Ableitungsknoten durchwandert, und - soferne es sich um Knoten handelt, die ein Nichtterminalsymbol beinhalten (Zeile 21) - die von der rekursiv aufgerufenen Funktion `NoOfSubTrees(SymbTabEntry, unsigned int)` gelieferten Indexwerte dieser Knoten zum temporären Index hinzuaddiert (Zeilen 22 und 23). In Zeile 25 wird schließlich geprüft, ob dieser Knoten ebenfalls das gesuchte Symbol beinhaltet und die geforderte maximale Ableitungstiefe nicht

überschreitet. Ist dies der Fall, dann ist der an dieser Stelle beginnende Teilbaum mitzuindexieren (Zeile 26). Der Wert des temporären Index dieses Knotens gibt nun also an, wieviele Bäume sich unterhalb dieses Knotens befinden, die die gewählte Ableitungstiefe nicht überschreiten, und das gesuchte Symbol beinhalten. In Zeile 28 wird die Funktion, mit dem errechneten temporären Index als Rückgabewert, beendet.

6.2.3.10 Depth()

Diese Funktion liefert die Höhe des Ableitungsbaumes, und wurde lediglich zu experimentellen Zwecken implementiert. Achtung! Die Höhe des Baumes ist ungleich der Ableitungstiefe! Die Ableitungstiefe ist definiert als die Anzahl der Nichtterminalsymbole, wohingegen die Höhe die Anzahl der nötigen Ableitungsschritte zum tiefsten Knoten beschreibt (das Niveau des tiefsten Knotens) [Duden Informatik, 1993, S. 56].

```
    /*------------------------------------------------------
      unsigned int TreeNode::Depth()

      Returning the depth (!= derivation depth) of this
 5    (sub-)tree.
      Implemented for experimental use only.
      ------------------------------------------------------
    */
    unsigned int TreeNode::Depth()
10  {
            unsigned int CurrentDepth;
            unsigned int ThisDepth;
            int i;

15          ThisDepth = 0;

            for(i=0;i<NoOfDerivSymbols;i++)
            {
                    CurrentDepth = Derivs[i]->Depth();
20
                    if(CurrentDepth>=ThisDepth)
                            ThisDepth = CurrentDepth + 1;
            }

25          return(ThisDepth);
    }
```

In Zeile 15 wird die temporäre Variable `ThisDepth`, die nach der Berechnung die Höhe des an dieser Stelle beginnenden Baumes beinhalten soll, mit null initialisert. In der in Zeile 17 beginnenden Schleife werden die Ableitungsknoten dieses Knotens durchwandert. Der temporären Variablen `CurrentDepth` wird bei jedem Durchlauf die Höhe des Baumes, der mit dem gerade behandelten Ableitungsknoten beginnt, zugewiesen (Zeile 19). Dies geschieht durch rekursiven Aufruf der Funktion `Depth()` im gerade behandelten Ableitungsknoten. Danach wird in Zeile 21 geprüft, ob der Baum, der mit dem momentan behandelten Ableitungsknoten beginnt, eine größere Höhe aufweist, als die bislang für diesen Knoten ermittelte Höhe `ThisDepth`. In diesem Fall wird die Variable `ThisDepth` auf die Höhe dieses Unterbaumes plus eins aktualisiert. In Zeile 25 schließlich wird die Funktion, unter Rückgabe des Ergebnisses, beendet.

6.2.3.11 CompareWith(TreeNode*)

Hierbei handelt es sich um eine Vergleichsfunktion, mit deren Hilfe überprüft werden kann, ob der an dieser Stelle beginnende Ableitungsbaum gleich dem Ableitungsbaum ist, der an dem durch den Parameter referenzierten Knoten beginnt.

```
    /*-------------------------------------------------------
      int TreeNode::CompareWith(TreeNode* TheOtherNode)

      Comparing the TreeNode with TheOtherNode,
 5    including all subtrees.
      -------------------------------------------------------
    */
    int TreeNode::CompareWith(TreeNode* TheOtherNode)
    {
10          int i;

            if(Symbol!=TheOtherNode->Symbol ||
            NoOfDerivSymbols!=TheOtherNode->NoOfDerivSymbols)
                    return(FALSE);
15          else
                    for(i=0;i<NoOfDerivSymbols;i++)
                            if(!Derivs[i]->CompareWith(TheOtherNode->Derivs[i]))
                                    return(FALSE);
            return(TRUE);
20  }
```

In Zeile 12 wird zunächst geprüft, ob dieser Knoten ein anderes Symbol beinhaltet als der Vergleichsknoten, oder ob für diesen Knoten eine andere Anzahl von Ableitungsknoten vermerkt ist als im Vergleichsknoten. Sollte dies der Fall sein, dann sind die zu vergleichenden Ableitungsbäume an dieser Stelle unterschiedlich, und es wird unter Rückgabe von `FALSE` abgebrochen (Zeile 13). Andernfalls werden in der in Zeile 16 beginnenden Schleife alle Ableitungsknoten dieses Knotens und des Vergleichsknotens durchwandert, und die mit diesen Knoten beginnenden Unterbäume durch rekursiven Aufruf von `CompareWith(TreeNode*)` miteinander verglichen (Zeile 17). Ergeben sich dabei zwei unterschiedliche Bäume, so wird die Funktion, unter Rückgabe von `FALSE`, abgebrochen. Wurden keine unterschiedlichen Unterbäume entdeckt, dann wird die Funktion mit `TRUE` als Rückgabewert beendet.

6.2.3.12 BuildUpPhenoTypeStringIn(char*, int&)

Diese Funktion ist eine Hilfsfunktion für die Funktion `GenoTypeString()` der Klasse `StartNode`. Sie baut im durch den ersten Parameter referenzierten Zeichenarray an der durch den zweiten Parameter gekennzeichneten Position einen String auf, der dem Phänotypen des an dieser Stelle beginnenden Ableitungsbaumes entspricht. Der String wird nicht durch ein Stringende-Zeichen abgeschlossen. Der zweite Parameter der Funktion wird in einem "call by reference" übergeben. Es handelt sich um eine globale Zählvariable, die von der ersten aufrufenden Instanz zur Verfügung gestellt werden muß.

```
    /*-------------------------------------------------------
      void TreeNode::BuildUpPhenoTypeStringIn(char* ResultString, int& Position)

      Building up the phenotype-string-representation in the
 5    ResultString-array. Position is the position of the
      current character
      -------------------------------------------------------
    */
    void TreeNode::BuildUpPhenoTypeStringIn(char* ResultString, int& Position)
10  {
            int i;

            if(NoOfDerivSymbols==0)
            {
15                  ResultString[Position] = '\0';
                    strcat(ResultString,Symbol->Symbol);
                    Position+= strlen(Symbol->Symbol);
            }
            else
20          {
                    for(i=0;i<NoOfDerivSymbols;i++)
                            Derivs[i]->BuildUpPhenoTypeStringIn(ResultString,Position);
            }
    }
```

In Zeile 13 wird geprüft, ob dieser Knoten ein Terminalsymbol beinhaltet. Sollte dies der Fall sein, so wird der String an der momentanen Position mit einem Stringende-Zeichen abgeschlossen (Zeile 15). In Zeile 16 wird mit Hilfe der Funktion `strcat(char*)` die Textrepräsentation des Symbols dem String an der Stelle des Stringende-Zeichens angefügt. Schließlich wird in Zeile 17 der Positionszähler um die Länge der Textrepräsentation des Symbols erhöht. Beinhaltet dieser Knoten jedoch ein Nichtterminalsymbol, so werden in der in Zeile 21 beginnenden Schleife die `BuildUpPhenoTyoeStringInt(...)`-Funktionen der Ableitungsknoten rekursiv aufgerufen. Die Funktion wird also rekursiv so lange aufgerufen, bis sie im Ableitungsbaum bei einem ein Terminalsymbol beinhaltenden Knoten angelangt ist. Dieses Terminalsymbol wird dann dem bisherigen String hinzugefügt, was letzlich im Aufbau des kompletten Phänotyp-Strings mündet.

6.2.3.13 PhenoTypeStringLength()

Diese Funktion ist ebenfalls eine Hilfsfunktion der Funktion `GenoTypeString()` der Klasse `StartNode`. Sie berechnet die Länge des Phänotyp-Strings (ohne Stringende-Zeichen), der sich aus dem an diesem Knoten beginnenden Teil-Baum ergibt.

```
    /*-------------------------------------------------------
      int TreeNode::PhenoTypeStringLength()

      Computing the length of the phenotype-string-representation
 5    of this tree from this node on, not counting the /0
      character
      -------------------------------------------------------
    */
    int TreeNode::PhenoTypeStringLength()
10  {
            int i;
            int j;
```

```
            i = 0;
15          if(NoOfDerivSymbols==0)
                    i+= strlen(Symbol->Symbol);
            else
                    for(j=0;j<NoOfDerivSymbols;j++)
                            i+= Derivs[j]->PhenoTypeStringLength();
20
            return(i);
    }
```

In Zeile 14 wird zunächst die temporäre Variable i mit null initialisiert. In Zeile 15 wird sodann geprüft, ob dieser Knoten ein Terminalsymbol beinhaltet. Sollte dies der Fall sein, dann wird die Länge der Stringrepräsentation des Symbols zu i hinzuaddiert. Andernfalls werden in der in Zeile 18 beginnenden Schleife alle Ableitungsknoten durchwandert, und die jeweiligen Stringlängen - ermittelt durch rekursiven Aufruf der Funktion `PhenoTypeStringLength(...)` in diesen Knoten - zur Variablen i hinzuaddiert (Zeile 19). In Zeile 21 wird das nun in der Variablen i zur Verfügung stehende Ergebnis zurückgegeben, und die Funktion beendet.

6.2.3.14 BuildUpGenoTypeStringIn(char*, int&, int)

Diese Funktion ist eine Hilfsfunktion für die Funktion `PhenoTypeString()` der Klasse `StartNode`. Sie baut im durch den ersten Parameter referenzierten Zeichenarray an der durch den zweiten Parameter gekennzeichneten Position einen String auf, der dem Genotypen des an dieser Stelle beginnenden Ableitungsbaumes entspricht. Der erzeugte String wird nicht von einem Stringende-Zeichen abgeschlossen. Der zweite Parameter der Funktion wird in einem "call by reference" übergeben. Es handelt sich um eine globale Zählvariable, die von der ersten aufrufenden Instanz zur Verfügung gestellt werden muß. Der dritte Parameter legt fest, ob der Genotyp in der kurzen oder in der langen Stringrepräsentation gefordert ist (siehe Kapitel 5.3.5 und Kapitel 5.3.7, Seite 64). Je nachdem nimmt dieser Parameter den Wert `SHORT` oder `LONG` an.

```
    /*-----------------------------------------------------------
      void TreeNode::BuildUpGenoTypeStringIn(char* ResultString,
                                             int& Position,
                                             int Representation)
 5
      Building up the genotype-string-representation in the
      ResultString-array. Position is the position of the
      current character
      -----------------------------------------------------------
10  */
    void TreeNode::BuildUpGenoTypeStringIn(char* ResultString,
                                           int& Position,
                                           int Representation)
    {
15          int i;
            int n;

            if(Symbol!=NULL)
            {
20                  if(Representation==LONG)
```

```
                {
                        if(NoOfDerivSymbols>0)
                        {
                                n = int(log10(NoOfDerivSymbols));
25                              for(i=n;i>=0;i--)
                                {
                                        ResultString[Position] = '0' +
                                        (
                                                int(NoOfDerivSymbols/pow(10,i)) -
30                                              10*int(NoOfDerivSymbols/pow(10,(i+1)))
                                        );
                                        Position++;
                                }
                                ResultString[Position] = '<';
35                              Position++;
                                ResultString[Position] = '\0';
                                strcat(ResultString,Symbol->Symbol);
                                Position+= strlen(Symbol->Symbol);
                                ResultString[Position] = '>';
40                              Position++;
                        }
                        else
                        {
                                ResultString[Position] = '"';
45                              Position++;
                                ResultString[Position] = '\0';
                                strcat(ResultString,Symbol->Symbol);
                                Position+= strlen(Symbol->Symbol);
                                ResultString[Position] = '"';
50                              Position++;
                        }
                }
                else
                {
55                      if(NoOfDerivSymbols>0)
                        {
                                n = int(log10(DerivationNo+1));
                                for(i=n;i>=0;i--)
                                {
60                                      ResultString[Position] = '0' +
                                        (
                                                int((DerivationNo+1)/pow(10,i)) -
                                                10*int((DerivationNo+1)/pow(10,(i+1)))
                                        );
65                                      Position++;
                                }
                                ResultString[Position] = '.';
                                Position++;
                        }
70              }
                for(i=0;i<NoOfDerivSymbols;i++)
                        Derivs[i]->
                        BuildUpGenoTypeStringIn(ResultString,Position,Representation);
        }
75  }
```

In Zeile 18 wird zunächst geprüft, ob dieser Knoten überhaupt ein Symbol beinhaltet (Normalfall). Sodann wird in Zeile 20 unterschieden, ob die lange oder die kurze Stringrepräsentation gefordert ist.

Ist die lange Stringrepräsentation gefordert, so wird mit Zeile 22 fortgesetzt. Hier wird geprüft, ob dieser Knoten ein Nichtterminalsymbol beinhaltet. Ist dies der Fall, dann ist die Anzahl der Ableitungssymbole, gefolgt vom spitz eingeklammerten Symbol selbst,

dem String hinzuzufügen. Dies geschieht in den Zeilen 24-40: Zuerst wird in Zeile 24 bestimmt, wieviele Stellen die Dezimaldarstellung der Zahl hat, die die Anzahl der Ableitungssymbole angibt. Dann wird in der in Zeile 25 beginnenden Schleife diese Zahl, Ziffer für Ziffer, dem String hinzugefügt (die Formel in den Zeilen 29 und 30 extrahiert die jeweilige Ziffer n), und nach jeder Ziffer wird der Positionszähler aktualisiert. Danach wird die "spitze Klammer auf" angehängt (Zeile 34) und der Positionszähler inkrementiert (Zeile 35). An die Stelle, an der nun das Symbol folgen muß, wird zunächst das Stringende-Zeichen geschrieben (Zeile 36), damit im Anschluß daran das Symbol selber mit Hilfe der Funktion `strcat(char*)` angehängt werden kann (Zeile 37). In Zeile 38 wird der Positionszähler entsprechend erhöht. Die fehlende "spitze Klammer zu" wird in den Zeilen 39 und 40 dem String hinzugefügt. Handelt es sich bei dem Symbol dieses Knotens jedoch um ein Terminalsymbol, dann wird einfach das unter Anführungszeichen gesetzte Symbol selbst dem String hinzugefügt. Dies geschieht, nach dem selben Prinzip wie zuvor, in den Zeilen 44-50.

Ist nicht die lange, sondern die kurze Stringrepräsentation gefordert, dann setzt das Programm nach der Zeile 20 mit der Zeile 52 fort. In Zeile 55 wird dann zunächst geprüft, ob das Symbol dieses Knotens ein Nichtterminalsymbol ist. Ist dies der Fall, dann wird in Zeile 57 bestimmt, wieviele Stellen die Dezimaldarstellung der Zahl hat, die die Nummer der Ableitung dieses Symbols angibt. Dann wird in der in Zeile 58 beginnenden Schleife diese Zahl, Ziffer für Ziffer, dem String hinzugefügt (die Formel in den Zeilen 62 und 63 extrahiert die jeweilige Ziffer n), und nach jeder Ziffer der Positionszähler aktualisiert. In den Zeilen 76 und 86 wird schließlich dem String noch ein Punkt hinzugefügt, und der Positionszähler wiederum aktualisiert.

In jedem Fall setzt die Funktion in Zeile 71 fort, wo in einer Schleife, so vorhanden, alle Ableitungsknoten durchwandert werden. In dieser Schleife wird bei jedem Ableitungsknoten die Funktion `BuildUpGenoTypeStringIn(...)` rekursiv aufgerufen. Dies mündet letztlich im vollständigen Aufbau des gewünschten Strings.

6.2.3.15 `GenoTypeStringLength(int)`

Diese Funktion ist ebenfalls eine Hilfsfunktion der Funktion `PhenoTypeString()` der Klasse `StartNode`. Sie berechnet die Länge des Genotyp-Strings (ohne Stringende-Zeichen), der sich aus dem an diesem Knoten beginnenden Teil-Baum ergibt. Der Parameter gibt an, ob die Länge des langen oder des kurzen Strings zu errechnen ist. Je nachdem nimmt dieser Parameter den Wert `SHORT` oder `LONG` an.

```
    /*-------------------------------------------------------
      int TreeNode::GenoTypeStringLentgh(int Representation)

      Computing the length of the genotype-string-representation
5     of this tree from this node on, not counting the /0
      character. Representation may be LONG or SHORT
      -------------------------------------------------------
    */
    int TreeNode::GenoTypeStringLength(int Representation)
```

```
10  {
            int i;
            int j;

            i = 0;
15          if(Symbol!=NULL)
            {
                    if(Representation==LONG)
                    {
                            if(NoOfDerivSymbols>0)
20                          {
                                    i = 1 + int(log10(NoOfDerivSymbols));
                                    i = i + 2 + strlen(Symbol->Symbol);
                            }
                            else
25                                  i = i + 2 + strlen(Symbol->Symbol);
                    }
                    else
                            if(NoOfDerivSymbols>0)
                                    i = 2 + int(log10(DerivationNo+1));
30
                    for(j=0;j<NoOfDerivSymbols;j++)
                            i+= Derivs[j]->GenoTypeStringLength(Representation);
            }

35          return(i);
    }
```

In Zeile 14 wird zunächst die temporäre Variable i mit null initialisiert. Diese Variable soll das Ergebnis der Berechnung aufnehmen. Dann wird in Zeile 15 geprüft, ob dieser Knoten überhaupt ein Symbol beinhaltet (Normalfall). Ist dies der Fall, dann wird in Zeile 17 unterschieden, ob die Länge der langen, oder der kurzen Stringrepräsentation gefordert ist.

Ist die Länge der langen Stringrepräsentation gefordert, wird das Programm in Zeile 19 fortgesetzt. Hier wird geprüft, ob es sich beim Symbol dieses Knotens um ein Nichttermialsymbol handelt. Trifft dies zu, dann wird in Zeile 21 die Länge der Dezimalzahl, die die Anzahl der Ableitungssymbole angibt, zu i dazuaddiert. In Zeile 22 wird i noch um die Länge der Stringdarstellung des Symbols plus 2 (für die beiden spitzen Klammern) erhöht. Handelt es sich beim Symbol dieses Knotens um ein Terminalsymbol, dann wird i um die Länge der Stringrepräsentation plus 2 (für die beiden Anführungszeichen) erhöht (Zeile 25).

Ist jedoch die Länge der kurzen Stringrepräsentation gefordert, dann wird das Programm nach der Zeile 17 in der Zeile 28 fortgesetzt. In dieser Zeile wird wieder getestet, ob das Symbol dieses Knotens ein Nichtterminalsymbol ist. Trifft dies zu, dann wird i um die Länge der Dezimalzahl, die die Nummer der Ableitung angibt, plus eins (für den Punkt) erhöht.

In jedem Fall setzt das Programm in Zeile 31 fort. In der in dieser Zeile beginnenden Schleife werden alle Ableitungsknoten durchwandert, und die durch rekursiven Aufruf der Funktion `GenoTypeStringLength(int)` ermittelten Stringlängen der Ableitungsknoten, zu i dazuaddiert.

6.2.3.16 EnhanceWith(ProdTabEntry*)

Diese Funktion erweitert diesen Knoten mit der durch den Parameter referenzierten Ableitung `ProdTabEntry` der Grammatik.

```
    /*-------------------------------------------------------
      void TreeNode::EnhanceWith(ProdTabEntry* Derivation)

      Enhancing the TreeNode with the given Derivation
 5    -------------------------------------------------------
    */
    void TreeNode::EnhanceWith(ProdTabEntry* Derivation)
    {
            int i;
10          DerivSymbol* CurrentDerivSymbol;

            NoOfDerivSymbols = Derivation->NoOfDerivSymbols;
            DerivationNo = int(Derivation - Symbol->DerivList);
            Derivs = new TreeNode*[NoOfDerivSymbols];
15
            CurrentDerivSymbol = Derivation->FirstSymbol;
            for(i=0;i<NoOfDerivSymbols;i++)
            {
                    if(CurrentDerivSymbol->Symbol==TreeLanguage->StartSymbol)
20                          Derivs[i] =
                            new StartNode(TreeLanguage, this, Root);
                    else
                            Derivs[i] =
                            new TreeNode(CurrentDerivSymbol->Symbol,
25                                       TreeLanguage, this, Root);

                    CurrentDerivSymbol = CurrentDerivSymbol->NextSymbol;
            }
    }
```

Zunächst wird in Zeile 11 die Anzahl der Ableitungsknoten und in Zeile 12 die Nummer der Ableitung gesetzt. In Zeile 13 wird ein entsprechend großes Array `Derivs` zur Aufnahme der Ableitungsknoten-Pointer erzeugt. In der in Zeile 17 beginnenden Schleife werden alle Symbole der durch den Parameter referenzierten Ableitung durchwandert. In Zeile 19 wird für jedes Symbol geprüft, ob es sich um ein Startsymbol handelt. Ist dies der Fall, dann wird der Baum um ein Objekt der Klasse `StartNode`, welches dieses Startsymbol beinhaltet, erweitert. (Zeilen 20 und 21). Ansonsten wird ein Ableitungsknoten der Klasse `TreeNode` erzeugt (Zeilen 23-25).

6.2.3.17 RandomEnhance(unsigned int, unsigned int&)

Diese Funktion erweitert durch rekursiven Selbstaufruf diesen Knoten zu einem Ableitungsbaum, wobei die Ableitungen aus den für das jeweilige Symbol zur Verfügung stehenden Ableitungen zufällig ausgewählt werden. Der erste Parameter gibt die maximal zulässige Ableitungstiefe (definiert als die Anzahl der Nichtterminalknoten im Baum) an. Der zweite Parameter stellt eine globale Zählvariable dar, die die bereits erzeugten Nichtterminalknoten mitzählt. Sie wird durch einen "call by reference" übergeben und muß von der aufrufenden Instanz zur Verfügung gestellt werden.

```
    /*------------------------------------------------------
      int TreeNode::RandomEnhance(unsigned int MaxDerivDepth,
                                  unsigned int &NoOfNonTerminalNodes)

 5    Enhancing the TreeNode with randomly selected
      derivations out of the related production table
      entry.

      Returning FALSE in case the random enhancement
10    has failed because of exceeding the given maximum
      derivation depth.

      NoOfNonTerminalNodes is kind of a global counter,
      countig the non-terminal symbols so far.
15    ------------------------------------------------------
    */
    int TreeNode::RandomEnhance(unsigned int MaxDerivDepth,
                                unsigned int &NoOfNonTerminalNodes)
    {
20          int i;

            static Uniran Random(RANDTREESEED);

            if(Symbol->DerivList!=NULL)
25          {
                    EnhanceWith(Symbol->
                    DerivList+Random.dice(Symbol->NoOfDerivs));

                    DerivationDepth = (NoOfDerivSymbols>0);
30                  for(i=0;i<NoOfDerivSymbols;i++)
                    {
                            if(Derivs[i]->Symbol->NoOfDerivs>0)
                            {
                                    NoOfNonTerminalNodes++;
35                                  if(NoOfNonTerminalNodes>MaxDerivDepth)
                                            return(FALSE);
                                    else
                                    {
                                            if(!Derivs[i]->RandomEnhance(
40                                            MaxDerivDepth, NoOfNonTerminalNodes))
                                                    return(FALSE);

                                            DerivationDepth =
                                            DerivationDepth +
45                                          Derivs[i]->DerivationDepth;
                                    }
                            }
                    }
            }
50         return(TRUE);
    }
```

In Zeile 24 wird zunächst geprüft, ob dieses Symbol ein Terminalsymbol oder ein Nichtterminalsymbol ist. Handelt es sich um ein Terminalsymbol, dann werden die Zeilen 25-49 übersprungen, und die Funktion, unter Rückgabe von `TRUE`, abgebrochen. Ansonsten wird in Zeile 26 fortgefahren. Hier wird zunächst, unter Zuhilfenahme der Funktion `EnhanceWith(ProdTabEntry*)`, dieser Knoten mit einer zufällig gewählten Ableitung des Knotensymbols erweitert. Die Zufallsauswahl geschieht durch die Funktion `dice(int)` des Zufallszahlengenerators `Uniran`. Die Funktion `dice(int)` liefert eine ganzzahlige Zufallszahl zwischen null (inklusive) und dem Parameterwert (exklusive). Die Ableitungstiefe des Knotens beträgt nun nicht mehr null, sondern mindestens

eins, was in `DerivationDepth` vermerkt wird (Zeile 29). In der in Zeile 30 beginnenden Schleife werden anschließend alle erzeugten Ableitungsknoten durchwandert. Für jeden Knoten wird dabei geprüft, ob es sich um einen Nichtterminalknoten handelt (Zeile 32). In diesem Fall wird der globale Zähler `NoOfNonTerminalNodes`, der die Anzahl der bereits erzeugten Nichtterminalknoten zählt, um eins erhöht (Zeile 34). Wird durch den anschließenden Vergleich in Zeile 35 entdeckt, daß die maximal zulässige Ableitungstiefe bereits überschritten ist, dann bricht die Funktion, unter Rückgabe von `FALSE`, ab (Zeile 36). Andernfalls wird in den Zeilen 39 und 40 dieser Ableitungsknoten durch rekursiven Aufruf der Funktion `RandomEnhance(...)` wiederum erweitert. Das Ergebnis der Erweiterung wird sogleich ausgewertet, und falls die Erweiterung nicht erfolgreich war, wird die Funktion, unter Rückgabe von `FALSE`, abgebrochen (Zeile 41). Andernfalls wird die gespeicherte Ableitungstiefe dieses Knotens `DerivationDepth` entsprechend aktualisert (Zeilen 43-45). Nach erfolgreicher Erweiterung wird die Funktion, unter Rückgabe von `TRUE`, beendet.

6.2.3.18 EnhanceWith(TreeNode*)

Diese Funktion erweitert diesen Baumknoten `TreeNode` mit der Vorlage des Baumes, der mit dem im Parameter referenzierten Knoten beginnt. Dies geschieht unter rekursiver Verwendung der Funktionen `StartNode::EnhanceWith(StartNode*)` und `TreeNode::EnhanceWith(TreeNode*)`.

```
    /*------------------------------------------------------
      void TreeNode::EnhanceWith(TreeNode* TheOtherNode)

      Enhancing the TreeNode with the derivation given
 5    by TheOtherNode.
      ------------------------------------------------------
    */
    void TreeNode::EnhanceWith(TreeNode* TheOtherNode)
    {
10          int i;

            NoOfDerivSymbols = TheOtherNode->NoOfDerivSymbols;
            DerivationNo = TheOtherNode->DerivationNo;
            Derivs = new TreeNode*[NoOfDerivSymbols];
15
            for(i=0;i<NoOfDerivSymbols;i++)
            {
                    if(TheOtherNode->Derivs[i]->Symbol==TreeLanguage->StartSymbol)
                            Derivs[i] =
20                          new StartNode(TreeLanguage, this, Root);
                    else
                            Derivs[i] =
                            new TreeNode(TheOtherNode->Derivs[i]->Symbol,
                                         TreeLanguage, this, Root);
25
                    Derivs[i]->DerivationDepth =
                    TheOtherNode->Derivs[i]->DerivationDepth;

                    TempIndex = 0;
30
                    if(TheOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                            Derivs[i]->EnhanceWith(TheOtherNode->Derivs[i]);
            }
    }
```

In Zeile 12 wird zunächst die Anzahl der Ableitungsknoten und in Zeile 13 die Nummer der Ableitung übernommen. In Zeile 14 wird ein entsprechend großes Array zur Aufnahme der Ableitungsknoten-Pointer geschaffen. In der in Zeile 16 beginnenden Schleife werden die Ableitungsknoten des zu kopierenden Knotens durchgegangen. In Zeile 18 wird geprüft, ob der momentan behandelte Ableitungsknoten ein Startknoten (`StartNode`) ist. In diesem Fall wird der Ableitung dieses Knotens ebenfalls ein `StartNode` hinzugefügt (Zeilen 19 und 20). Andernfalls handelt es sich um einen `TreeNode` (Zeilen 22-24). In den Zeilen 26-29 werden die Ableitungstiefe und der temporäre Index übernommen. Schließlich wird in Zeile 31 geprüft, ob der soeben kopierte Ableitungsknoten selber Ableitungen besitzt. In diesem Fall wird die Funktion `EnhanceWith(...)` rekursiv aufgerufen.

6.2.3.19 XEnhanceWith(TreeNode*, TreeNode*, TreeNode*)

Diese Funktion ist eine Hilfsfunktion der Funktion `RootNode::CrossOver(...)`. Sie erweitert durch rekursiven Selbstaufruf diesen Knoten zu einem vollständigen Ableitungsbaum. Dabei nimmt sie zunächst den durch den ersten Pararameter referenzierten Baum als Vorlage. Wenn der Kreuzungsknoten, referenziert durch den zweiten Parameter, erreicht ist, wird der Erweiterungsvorgang für diesen Kreuzungsknoten - unter Zuhilfenahme der Funktion `EnhanceWith(TreeNode*)` - mit dem durch den dritten Parameter referenzierten Teilbaum fortgesetzt. An die aufrufende Instanz gibt diese Funktion einen Pointer auf den neuen Kreuzungsknoten zurück.

```
    /*-------------------------------------------------------
      TreeNode* TreeNode::XEnhanceWith(TreeNode* TheFirstOtherNode,
                                       TreeNode* TheCrossOverPoint,
                                       TreeNode* TheSecondOtherNode)
 5
      Enhancing the TreeNode recursively with the derivation
      given by TheFirstOtherNode, until during this copying
      TheCrossOverPoint occurs. From this point on, the
      enhancement is continued recursively
10    until the end of the tree.
      -------------------------------------------------------
    */
    TreeNode* TreeNode::XEnhanceWith(TreeNode* TheFirstOtherNode,
                                     TreeNode* TheCrossOverPoint,
15                                   TreeNode* TheSecondOtherNode)
    {
            int i;
            TreeNode* NewCrossOverPoint;

20          if(TheFirstOtherNode==TheCrossOverPoint)
            {
                    NewCrossOverPoint = this;
                    EnhanceWith(TheSecondOtherNode);
            }
25
            else

            {
                    NoOfDerivSymbols = TheFirstOtherNode->NoOfDerivSymbols;
30                  DerivationNo = TheFirstOtherNode->DerivationNo;
                    Derivs = new TreeNode*[NoOfDerivSymbols];
```

```
                    NewCrossOverPoint = NULL;

                    for(i=0;i<NoOfDerivSymbols;i++)
35                  {
                            if(
                            TheFirstOtherNode->Derivs[i]->Symbol==
                            TreeLanguage->StartSymbol)
                                    Derivs[i] =
40                                  new StartNode(TreeLanguage, this, Root);
                            else
                                    Derivs[i] =
                                    new TreeNode(TheFirstOtherNode->Derivs[i]->Symbol,
                                            TreeLanguage, this, Root);
45
                            Derivs[i]->DerivationDepth =
                            TheFirstOtherNode->Derivs[i]->DerivationDepth;

                            Derivs[i]->TempIndex = 0;
50
                            if(TheFirstOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                                    if(NewCrossOverPoint==NULL)
                                            NewCrossOverPoint =
                                            Derivs[i]->
55                                          XEnhanceWith(TheFirstOtherNode->Derivs[i],
                                            TheCrossOverPoint, TheSecondOtherNode);
                                    else
                                            Derivs[i]->
                                            XEnhanceWith(TheFirstOtherNode->Derivs[i],
60                                          TheCrossOverPoint,TheSecondOtherNode);
                    }
            }

            return(NewCrossOverPoint);
65  }
```

In Zeile 20 wird zunächst geprüft, ob der Startknoten des durch den ersten Parameter referenzierten Baumes gleichzeitig schon der Kreuzungspunkt ist. In diesem Fall ist dieser Knoten der neue Kreuzungspunkt, und die lokale Pointer-Variable `NewCrossOverPoint`, die den Rückgabewert der Funktion aufzunehmen hat, wird auf genau diesen Knoten gesetzt (Zeile 22). Sodann wird dieser Knoten unter Zuhilfenahme der Funktion `EnhanceWith(TreeNode*)` mit dem durch den ersten Parameter referenzierten Baum erweitert. Die Zeilen 28-62 werden übersprungen, und die Funktion, unter Rückgabe des Pointers auf den neuen Kreuzungsknoten, beendet.

Fallen der im ersten Parameter referenzierte Knoten und der im zweiten Parameter referenzierte Kreuzungsknoten nicht zusammen, so beginnt das Programm in Zeile 29. Dort werden für diesen Knoten zunächst die Anzahl der Ableitungsknoten (Zeile 29) und die Ableitungsnummer (Zeile 30) unter Vorlage des im ersten Parameter referenzierten Knotens eingestellt. In Zeile 31 wird ein entsprechend großes `Derivs`-Array zur Aufnahme der Ableitungsknoten-Pointer erzeugt. Die lokale Pointer-Variable zur Aufnahme des Rückgabewertes wird vorerst auf `NULL` gestellt, was bedeutet, daß der neue Kreuzungspunkt noch nicht erzeugt wurde (Zeile 32). In der in Zeile 34 beginnenden Schleife werden nun (so vorhanden) alle Ableitungsknoten des durch den ersten Parameter referenzierten Knotens durchlaufen. Für jeden Ableitungsknoten wird geprüft, ob es sich um einen ein Startsymbol beinhaltenden Knoten handelt, oder nicht

(Zeilen 36-38). Je nachdem wird die Ableitung dieses Knotens um einen `StartNode` (Zeilen 39 und 40), oder um einen `TreeNode` erweitert (Zeilen 42-44). In den Zeilen 49-52 wird in jedem Fall die Ableitungstiefe und der temporäre Index dieses Knotens zugewiesen. In Zeile 51 wird sodann getestet, ob der Knoten, der als Vorlage dient, ein Nichtterminalsymbol beinhaltet. Ist dies der Fall, so wird in Zeile 52 getestet, ob innerhalb dieser Schleife der neue Kreuzungspunkt schon gefunden wurde. Wurde er noch nicht gefunden, dann wird die Funktion `XEnhanceWith(...)` rekursiv für den momentan aktuellen Ableitungsknoten aufgerufen, und der Rückgabewert der lokalen Pointer-Variablen `NewCrossOverPoint` zugewiesen. Andernfalls wird die Funktion `XEnhanceWith(...)` rekursiv für den momentan aktuellen Ableitungsknoten ohne Veränderung der Pointer-Variablen `NewCrossOverPoint` aufgerufen. Nach Beendigung der Schleife wird die Funktion, unter Rückgabe einer Referenz auf den eventuell inzwischen erzeugten neuen Kreuzungsknoten, beendet.

6.2.3.20 EnhanceWithString(char*, int&)

Diese Funktion ist eine Hilfsfunktion der Funktion `RootNode::Set(char*)`. Sie erweitert diesen Knoten anhand des im ersten Parameters referenzierten Strings. Dieser String muß den Ableitungsbaum in der kurzen Stringrepräsentation (siehe Kapitel 5.3.5) enthalten. Der zweite Parameter wird in einem "call by reference" übergeben, und ist eine globale Zählvariable, die von der aufrufenden Instanz zur Verfügung gestellt werden muß. Er gibt die momentan aktuelle Position im String an.

```
    /*-------------------------------------------------------
      int TreeNode::EnhanceWithString(char* TheString, int& Position)

      Enhancing this tree with the tree given in TheString from
 5    Position on. TheString must have the short phenotype format.
      -------------------------------------------------------
    */
    int TreeNode::EnhanceWithString(char* TheString, int& Position)
    {
10          int i;
            int j;
            int s;

            i = 0;
15          do
            {
                    switch(TheString[Position])
                    {
                            case '0':
20                                  i = 10*i + 0;
                                    break;
                            case '1':
                                    i = 10*i + 1;
                                    break;
25                          case '2':
                                    i = 10*i + 2;
                                    break;
                            case '3':
                                    i = 10*i + 3;
30                                  break;
                            case '4':
```

```
                        i = 10*i + 4;
                        break;
                case '5':
35                      i = 10*i + 5;
                        break;
                case '6':
                        i = 10*i + 6;
                        break;
40              case '7':
                        i = 10*i + 7;
                        break;
                case '8':
                        i = 10*i + 8;
45                      break;
                case '9':
                        i = 10*i + 9;
                        break;
                case '.':
50                      break;
                default:
                        return(FALSE);
            }
            Position++;
55      } while(TheString[Position]!='.');

        if(i>Symbol->NoOfDerivs)
            return(FALSE);

60      EnhanceWith(Symbol->DerivList+(i-1));

        s = TRUE;
        for(j=0;j<NoOfDerivSymbols;j++)
        {
65          if(Derivs[j]->Symbol->NoOfDerivs)
                s = Derivs[j]->EnhanceWithString(TheString, Position);
            if(!s)
                break;
        }
70      return(s);
    }
```

Zu Beginn wird erwartet, daß der Positionszähler auf die erste Ziffer einer Zahl im String zeigt. Die lokale Variable i, die diese Zahl aufnehmen soll, wird in Zeile 14 mit null initialisiert. In der in Zeile 15 beginnenden Schleife wird dann Ziffer für Ziffer verarbeitet, bis schließlich an der aktuellen Position ein Punkt steht und die Zahl somit ganz eingelesen wurde (Zeile 55). Die einzelnen Ziffern werden in der `switch`-Anweisung, beginnend in Zeile 17, ausgewertet. Jede Ziffer von null bis neun bewirkt, daß die bisher erkannte Ziffernfolge um eine Zehnerpotenz erhöht, und die neue Ziffer in der Einerstelle dazuaddiert wird (Zeilen 19-48). Auch ein Punkt wird akzeptiert (Zeilen 49 und 50). Alle anderen Zeichen führen, da es sich offensichtlich um einen syntaktisch nicht korrekten String handelt, zum vorzeitigen Abbruch der Funktion unter Rückgabe von `FALSE`. In Zeile 54 wird der Positionszähler bei jedem Schleifendurchlauf entsprechend inkrementiert. In Zeile 57 wird nach dem Einlesen der Zahl getestet, ob die eingelesene Zahl größer ist, als die Anzahl der möglichen Ableitungen des Symbols, das diesem Knoten zugeordnet ist. Sollte dies zutreffen, dann liegt offensichtlich ein Fehler vor, und die Funktion wird unter Rückgabe von `FALSE` abgebrochen (Zeile 58). Ansonsten wird dieser Knoten, unter Zuhilfenahme der Funktion `EnhanceWith(SymbTabEntry*)`

mit der angegebenen Ableitung erweitert. Sodann werden in der in Zeile 63 beginnenden Schleife, so vorhanden, alle Ableitungsknoten durchlaufen. Bei allen Ableitungsknoten, die Nichtterminalsymbole beinhalten (getestet in Zeile 65), wird die Funktion `EnhanceWithString(...)` rekursiv aufgerufen. Tritt bei einer dieser Erweiterungen ein Fehler auf (was aus den Rückgabewerten der aufgerufenen Funktionen ersichtlich ist, die in der lokalen Variablen s gespeichert werden), so wird diese Funktion, unter Rückgabe von `FALSE`, abgebrochen (Zeilen 67 und 68). Nach Beendigung der Schleife wird die Funktion, unter Rückgabe des Inhalts von s, der an dieser Stelle `TRUE` sein muß, beendet (Zeile 70).

6.2.3.21 `Evaluate(...)`

Diese Funktion kann vom Anwender implementiert werden, und soll das durch den an dieser Stelle beginnenden Ableitungsbaum dargestellte Individuum auswerten. Wie dies zu geschehen hat, sowie eine mögliche Implementierung für das XOR-Problem, ist bereits in Kapitel 5.2.2 ausführlich beschrieben worden. Für die folgenden Ausprägungen der Methode `TreeNode::Evaluate(...)` sind bereits Prototypeinträge im Headerfile `ga.h` vorgesehen.

```
int Evaluate();
int Evaluate(int);
int Evaluate(int, int);
int Evaluate(int, int, int);
int Evaluate(int, int, int, int);
float Evaluate(float);
float Evaluate(float, float);
float Evaluate(float, float, float);
float Evaluate(float, float, float, float);
```

6.3 Die Population

Die Klasse `Population` beinhaltet in ihrem Datenteil alle Individuen (i.e. Ableitungsbäume, beginnend mit einem `RootNode`-Objekt) einer Generation. Hinzu kommt die sogenannte "virtuelle" Population, die aus Pointern auf `StartNode`-Knoten der tatsächlichen Population besteht. Damit ist es möglich, Unterableitungsbäume in die Auswahl mit hineinzunehmen. Die entsprechenden Zusammenhänge sind in Abbildung 14 bereits gezeigt worden. Die meisten Zugriffe des Programms auf die Population erfolgen über die virtuelle Population. Selbst wenn also keine Aufnahme von Unterbäumen in den Selektionsprozeß erwünscht ist, muß es dennoch immer Mitglieder der virtuellen Population geben, die in diesem Fall genau die Wurzelknoten aller Mitglieder der tatsächlichen Population referenzieren. Die Klasse `Population` stellt außerdem alle Funktionen zur Fitnessberechnung und Fitness-Skalierung, sowie all die genetischen Operatoren zur Verfügung, die nicht auf Informationen aus anderen (Eltern-)Populationen angewiesen sind.

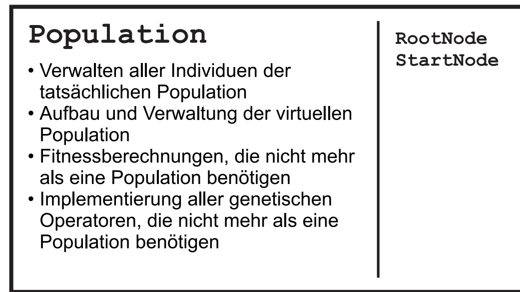


Abbildung 28: Die Klasse `Population`

6.3.1 Die Datenstruktur

Die im Datenteil der Klassendefintion der Klasse `Population` stehenden Variablen haben folgende Bedeutung:

- `ActualPopSize`: Gibt die Anzahl der Individuen in der tatsächlichen Population an.
- `Actual`: Referenziert das Array von `RootNode`-Objekten, die die einzelnen Individuen darstellen. Dieses Array beinhaltet also die tatsächliche Population.
- `ActualPopArraySize`: Gibt die Größe dieses Arrays an, die unter Umständen von der Populationsgröße `ActualPopSize` abweichen kann. (Es muß nicht immer das ganze Array mit Individuen gefüllt sein).
- `VirtualPopSize`: Gibt die Anzahl der durch die virtuelle Population referenzierten Individuen an.
- `Virtual`: Referenziert das Array von `StartNode`-Pointern, die auf Baumknoten von Individuen der tatsächlichen Population zeigen. Dieses Array beinhaltet also die virtuelle Population.
- `VirtualPopArraySize`: Gibt die Größe dieses Arrays an, die unter Umständen von der Populationsgröße `VirtualPopSize` abweichen kann. (Es muß nicht immer das ganze Array gefüllt sein).
- `MatingListNoOfEntries`: Gibt die Anzahl von Einträgen in der sogenannten "mating-list" an, die im `MatingList`-Array gehalten wird.
- `MatingList`: Referenziert ein Array von Objekten der Struktur `MatingListEntry`. Ein Objekt der Struktur `MatingListEntry` enthält zwei Pointer auf zwei Objekte der Klasse `StartNode`. Ein Eintrag in der "mating-list" referenziert also ein Paar von Individuen, welche einander durch Selektion und Mating zugeordnet wurden.

- `MatingListArraySize`: Gibt die Größe dieses Arrays an, die unter Umständen von der Anzahl von Einträgen in der Mating-List abweichen kann. (Es muß nicht immer das ganze Array gefüllt sein).
- `BestIndividual`: Ist ein Pointer auf das beste Individuum der Population.
- `WorstIndividual`: Ist ein Pointer auf das schlechteste Individuum der Population.
- `SmallesSubTreeOfVirtualPop`: Gibt die kleinste Ableitungstiefe an, mit der ein Teilableitungsbaum noch in die virtuelle Population aufgenommen werden darf. Die Individuen der tatsächlichen Population werden jedoch in jedem Fall aufgenommen, egal ob sie diese Mindestgröße erreichen oder nicht. Ein Spezialfall ist ein Eintrag von null in dieser Variablen: In diesem Fall werden ausschließlich die Individuen der tatsächlichen Population, ohne irgendwelche Unterbäume, in die virtuelle Population aufgenommen.
- `AdjustedFitnessSum`: Gibt die Summe der adjustierten Fitnesswerte aller Individuen der virtuellen Population an.
- `ParentsAvailable`: Gibt an, ob die Elterngeneration dieser Population noch verfügbar ist, oder nicht.

6.3.2 Erzeugung und Vernichtung der Population

6.3.2.1 `Population()`

```
    /*-------------------------------------------------------
      Population::Population()

      Constructor: Creating an empty Population
 5    -------------------------------------------------------
    */
    Population::Population()
    {
            ActualPopSize = 0;
10          ActualPopArraySize = 0;
            Actual = NULL;
            VirtualPopSize = 0;
            VirtualPopArraySize = 0;
            Virtual = NULL;
15          ParentsAvailable = NO;
            AdjustedFitnessSum = 0;
            MatingList = NULL;
            MatingListArraySize = 0;
            MatingListNoOfEntries = 0;
20          SmallestSubTreeOfVirtualPop = 0;
    }
```

Der Leerkonstruktor definiert zunächst, daß diese Population keine Individuen beinhaltet (Zeilen 9-14) und keiner Elternpopulation zugeordnet ist (Zeile 15). Die Summe der

adjustierten Fitnesswerte wird mit null initialisiert (Zeile 16). Dann wird festgehalten, daß keine "mating-list" erzeugt wurde (Zeilen 17-19). Schließlich wird als Defaultwert für `SmallestSubTreeOfVirtualPop` null festgelegt (Zeile 20).

6.3.2.2 Population(Language*, int, int)

Dieser Konstruktor legt fest, daß diese Population der im ersten Parameter referenzierten Grammatik (`Language`) zugeordnet ist und erzeugt eine initiale Zufallspopulation aus sovielen Individuen, wie im zweiten Parameter angegeben wurden. Der dritte Parameter gibt die maximal zulässige Ableitungstiefe der Individuen in der tatsächlichen Population an. Bei der Zufallserzeugung wird auf die Gleichverteilung der Population keine Rücksicht genommen. Die in diesem Konstruktor erzeugte virtuelle Population enthält genau die Individuen der tatsächlichen Population, jedoch keine Unterbäume.

```
    /*-------------------------------------------------------
      Population::Population(Language* L, int PopSize, int MaxDepth)

      Constructor: Creating a new Population of language L
 5    with PopSize Trees, each with maximum derivation depth
      MaxDepth
      -------------------------------------------------------
    */
    Population::Population(Language* L, int PopSize, int MaxDepth)
10  {
            ActualPopSize = PopSize;
            ActualPopArraySize = PopSize;
            VirtualPopSize = 0;
            VirtualPopArraySize = 0;
15          Virtual = NULL;
            SmallestSubTreeOfVirtualPop = 0;

            Actual = new RootNode[PopSize](L, MaxDepth, this);

20          GenerateVirtualPop();
            ParentsAvailable = NO;
            AdjustedFitnessSum = 0;
            MatingList = new MatingListEntry[VirtualPopSize];
            MatingListArraySize = VirtualPopSize;
25          MatingListNoOfEntries = VirtualPopSize;
    }
```

In den Zeilen 11 und 12 wird zunächst die initiale Populationsgröße und die Größe des `Actual`-Arrays festgelegt. Sodann wird in den Zeilen 13-15 die virtuelle Population vorerst als nicht existent vermerkt. In Zeile 16 wird festgelegt, daß ausschließlich die Individuen der tatsächlichen Population, aber keine Unterbäume in die virtuelle Population aufgenommen werden sollen. In Zeile 18 wir dann die tatsächliche initiale Zufallspopulation unter Zuhilfenahme der RootNode()-Konstruktoren erzeugt. Die Funktion `GenerateVirtualPop()` in Zeile 20 erzeugt sodann eine virtuelle Population, die ausschließlich die Individuen der tatsächlichen Population, aber keine Unterbäume enthält. Zuletzt wird eine "mating-list" erzeugt, die genau die Größe der erzeugten virtuellen Population hat (Zeile 23-25).

6.3.2.3 GenerateVirtualPop()

Diese, im vorigen Konstruktor verwendetete Funktion, erzeugt eine virtuelle Population, die ausschließlich die Individuen der tatsächlichen Population, aber keine Unterbäume enthält.

```
    /*-----------------------------------------------------
      void Population::GenerateVirtualPop()

      Generating a virtual population (pointers to the trees
 5    of the real population) not including the subtrees
      -----------------------------------------------------
    */
    void Population::GenerateVirtualPop()
    {
10          int i;

            SmallestSubTreeOfVirtualPop = 0;

            if(VirtualPopArraySize<ActualPopSize)
15                  ReSizeVirtualPop(ActualPopSize,NO);
            else
                    VirtualPopSize = ActualPopSize;

            for(i=0;i<ActualPopSize;i++)
20                  Virtual[i] = Actual+i;
    }
```

In Zeile 12 wird zunächst durch Initialisierung der Variablen `SmallestSubTreeOfVirtualPop` festgelegt, daß ausschließlich die Individuen der tatsächlichen Population, aber keine Unterbäume in die virtuelle Population aufgenommen werden sollen. Sodann wird in Zeile 14 geprüft, ob die Größe des durch `Virtual` referenzierten Arrays ausreicht, um die zu erzeugende virtuelle Population aufzunehmen. Ist dies nicht der Fall, wird das Array unter Zuhilfenahme der Funktion `ReSizeVirtualPop(...)` vergrößert. Diese Funktion erwartet als ersten Parameter die gewünschte Größe der virtuellen Population. Der zweite Parameter gibt an, ob der Inhalt der bisherigen virtuellen Population in die neu zu erzeugende übernommen werden soll, oder nicht. Im speziellen Fall wird als erster Parameter die Größe der tatsächlichen Population angegeben, da die virtuelle Population genau die gleiche Größe haben soll, und als zweiter Parameter `NO`, da der Inhalt der bisherigen virtuellen Population nicht mehr von Belang ist. Hat die Prüfung in Zeile 14 jedoch ergeben, daß das bestehende Array zur Aufnahme der neuen virtuellen Population groß genug ist, dann wird lediglich die Größe der neuen virtuellen Population in der Variablen `VirtualPopSize` vermerkt (Zeile 17). Schließlich wird in der in Zeile 19 beginnenden Schleife die virtuelle Population mit Pointern auf die Wurzelknoten der tatsächlichen Population aufgefüllt (Zeile 20).

6.3.2.4 GenerateVirtualPopInclSubTrees(unsigned int)

Diese Funktion erzeugt eine virtuelle Population, in die zunächst, unabhängig von ihrer Ableitungstiefe, alle Mitglieder der tatsächlichen Population aufgenommen werden,

und zusätzlich alle Unterbäume, die die im Parameter angegebene Mindestableitungstiefe nicht unterschreiten. Ein Spezialfall ist die Angabe von null als Parameter: In diesem Fall werden ausschließlich die Individuen der tatsächlichen Population, ohne irgendwelche Unterbäume, in die virtuelle Population aufgenommen.

```
    /*-------------------------------------------------------
      void Population::GenerateVirtualPopInclSubTrees(unsigned int MinDepth)

      Generating a virtual population (pointers to the trees
 5    of the real population) including all subtrees larger
      than MinDepth.
      -------------------------------------------------------
    */
    void Population::GenerateVirtualPopInclSubTrees(unsigned int MinDepth)
10  {

            int i;

            VirtualPopSize = 0;
15          SmallestSubTreeOfVirtualPop = MinDepth;

            for(i=0;i<ActualPopSize;i++)
                    Actual[i].AddCompleteSubTreesTo(this,MinDepth);
    }
```

In den Zeilen 14 und 15 wird zunächst die Größe der virtuellen Population mit null festgelegt, und der Wert für den kleinsten Unterableitungsbaum, der noch in die Population mit aufgenommen werden soll, entsprechend gesetzt. In der in Zeile 17 beginnenden Schleife werden sodann alle Mitglieder der tatsächlichen Population durchgegangen, und in jedem Mitglied wird die Funktion `StartNode::AddCompleteSubTreesTo(...)` aufgerufen. Diese Funktion durchwandert rekursiv den gesamten Ableitungsbaum, und fügt alle Unterbäume, die die im zweiten Parameter angegebene Ableitungstiefe nicht unterschreiten, unter Verwendung der Funktion `Population::AddTreeToVirtualPop(...)` dem virtuellen Teil der im ersten Parameter referenzierten Population hinzu. Der Baum des Startknotens, in dem diese Funktion zuerst aufgerufen wurde, wird der virtuellen Population, unabhängig von seiner Größe, in jedem Fall hinzugefügt.

6.3.2.5 `AddTreeToVirtualPop(StartNode*)`

Diese Funktion fügt das im Parameter referenzierte Individuum der virtuellen Population am Ende hinzu.

```
    /*-------------------------------------------------------
      void Population::AddTreeToVirtualPop(StartNode* Tree)

      Adding a tree to the virtual population.
 5    -------------------------------------------------------
    */
    void Population::AddTreeToVirtualPop(StartNode* Tree)
    {
            if(Tree!=NULL)
```

```
10          {
                    if(VirtualPopSize>=VirtualPopArraySize)
                            ReSizeVirtualPop(VirtualPopSize+1,YES);
                    else
                            VirtualPopSize++;
15
                    Virtual[VirtualPopSize-1] = Tree;
            }
    }
```

In Zeile 9 wird zunächst geprüft, ob der Parameter überhaupt einen Pointer auf ein hinzuzufügendes Individuum beinhaltet, oder lediglich einen `NULL`-Pointer. Wird durch den Parameter ordnungsgemäß ein hinzuzufügendes Individuum referenziert, dann wird in Zeile 11 getestet, ob der Platz im bestehenden `Virtual`-Array bereits erschöpft ist. Sollte dies der Fall sein, dann wird das Array mit Hilfe der Funktion `ReSizeVirtualPop(int, int)` redimensioniert. Diese Funktion erwartet als ersten Parameter die gewünschte Größe der virtuellen Population. Der zweite Parameter gibt an, ob der Inhalt der bisherigen virtuellen Population in die neu zu erzeugende übernommen werden soll oder nicht. Im speziellen Fall wird als erster Parameter die Größe der bisherigen virtuellen Population plus eins, und als zweiter Parameter `YES` angegeben, da ansonsten alle bisherigen Einträge der virtuellen Population vernichtet würden. Ist der Platz im `Virtual`-Array noch nicht erschöpft, dann wird einfach der Inhalt der `VirtualPopSize`-Variablen um eins hinaufgezählt. In Zeile 16 schließlich wird der im Parameter angegebene Pointer der virtuellen Population hinzugefügt.

6.3.2.6 GenerateAlmostUniformActualPop(Language*, int, int)

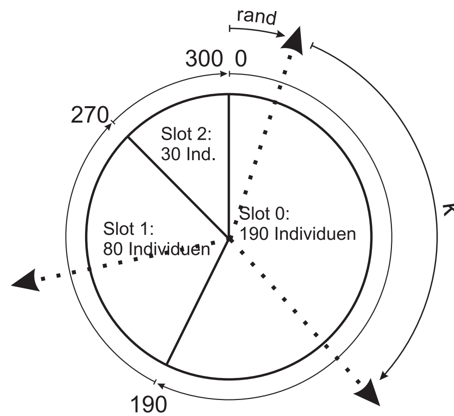
Diese Funktion löscht eine eventuell vorhandene tatsächliche und virtuelle Population, und erzeugt nach der in Kapitel 4.1.2 beschriebenen Heuristik eine neue, nach Größe der Individuen gleichverteilte tatsächliche Population. Der erste Parameter referenziert dabei die Grammatik, der zweite Parameter gibt die gewünschte Größe der tatsächlichen Population und der dritte Parameter die maximal zulässige Ableitungstiefe an. Die virtuelle Population wird in gleicher Größe wie die tatsächliche Population erstellt und enthält ausschließlich Pointer auf die Wurzelknoten der tatsächlichen Population, aber keine Pointer auf Unterbäume.

```
    /*-------------------------------------------------------
      Population::GenerateAlmostUniformActualPop(Language* L, int PopSize, int Depth)

      Generating an more or less uniform initital population
 5    using the heuristic due to

      Andreas Geyer Schulz
      Fuzzy Rule-Based Expert Systems and Genetic Machine Learning
      (Studies in Fuzziness, Vol. 3),
10    Physica-Verlag, Wien 1995.
      -------------------------------------------------------
    */
    int Population::GenerateAlmostUniformActualPop(Language* L, int PopSize,  int Depth)
    {
15          static Uniran SpinningWheel(RANDPOPINIT);
```

```
            SuperFloat* RouletteSlot;
            SuperFloat* NoOfAvailableIndividuals;
            SuperFloat SelectedSlot;
            SuperFloat Steps;
 20         int* TargetNumberOfIndividuals;
            RootNode* TempTree;
            int i;
            int j;
            SuperFloat k;
 25
            if(L->ArbCard==NULL)
            {
                    cout << "Computing search space size up to n = " << Depth;
                    cout << "..." << endl;
 30                 L->ComputeArbSearchSpace(Depth);
                    cout << "done." << endl;
            }
            else
                    if(L->ArbCardSize<Depth)
 35                 {
                            cout << "Computing search space size up to n = " << Depth;
                            L->ComputeArbSearchSpace(Depth);
                            cout << "done." << endl;
                    }
 40
            cout << "Generating population..." << endl;

            if(Actual!=NULL)
                    delete [] Actual;
 45
            ActualPopSize = PopSize;
            ActualPopArraySize = PopSize;
            Actual = new RootNode[PopSize];

 50
            // Now the roulette-wheel construction...

            RouletteSlot = new SuperFloat[Depth+1];
            NoOfAvailableIndividuals = new SuperFloat[Depth+1];
 55         TargetNumberOfIndividuals = new int[Depth+1];

            k = 0.0;
            for(i=0;i<=Depth;i++)
            {
 60                 RouletteSlot[i] = L->ArbCard[i] + k;
                    k = RouletteSlot[i];
                    NoOfAvailableIndividuals[i] = L->ArbCard[i];
                    TargetNumberOfIndividuals[i] = 0;
            }
 65

            // stochastic universal sampling

            Steps = 1.0/PopSize;
 70         k = k*Steps;
            SelectedSlot = SpinningWheel.rand();
            SelectedSlot = k * SelectedSlot;
            for(i=0;i<PopSize;i++)
            {
75                  j = Depth-1;
                    while(j>=0 && RouletteSlot[j]>SelectedSlot)
                            j--;

                    if(j<=0 && RouletteSlot[0]>SelectedSlot)
 80                         TargetNumberOfIndividuals[0]++;
                    else
                            TargetNumberOfIndividuals[j+1]++;
```

```
                SelectedSlot = SelectedSlot + k;
 85         }

            // Production of Individuals

            i = 0;
 90         while(i<PopSize)
            {
                TempTree = new RootNode(L, Depth, this);
                if(TargetNumberOfIndividuals[TempTree->DerivationDepth]>0)
                {
 95                 if(NoOfAvailableIndividuals[TempTree->DerivationDepth].Mant>0)
                    {
                        j=0;
                        while(j<i && !(Actual[j]==*TempTree))
                            j++;
100
                        if(Actual[j]==*TempTree)
                            delete TempTree;
                        else
                        {
105                         Actual[i] = *TempTree;
                            i++;
                            TargetNumberOfIndividuals[TempTree->
                                                      DerivationDepth]--;

110                         NoOfAvailableIndividuals[TempTree->
                                                     DerivationDepth]--;
                        }
                    }
                    else
115                 {
                        Actual[i] = *TempTree;
                        i++;
                        TargetNumberOfIndividuals[TempTree->DerivationDepth]--;
                    }
120             }
                else
                    delete TempTree;
            }

125         delete [] RouletteSlot;
            delete [] NoOfAvailableIndividuals;
            delete [] TargetNumberOfIndividuals;

            GenerateVirtualPop();
130         ParentsAvailable = NO;
            AdjustedFitnessSum = 0;

            MatingList = new MatingListEntry[VirtualPopSize];
            MatingListArraySize = VirtualPopSize;
135         MatingListNoOfEntries = VirtualPopSize;

            cout << "done..." << endl;

            return(TRUE);
140     }
```

Der Algorithmus der obigen Funktion ist aus [Geyer-Schulz, 1994] entnommen. Die Funktion läßt sich in folgende Abschnitte gliedern:



NoOfAvailableIndividuals:

0	1	2
190	80	30

RouletteSlot:

0	1	2
190	270	300

NoOfSelectedIndividuals:

0	1	2
2	1	0

Abbildung 29: Auswahl von Individuen mittels stochastic universal sampling

- **Initialisierungen und Suchraumberechnung**

  In Zeile 21 wird zunächst geprüft, ob in der angegebenen Grammatik bereits eine Suchraumtabelle zur Verfügung steht. Ist dies nicht der Fall, dann wird diese unter Ausgabe einer Meldung (Zeilen 28 und 29) und unter Zuhilfenahme der Funktion `Language::ComputeArbSearchSpaceSize(int)` bis zur benötigten Ableitungstiefe berechnet (Zeile 30). Anschließend wird die Meldung "done." ausgegeben (Zeile 31). Gibt es bereits eine Suchraumtabelle, so wird in Zeile 29 geprüft, ob sie entsprechend tief errechnet wurde. Ist dies nicht der Fall, dann wird in Zeile 36 eine entsprechende Meldung ausgegeben und die Tabelle unter Zuhilfenahme der Funktion `Language::ComputeArbSearchSpaceSize(int)` bis zur benötigten Ableitungstiefe berechnet (Zeile 37). Die Berechnung wird wieder mit der Meldung "done." ab geschlossen (Zeile 38). In Zeile 41 wird sodann mit der Meldung "Generating population..." der Start der eigentlichen Heuristik angekündigt. In Zeile 43 wird zuvor aber noch getestet, ob bereits eine alte (tatsächliche) Population vorhanden ist. Sollte dies der Fall sein, dann wird sie in Zeile 44 vernichtet. In den Zeilen 46-48 wird die Populations- und Arraygröße für die neu zu erstellende Population entsprechend eingestellt und das Array erzeugt.

- **Erzeugung des Roulett-Rades**

  Im nächsten Schritt wird ein Abbild des Roulettrades mit den verschieden großen Sektoren im durch `RouletteSlot` referenzierten Array erzeugt. Das Array selber wird in Zeile 53 erzeugt. In Zeile 54 und 55 werden zudem zwei Arrays erzeugt, die für eine bestimmte Ableitungstiefe (Index des Arrays) die Zahl der zur Verfügung stehenden Individuen, entnommen aus der Suchraumtabelle, sowie die Zielanzahl von Individuen für eine bestimmte Ableitungstiefe aufnehmen sollen. In der in Zeile 58 beginnenden Schleife werden die Arrays aufgefüllt. Als Zielanzahl wird vorerst stets null gespeichert (Zeile 63) Die Anzahl der verfügbaren (unterschiedlichen) Individuen für eine bestimmte Ableitungstiefe wird aus der Suchraumtabelle kopiert, und das durch `RouletteSlot` referenzierte Array nimmt für jede Ableitungstiefe die kumulierten Werte der Suchraumtabelle auf. Dies bedeutet, daß der einer bestimmten Ableitungstiefe i im `RouletteSlot`-Array zugeordnete Wert `RouletteSlot[i]` die obere Begrenzung des i-ten Sektors des Rouletterades bildet. Der Wert in `RouletteSlot[i-1]` bildet die untere Begrenzung des i-ten Sektors. Die untere Begrenzung für `RouletteSlot[0]` ist null. Nach Durchlaufen der Schleife ist das durch `RouletteSlot` referenzierte Array sowie das Array, welches die Anzahl der für jede Ableitungstiefe zur Verfügung stehenden unterschiedlichen Individuen beinhaltet, richtig zugewiesen. Abbildung 29 veranschaulicht die Umsetzung der Sektoren des Roulett-Rades in die durch `RouletteSlot` und durch `NoOfAvailableIndividuals` referenzierten Arrays.

- **Berechnung der Zielanzahl von Individuen für jede Größe**

  Die Bestimmung der Zielanzahl erfolgt mit Hilfe des "stochastic universal sampling" nach [Baker, 1987]. Basis ist das - zuvor erstellte - durch `RouletteSlots` referenzierte Array. In den Zeilen 69 und 70 wird zunächst der ganze "Umfang" des Roulettrades durch die Anzahl der zu erzeugenden Individuen dividiert. Dadurch erhält man den Abstand k zweier benachbarter "Speichen" des "Auswahlrades" (siehe Abbildung 29). In den Zeilen 71 und 72 wird im Bereich zwischen Null und "Speichenabstand" k ein zufälliger Punkt $rand$ gesucht. In der in Zeile 73 beginnenden Schleife werden von diesem zufällig gewählten Punkt an durch Hinzuaddieren des "Speichenabstandes" k die entsprechenden Individuen bestimmt. In welchen Sektor des Rouletterades die momentan behandelte "Speiche" zu liegen kommt, wird in den Zeilen 75-84 ausgewertet. Beginnend mit der obersten Grenze des letzten Sektors wird das `RouletteSlot`-Array in der in Zeile 76 beginnenden Schleife solange durchlaufen, bis eine obere Sektorgrenze gefunden wurde, die kleiner ist als der gewählte Punkt, oder bis das ganze Roulett-Rad durchlaufen wurde. In der Zeile 82 wird die Zielanzahl des entsprechenden Sektors um eins erhöht. Die Abfrage in Zeile 79 ist nötig, da die untere Begrenzung des nullten Sektors (die gleich null ist) nicht im durch `RouletteSlots` referenzierten Array abgebildet wird. Abbildung 29 veranschaulicht dies.

- **Erzeugung der Individuen**

  Im nächsten Schritt wird die in Zeile 90 beginnende Schleife so oft durchlaufen, bis die entsprechende Anzahl von Individuen erzeugt wurde. In Zeile 92 wird ein, vorerst temporäres, Zufallsindividuum erzeugt. In Zeile 93 wird sodann verglichen, ob die Zielanzahl für Individuen der Ableitungstiefe des soeben erzeugten noch größer null ist. Sollte dies nicht der Fall sein, so wird das temporäre Individuum vernichtet (Zeile 122), und es konnte in diesem Schleifendurchlauf der Population kein Individuum hinzugefügt werden. Ist die Zielanzahl jedoch ungleich null, dann wird im nächsten Schritt in Zeile 95 getestet, ob es noch Individuen dieser Ableitungstiefe geben kann, die sich von den bereits in die Population aufgenommenen unterscheiden. Ist dies der Fall, dann wird in der in Zeile 98 beginnenden Schleife getestet, ob es in der Population bereits ein identisches Individuum gibt, oder nicht. Gibt es bereits ein identisches Individuum (Zeile 101), dann wird das temporäre Individuum vernichtet (Zeile 102), und es konnte in diesem Schleifendurchlauf der Population kein Individuum hinzugefügt werden. Andernfalls wird das Individuum in die Population aufgenommen (Zeile 105), die erfolgreiche Erzeugung eines Individuums im Zähler i vermerkt (Zeile 106), und die Zielanzahl und die Anzahl der zur Verfügung stehenden Individuen dieser Ableitungstiefe um eins vermindert (Zeilen 107 - 110). Stellt sich im Test in der Zeile 95 jedoch heraus, daß es gar keine Individuen dieser Größe mehr geben kann, die sich von den bereits in die Population aufgenommenen unterscheiden, dann wird das Individuum der Population auf jeden Fall hinzugefügt (Zeile 116), die erfolgreiche Erzeugung eines Individuums im Zähler i vermerkt (Zeile 117) und die Zielanzahl für Individuen dieser Größe um eins vermindert (Zeile 118).

- **Erzeugung der virtuellen Population und Abschluß**

  Nachdem alle Individuen der tatsächlichen Population erzeugt wurden, werden zunächst in den Zeilen 125-127 die erzeugten temporären Arrays vernichtet. Danach wird in Zeile 129, unter Zuhilfenahme der Funktion `GenerateVirtualPop()` die virtuelle Population aufgefüllt. In Zeile 130 wird festgehalten, daß es zu dieser Population keine Elternpopulation gibt. In den Zeilen 131-133 wird schließlich eine "mating-list" vorbereitet, die der Größe der virtuellen Population entspricht. In Zeile 137 wird durch Ausgabe der Meldung "done..." das Ende der Funktion angezeigt, und die Funktion wird dann, unter Rückgabe von `TRUE`, abgeschlossen (Zeile 139).

6.3.2.7 Der Destruktor von `Population()`

Der Destruktor sorgt dafür, daß der durch die tatsächliche und die virtuelle Population belegte Speicherplatz, sowie der durch die Mating-List belegte Platz, wieder freigegeben wird.

```
    /*------------------------------------------------------
      Population::~Population()

      Destructor: Deleting the actual and the virtual
 5    population and the mating-list.
      ------------------------------------------------------
    */
    Population::~Population()
    {
10          if(ActualPopArraySize>0)
                    delete [] Actual;

            if(VirtualPopArraySize>0)
                    delete [] Virtual;
15
            if(MatingListArraySize>0)
                    delete [] MatingList;
    }
```

In Zeile 10 wird geprüft, ob es ein zu löschendes `Actual`-Array gibt. Ist dies der Fall, so wird dieses in Zeile 11 vernichtet. Genauso wird in den Zeilen 3 und 4 mit dem `Virtual`-Array, und in den Zeilen 16 und 17 mit dem MatingList-Array verfahren.

6.3.3 Fitnessberechnungen

6.3.3.1 `ApplyRawFitness()`

Dieses Individuum errechnet und speichert, unter Zuhilfenahme der vom Anwender zu implementierenden `TreeNode::ObjFuncVal()`-Methode, den Rohfitnesswert jedes in der virtuellen Population vermerkten Individuums.

```
    /*------------------------------------------------------
      void Population::ApplyRawFitness()

      Computing the raw fitness value (objective function value)
 5    for all trees in the virtual population.
      ------------------------------------------------------
    */
    void Population::ApplyRawFitness()
    {
10          int i;

            for(i=0;i<VirtualPopSize;i++)
                    Virtual[i]->RawFitness = Virtual[i]->ObjFuncVal();
    }
```

In der in Zeile 12 beginnenden Schleife werden alle in der virtuellen Population referenzierten Individuen durchlaufen, und der mit Hilfe der `TreeNode::ObjFuncVal()`-Methode errechnete Fitnesswert gespeichert (Zeile 12).

6.3.3.2 `ComputeBestAndWorstIndividual(int)`

Diese Funktion bestimmt und speichert das beste und schlechteste in der virtuellen Population vermerkte Individuum. Als Parameter wird das Optimierungsziel `MAXIMIZE` oder `MINIMIZE` erwartet. Die Rohfitnesswerte müssen zuvor bereits berechnet worden sein.

```
    /*-------------------------------------------------------
      void Population::ComputeBestAndWorstIndividual(int goal)

      Computing the best and worst individual of this
 5    population depending on the goal (MAXIMIZE or MINIMIZE)
      -------------------------------------------------------
    */
    void Population::ComputeBestAndWorstIndividual(int goal)
    {
10          int i;

            if(VirtualPopSize!=0)
            {
                    BestIndividual = Virtual[0];
15                  WorstIndividual = BestIndividual;

                    if(goal==MINIMIZE)
                    {
                            for(i=0;i<VirtualPopSize;i++)
20                          {
                                    if(Virtual[i]->RawFitness<BestIndividual->RawFitness)
                                            BestIndividual = Virtual[i];

                                    if(Virtual[i]->RawFitness>WorstIndividual->RawFitness)
25                                          WorstIndividual = Virtual[i];
                            }
                    }

                    else
30
                    {
                            for(i=0;i<VirtualPopSize;i++)
                            {
                                    if(Virtual[i]->RawFitness>BestIndividual->RawFitness)
35                                          BestIndividual = Virtual[i];

                                    if(Virtual[i]->RawFitness<WorstIndividual->RawFitness)
                                            WorstIndividual = Virtual[i];
                            }
40                  }
            }
    }
```

In Zeile 12 wird zunächst geprüft, ob es überhaupt eine virtuelle Population gibt. Ist dies der Fall, dann wird vorerst angenommen, daß das beste und das schlechteste Individuum gleich dem ersten Individuum der virtuellen Population wären (Zeile 14 und 15). Sodann wird in Zeile 17 geprüft, ob das im Parameter definierte Optimierungsziel eine Minimierungsaufgabe oder eine Maximierungsaufgabe festlegt. Handelt es sich um eine Minimierungsaufgabe, dann wird in der in Zeile 19 beginnenden Schleife die virtuelle Population durchlaufen. Jedes Individuum wird dahingehend getestet, ob es einen kleineren Rohfitnesswert hat, als das beste bisher gefundene Individuum (Zeile 21), und wird gegebenenfalls zum neuen besten Individuum erklärt (Zeile 22). Ebenso wird getestet, ob es einen größeren Rohfitnesswert aufweist, als das schlechteste bisher gefundene (Zeile 24) und gegebenenfalls zum neuen schlechtesten Individuum erklärt (Zeile 25). Handelt es sich jedoch um eine Maximierungsaufgabe, dann wird stattdessen die in der Zeile 32 beginnenden Schleife durchlaufen. Jedes Individuum wird dahingehend geprüft, ob es einen größeren Rohfitnesswert hat, als das beste bisher gefundene Individuum (Zeile 34), und wird gegebebenfalls zum neuen besten Individuum erklärt

(Zeile 35). Ebenso wird getestet, ob es einen kleineren Rohfitnesswert aufweist, als das schlechteste bisher gefundene (Zeile 37) und gegebenenfalls zum neuen schlechtesten Individuum erklärt (Zeile 38).

6.3.3.3 `StandardizeWith(float, int)`

Diese Funktion errechnet und speichert für jedes in der virtuellen Population vermerkte Individuum die standardisierte Fitness. Der erste Parameter gibt dabei den Rohfitnesswert an, der gerade eine standardisierte Fitness von null ergeben soll. Der zweite Parameter definiert das Optimierungsziel: `MAXIMIZE` für Maximierungsaufgaben, `MINIMIZE` für Minimierungsaufgaben. Die standardisierte Fitness muß zuvor schon berechnet worden sein.

```
     /*-------------------------------------------------------
       Population::void StandardizeWith(float BestRawFitness, int goal)

       Computing the StandardizedFitness for each individual of
 5     the virtual population, considering the goal (MINIMIZE
       or MAXIMIZE).

       Better individuals get a lower fitness value and the
       lowest possible standardized fitness value of zero is
10     assigned to  the individual with RawFitness ==
       BestRawFitness.
       -------------------------------------------------------
     */
     void Population::StandardizeWith(float BestRawFitness, int goal)
15   {
             int i;

             if(goal==MINIMIZE)
             {
20                   for(i=0;i<VirtualPopSize;i++)
                             Virtual[i]->StandardizedFitness =
                             Virtual[i]->RawFitness - BestRawFitness;
             }
             else
25           {
                     for(i=0;i<VirtualPopSize;i++)
                             Virtual[i]->StandardizedFitness =
                             BestRawFitness - Virtual[i]->RawFitness;
             }
30    }
```

In Zeile 18 wird zunächst geprüft, ob das im zweiten Parameter angegebene Optimierungsziel eine Minimierungsaufgabe oder eine Maximierungsaufgabe festlegt. Handelt es sich um eine Minimierungsaufgabe, dann wird in der in Zeile 20 beginnenden Schleife die virtuelle Population durchlaufen, und für jedes in der virtuellen Population vermerkte Individuum die standardisierte Fitness entprechend einer Minimierungsaufgabe berechnet und gespeichert (Zeile 21 und 22). Handelt es sich jedoch um eine Maximierungsaufgabe, dann wird die in Zeile 26 beginnende Schleife durchlaufen, und für jedes in der virtuellen Population vermerkte Individuum die standardisierte Fitness entsprechend einer Maximierungsaufgabe berechnet und gespeichert (Zeile 21 und 22).

6.3.3.4 InvertFitnessValues()

Diese Funktion errechnet für jedes in der virtuellen Population vermerkte Individuum an Hand der standardisierten Fitness einen adjustierten Fitnesswert nach der in Kapitel 4.2.2 beschriebenen Methode. Außerdem wird die Summe der adjustierten Fitnesswerte in der Population gespeichert.

```
    /*-----------------------------------------------------
      void Population::InvertFitnessValues()

      Computing the AdjustedFitness for every tree in the
 5    virtual population in a way that better trees achieve
      a higher adjusted fitness value. The worst tree in the
      population gets a fitness value of zero.
      The sum over the adjusted fitness values is stored in
      Population::AdjustedFitnessSum
10
      Alternative to Farm::AdjustFitnessValues()

      StandardizedFitness must have been already computed.
      -----------------------------------------------------
15  */
    void Population::InvertFitnessValues()
    {
            int i;

20          AdjustedFitnessSum = 0.0;
            for(i=0;i<VirtualPopSize;i++)
            {
                    Virtual[i]->AdjustedFitness =
                    WorstIndividual->StandardizedFitness -
25                  Virtual[i]->StandardizedFitness;

                    AdjustedFitnessSum+=
                    Virtual[i]->AdjustedFitness;
            }
30  }
```

In Zeile 20 wird zunächst der Wert für die Summe der adjustierten Fitnesswerte mit null initialisiert. Danach wird in der in Zeile 21 beginnenden Schleife die virtuelle Population durchlaufen. Jedes in der virtuellen Population vermerkte Individuum bekommt einen standardisierten Fitnesswert zugewiesen (Zeilen 23-25), und die Summe der adjustierten Fitnesswerte wird entsprechend erhöht (Zeilen 27 und 28).

6.3.3.5 NormalizeFitnessValues()

Diese Funktion errechnet für jedes in der virtuellen Population vermerkte Individuum an Hand der adjustierten Fitness eine normalisierte Fitness, in der Art, daß die Summe der normalisierten Fitnesswerte genau eins ergibt. Die adjustierten Fitnesswerte müssen zuvor bereits berechnet worden sein. Haben alle in der virtuellen Population vermerkten Individuen einen adjustierten Fitnesswert von null, dann erhält jedes Individuum einen standardisierten Fitnesswert der gleich dem Kehrwert der Anzahl der Individuen ist.

```
     /*-----------------------------------------------------
       void Population::NormalizeFitnessValues()

       Computing the NormalizedFitness for every tree in the
 5     virtual population, in a way that the sum over all
       fitness values equals 1.

       If the AdjustedFitnessSum is zero,
       every individual receives a NormalizedFitness value
10     of 1/PopSize.

       AdjustedFitness and AdjustedFitnessSum must have been
       already computed.
       ----------------------------------------------------
15   */

     void Population::NormalizeFitnessValues()
     {
             int i;
20
             if(AdjustedFitnessSum!=0)
                     for(i=0;i<VirtualPopSize;i++)
                             Virtual[i]->NormalizedFitness =
                             Virtual[i]->AdjustedFitness /
25                           AdjustedFitnessSum;
             else
                     for(i=0;i<VirtualPopSize;i++)
                             Virtual[i]->NormalizedFitness =
                             1.0/VirtualPopSize;
30   }
```

In Zeile 21 wird zunächst geprüft, ob die Summe der adjustierten Fitnesswerte ungleich null ist (Normalfall). Trifft dies zu, dann wird in der in Zeile 22 beginnenden Schleife die virtuelle Population durchlaufen. Jedes in der virtuellen Population vermerkte Individuum erhält einen normalisierten Fitnesswert, der sich durch Division der adjustierten Fitness durch die Summe der adjustierten Fitnesswerte berechnet, zugewiesen (Zeilen 23-25). Stellt sich jedoch heraus, daß die Summe der adjustierten Fitnesswerte genau null beträgt, dann wird in der in Zeile 27 beginnenden Schleife die virtuelle Population durchlaufen, und jedes in der Population referenzierte Individuum erhält als normalisierten Fitnesswert den Kehrwert der Populationsgröße der virtuellen Population zugewiesen (Zeilen 28 und 29).

6.3.4 Genetische Operatoren

6.3.4.1 `PropSelect()`

Diese Funktion errechnet für jedes in der virtuellen Population vermerkte Individuum an Hand der normalisierten Fitness eine target sampling rate nach der in Kapitel 4.3.1 gezeigten Methode der proportionalen Selektion. Die normalisierte Fitness muß zuvor bereits berechnet worden sein.

```
    /*------------------------------------------------------
      void Population::PropSelect()

      Computing a target sampling rate for every tree
 5    of the virtual population in proportion to the stored
      normalized fitness values.

      Alternative to LinearRankSelect()

10    NormalizedFitness must have been already computed.
      ------------------------------------------------------
    */
    void Population::PropSelect()
    {
15          int i;
            float AverageFitness;

            AverageFitness = 1.0 / VirtualPopSize;

20          for(i=0;i<VirtualPopSize;i++)
                    Virtual[i]->TargSamplRate =
                    Virtual[i]->NormalizedFitness / AverageFitness;
    }
```

In Zeile 18 wird zunächst die durchschnittliche normalisierte Fitness errechnet, die, da die Summe aller standardisierten Fitnesswerte genau eins betragen muß, sich als Kehrwert der Größe der virtuellen Population errechnet. In der in Zeile 20 beginnenden Schleife werden alle in der virtuellen Population vermerkten Individuen durchlaufen. Jedes Individuum erhält eine Auswahlwahrscheinlichkeit (*target sampling rate* - tsr) zugewiesen, die sich als normalisierte Fitness des Individuums dividiert durch die durchschnittliche normalisierte Fitness errechnet.

6.3.4.2 LinearRankSelect(float)

Diese Funktion errechnet nach der in Kapitel 4.3.2 vorgestellten Methode eine Auswahlwahrscheinlickeit (*target sampling rate*) für alle in der virtuellen Population vermerkten Individuen. Als Parameter muß die gewünschte target sampling rate a_{min} für das schlechteste Individuum angegeben werden. Der zulässige Wertebereich beträgt $0 \leq a_{min} \leq 1$, jedoch werden Parameterangaben außerhalb des Wertebereiches automatisch auf die jeweilige Grenze des Wertebereichs umgerechnet.

```
    /*------------------------------------------------------
      void Population::LinearRankSelect(float amin)

      Calculating a target sampling rate for every tree
 5    of the virtual population according to its rank.

      amin gives the target sampling rate for the
      worst individual in the virtual population and
      can be used for regulating the selection pressure.
10
      0 <= amin <= 1

      amin = 0 ... high selection pressure
      amin = 1 ... no meaningful selection at all
```

```
15
      Alternative to PropSelect()
      -----------------------------------------------------
    */
    void Population::LinearRankSelect(float amin)
20  {
            int i;
            int j;
            int Worst;
            float amax;
25          StartNode* Temp;
            StartNode** Ranked;

            if(amin<0.0)
                    amin = 0.0;
30
            if(amin>1.0)
                    amin = 1.0;

            amax = 2.0 - amin;
35          Ranked = new StartNode*[VirtualPopSize];

            // ranking the individuals...

            for(i=0;i<VirtualPopSize;i++)
40                  Ranked[i] = Virtual[i];

            for(i=0;i<VirtualPopSize-1;i++)
            {
                    Worst = i;
45
                    for(j=i+1;j<VirtualPopSize;j++)
                            if(Ranked[j]->NormalizedFitness<
                            Ranked[Worst]->NormalizedFitness)
                                    Worst = j;
50
                    Temp = Ranked[i];
                    Ranked[i] = Ranked[Worst];
                    Ranked[Worst] = Temp;
            }
55
            // .. and computing the tsr according to the rank

            if(VirtualPopSize>1)
                    for(i=0;i<VirtualPopSize;i++)
60                          Ranked[i]->TargSamplRate =
                            amin + (amax - amin)*i/(VirtualPopSize-1);
            else
                    if(VirtualPopSize>0)
                            Ranked[0]->TargSamplRate = 1.0;
65
            delete [] Ranked;
    }
```

Zunächst wird in Zeile 28 geprüft, ob der Wert des Parameters die zulässige Untergrenze von null unterschreitet. In diesem Fall wird er auf null korrigiert (Zeile 29). In Zeile 31 wird danach getestet, ob der Parameter die zulässige Obergrenze von eins überschreitet. Ist dies der Fall, wird er auf eins korrigiert (Zeile 32). Anschließend wird in Zeile 34 die target sampling rate a_{max}, die dem besten Individuum zuzurechnen ist, berechnet. In Zeile 35 wird ein temporäres Array von Pointern auf Individuen (`StartNode`-Objekten) erzeugt, dessen Inhalt in Folge nach der normalisierten Fitness dieser Individuen sortiert wird. Die Sortierung erfolgt in der in Zeile 42 beginnenden Schleife nach einem

einfachen *in-situ* Verfahren: In der in Zeile 46 beginnenden Schleife wird aus allen bislang nicht sortierten Individuen das schlechtesete herausgesucht (Zeilen 47-49), und nach Beenden der Schleife der Pointer auf das gefundene schlechteste Individuum mit dem gerade aktuellen vertauscht (Zeilen 51-53). Dies wird solange gemacht, bis das ganze Array sortiert ist. In Zeile 58 wird schließlich getestet, ob die Größe der virtuellen Population eins übersteigt (Normalfall). Ist dies nämlich nicht der Fall, dann kann die Formel zur Berechnung der Auswahlwahrscheinlichkeit (*target sampling rate*) in den Zeilen 60 und 61 nicht angewendet werden, da eine Division durch null die Folge wäre. In diesem Fall wird die target sampling rate des einzigen Individuums der virtuellen Population (so vorhanden) mit eins festgesetzt (Zeilen 63 und 64). Ansonsten wird die target fitness rate aus dem jeweiligen Rang errechnet (Zeilen 60 und 61). In Zeile 66 schließlich wird nach erfolgter Berechnung das temporäre Array wieder vernichtet.

6.3.4.3 `StochSamplWithRepl(int)`

Diese Funktion wählt nach der in Kapitel 4.4.1 beschriebenen "Stochastic Sampling with Replacement"-Methode aus allen in der virtuellen Population vermerkten Individuen soviele aus, wie im Parameter angegeben werden. Die `Parent1`-Teile der Mating-List werden mit Pointern auf die ausgewählten Individuen ausgefüllt.

```
    /*-------------------------------------------------------
      void Population::StochSamplWithRepl(int n)

      Fills the Partner1 entries of the MatingList.
 5
      Selection method is the "roulette-wheel-method".
      The size of the roulette-wheel-slots depends on
      the stored target sampling rate (TargSamplRate),
      which must have been calculated before.
10    -------------------------------------------------------
    */
    void Population::StochSamplWithRepl(int n)
    {

15          int i;
            int j;
            float* RouletteSlot;
            float k;
            float SelectedSlot;
20          static Uniran SpinningWheel(RANDSAMPLSEED);

            if(n<1)
            {
                    n = VirtualPopSize;
25                  cerr << "Error! Stochastic sampling with replacement called with n=" << n;
                    cerr << "." << endl << "n will be ";
                    cerr << VirtualPopSize << " (current ";
                    cerr << "population size) instead." << endl;
            }
30
            if(n>MatingListArraySize)
                    ReSizeMatingList(n);
            else
                    MatingListNoOfEntries = n;
35
            RouletteSlot = new float[VirtualPopSize];

            k = 0.0;
```

```
            for(i=0;i<VirtualPopSize;i++)
40          {
                    RouletteSlot[i] =
                    Virtual[i]->TargSamplRate + k;
                    k = RouletteSlot[i];
            }
45
            for(i=0;i<n;i++)
            {

                    SelectedSlot = SpinningWheel.rand()*k;
50
                    j = VirtualPopSize-2;
                    while(j>=0 && RouletteSlot[j]>SelectedSlot)
                            j--;

55                  if(j<=0 && RouletteSlot[0]>SelectedSlot)
                            MatingList[i].Partner1 = Virtual[0];
                    else
                            MatingList[i].Partner1 = Virtual[j+1];
            }
60
            delete [] RouletteSlot;

    }
```

In Zeile 22 wird zunächst getestet, ob der Wert des Parameters, der die Anzahl der auszuwählenden Individuen angibt, kleiner als eins ist. Sollte dies der Fall sein, dann wird als Parameterwert die Größe der virtuellen Population angenommen (Zeile 24) und eine entsprechende Meldung ausgegeben (Zeilen 25-28). In Zeile 31 wird sodann geprüft, ob mehr Individuen auszuwählen sind, als im momentanen Array zur Aufnahme der Mating-List Platz hätten. Ist dies der Fall, dann wird die Mating-List unter Zuhilfenahme der Funktion `ReSizeMatingList(int)` redimensioniert (Zeile 32). Andernfalls wird einfach der Zähler, der die Anzahl der Einträge in der Mating-List aufnimmt, entsprechend eingestellt (Zeile 34). In Zeile 36 wird dann ein Array erzeugt, dessen Inhalt in Folge das Roulettrad der "stochastic sampling with replacement"-Methode abbilden soll. Wie diese Abbildung erfolgt, wurde bereits im Kapitel 6.3.2.6 auf Seite 162 gezeigt. In der in Zeile 39 beginnenden Schleife wird das durch `RouletteSlots` referenzierte Array entsprechend mit den kumulierten target sampling rates aufgefüllt (Zeilen 41-43). In der in Zeile 46 beginnenden Schleife erfolgt dann die eigentliche Auswahl von n Individuen: In Zeile 49 wird zunächst eine Zufallszahl im Bereich zwischen 0 und der oberen Grenze des letzten Roulett-Segments erzeugt. In der in Zeile 52 beginnenden Schleife wird sodann getestet, in welches Segment die erzeugte Zufallszahl fällt. Beginnend von der obersten Grenze des letzten Sektors wird das `RouletteSlot`-Array in der in Zeile 52 beginnenden Schleife solange durchlaufen, bis eine obere Sektorgrenze gefunden wurde, die kleiner ist als der gewählte Punkt, oder bis das ganze Roulettrad durchlaufen wurde. In Zeilen 58 wird das so gewählte Individuum als nächster `Partner1`-Eintrag der Mating-List hinzugefügt. Die Abfrage in Zeile 55 ist nötig, da die untere Begrenzung des nullten Sektors (die gleich null ist) nicht im durch `RouletteSlots` referenzierten Array abgebildet wird. Die Zusammenhänge zwischen dem gedachten Roulettrad und dem `RouletteSlots`-Array können auch der Abbildung 29 entnommen werden. Nach erfolgter Selektion wird schließlich das erzeugte temporäre Array in Zeile 61 wieder vernichtet.

6.3.4.4 StochUnivSampl()

Diese Funktion wählt nach der in Kapitel 4.4.2 beschriebenen "Stochastic Universal Sampling"-Methode aus allen in der virtuellen Population vermerkten Individuen soviele aus, wie im Parameter angegeben werden. Die `Parent1`-Teile der Mating-List werden mit Pointern auf die ausgewählten Individuen ausgefüllt.

```
    /*-------------------------------------------------------
      void Population::StochUnivSampl(int n)

      Fills the Partner1 entries of the MatingList
 5    with treepointers
      refering to randomly selected trees with the
      "stochastic universal sampling methode" based
      on the target sampling reate (TargSamplRate).

10    The array is created at runtime. If n exceedes
      the largest array so far (ArraySize), the old
      one is deleted, and a new one created.
      -------------------------------------------------------
    */
15  void Population::StochUnivSampl(int n)
    {

            int i;
            int j;
20          float* RouletteSlot;
            float k;
            float SelectedSlot;
            static Uniran SpinningWheel(STOCHUNIVSAMPLSEED);

25          if(n<1)
            {
                    n = VirtualPopSize;
                    cerr << "Error! Stochastic universal sampling with replacement"
                    cerr << "called with n=" << n;
30                  cerr << "." << endl << "n will be ";
                    cerr << VirtualPopSize << " (current ";
                    cerr << "population size) instead." << endl;
            }

35          if(n>MatingListArraySize)
                    ReSizeMatingList(n);
            else
                    MatingListNoOfEntries = n;

40          RouletteSlot = new float[VirtualPopSize];

            k = 0.0;
            for(i=0;i<VirtualPopSize;i++)
            {
45                  RouletteSlot[i] =
                    Virtual[i]->TargSamplRate + k;
                    k = RouletteSlot[i];
            }

50          k/= n;
            SelectedSlot = SpinningWheel.rand()*k;

            for(i=0;i<n;i++)
            {
55                  j = VirtualPopSize-2;
                    while(j>=0 && RouletteSlot[j]>SelectedSlot)
                            j--;
```

```
                    if(j<=0 && RouletteSlot[0]>SelectedSlot)
60                          MatingList[i].Partner1 = Virtual[0];
                    else
                            MatingList[i].Partner1 = Virtual[j+1];

                    SelectedSlot+=k;
65          }

            delete [] RouletteSlot;
    }
```

In Zeile 25 wird zunächst getestet, ob der Wert des Parameters, der die Anzahl der auszuwählenden Individuen angibt, kleiner als eins ist. Sollte dies der Fall sein, dann wird als Parameterwert die Größe der virtuellen Population angenommen (Zeile 27) und eine entsprechende Meldung ausgegeben (Zeilen 28-31). In Zeile 35 wird sodann geprüft, ob mehr Individuen auszuwählen sind, als im momentanen Array zur Aufnahme der Mating-List Platz hätten. Ist dies der Fall, dann wird die Mating-List unter Zuhilfenahme der Funktion `ReSizeMatingList(int)` redimensioniert (Zeile 36). Andernfalls wird einfach der Zähler, der die Anzahl der Einträge in der Mating-List aufnimmt, entsprechend eingestellt (Zeile 38). In Zeile 40 wird dann ein Array erzeugt, dessen Inhalt in Folge das Roulettrad der "stochastic universal sampling"-Methode abbilden soll. Wie diese Abbildung erfolgt, wurde bereits im Kapitel 6.3.2.6 gezeigt. In der in Zeile 43 beginnenden Schleife wird das durch `RouletteSlots` referenzierte Array entsprechend mit den kumulierten target sampling rates aufgefüllt (Zeilen 45-47). In Zeile 50 wird der Abstand k zweier benachbarer "Radspeichen" des "Selektionsrades" berechnet (siehe Abbildung 29). In Zeile 51 wird eine Zufallszahl zwischen 0 und k erzeugt, die die Zufallsposition $rand$ des "Selektionsrades" bestimmt. In der in Zeile 53 beginnenden Schleife erfolgt dann die eigentliche Auswahl von n Individuen. In der in Zeile 52 beginnenden Schleife wird getestet, in welches Segment die erzeugte Zufallszahl fällt. Beginnend von der obersten Grenze des letzten Sektors wird das `RouletteSlot`-Array in der in Zeile 56 beginnenden Schleife solange durchlaufen, bis eine obere Sektorgrenze gefunden wurde, die kleiner ist als der gewählte Punkt, oder bis das ganze Roulettrad durchlaufen wurde. In Zeile 58 wird das so gewählte Individuum als nächster `Partner1`-Eintrag der Mating-List hinzugefügt. Die Abfrage in Zeile 59 ist nötig, da die untere Begrenzung des nullten Sektors (die gleich null ist) nicht im durch `RouletteSlots` referenzierten Array abgebildet wird. Nach jedem Schleifendurchlauf wird der Selektionspunkt der nächsten "Radspeiche" durch Dazuaddieren des Speichenabstandes k ermittelt (Zeile 64). Die Zusammenhänge zwischen dem gedachten Roulettrad und dem `RouletteSlots`-Array können auch der Abbildung 29 entnommen werden. Nach erfolgter Selektion wird schließlich das erzeugte temporäre Array in Zeile 67 wieder vernichtet.

6.3.4.5 RandomMate()

Diese Funktion füllt die `Partner2`-Einträge der Mating-List durch einfache Zufallsauswahl aus den `Partner1`-Einträgen. Dadurch ist nach Aufruf dieser Funktion jedem durch die `Partner1`-Einträge der Mating-List referenzierten Individuum ein durch einen `Partner2`-Eintrag referenziertes Individuum zugeordnet.

```
    /*-------------------------------------------------------
      void Population::RandomMate()

      Selects a Partner2 for every Partner1 in the
 5    MatingList randomly.
      -------------------------------------------------------
    */
    void Population::RandomMate()
    {
10          int i;
            static Uniran Mating(MATINGDICESEED);

            for(i=0;i<MatingListNoOfEntries;i++)
                    MatingList[i].Partner2 =
15                  MatingList[Mating.dice(MatingListNoOfEntries)].Partner1;
    }
```

In der in Zeile 13 beginnenden Schleife wird die gesamte Mating-List durchlaufen. Jedem Eintrag wird ein `Partner2` zugewiesen, der zufällig aus den `Partner1`-Einträgen ausgewählt wird (Zeilen 14 und 15). Die Zufallsauswahl geschieht durch die Funktion `dice(int)` des Zufallszahlengenerators `Uniran`. Die Funktion `dice(int)` liefert eine ganzzahlige Zufallszahl zwischen null (inklusive) und dem Parameterwert (exklusive).

6.3.4.6 RandomPermutationMate()

Diese Funktion füllt die `Partner2`-Einträge der Mating-List durch eine Zufallspermutation der `Partner1`-Einträge diese Liste. Dadurch ist nach Aufruf dieser Funktion jedem der durch die `Partner1`-Einträge der Mating-List referenzierten Individuen ein durch einen `Partner2`-Eintrag referenziertes Individuum zugeordnet.

```
    /*-------------------------------------------------------
      void Population::RandomPermutationMate()

      Selects a Partner2 for every Partner1 in the
 5    MatingList by random permutation
      -------------------------------------------------------
    */
    void Population::RandomPermutationMate()
    {
10          int i;
            int j;
            StartNode* Temp;
            static Uniran Mating(PERMUTSEED);

15          for(i=0;i<MatingListNoOfEntries;i++)
                    MatingList[i].Partner2 =
                    MatingList[i].Partner1;

            for(i=0;i<MatingListNoOfEntries-1;i++)
20          {
                    j = i + Mating.dice(MatingListNoOfEntries-i);
                    Temp = MatingList[i].Partner2;
                    MatingList[i].Partner2 = MatingList[j].Partner2;
                    MatingList[j].Partner2 = Temp;
25          }
    }
```

In der in Zeile 15 beginnenden Schleife werden zunächst alle `Partner1`-Einträge in die `Partner2`-Einträge umkopiert (Zeile 16 und 17). Danach werden die `Partner2`-Einträge in der in Zeile 19 beginnenden Schleife in einem *in situ* Verfahren verwürfelt: Aus den noch nicht verwürfelten Einträgen wird zufällig einer herausgewählt (Zeile 21). Der momentan aktuelle Eintrag wird dann mit dem herausgewählten vertauscht (Zeilen 21-23). Dies wird solange gemacht, bis alle Einträge verwürfelt sind. Die Zufallsauswahl geschieht durch die Funktion `dice(int)` des Zufallszahlengenerators `Uniran`. Die Funktion `dice(int)` liefert eine ganzzahlige Zufallszahl zwischen null (inklusive) und dem Parameterwert (exklusive).

6.3.5 Hilfsfunktionen

6.3.5.1 `ReSizeActualPop(int)`

Diese Funktion erzeugt das in `Actual` referenzierte Array neu und vernichtet das alte. Der Parameter gibt dabei an, wieviele Einträge in dem neuen Array Platz finden müssen. Diese Funktion erzeugt jedoch stets ein größeres Array, um Pufferplatz für eine einfach handzuhabende Erweiterung der tatsächlichen Population zu schaffen. Diese Puffergröße wird dynamisch verwaltet und liegt stets in der Größenordnung der bisherigen Population. Die im Parameter angegebene Populationsgröße wird in der Variablen `ActualPopSize` gespeichert.

```
    /*--------------------------------------------------------
      void Population::ReSizeActualPop(int NewSize)

      Resizing the actual population array in steps of the
 5    current size of the array, not saving the content
      of the old array.
      --------------------------------------------------------
    */
    void Population::ReSizeActualPop(int NewSize)
10  {
            RootNode* NewActualPop;
            int NewArraySize;
            int SizingSteps;

15          if(NewSize>ActualPopSize)
            {
                    if(!ActualPopSize)
                            SizingSteps = NewSize;
                    else
20                          SizingSteps = ActualPopSize;

                    NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;

                    NewActualPop = new RootNode[NewArraySize];
25                  delete [] Actual;

                    Actual = NewActualPop;
                    ActualPopArraySize = NewArraySize;
            }
30          ActualPopSize = NewSize;
    }
```

In Zeile 15 wird zunächst geprüft, ob das in `Actual` referenzierte Array ohnedies ausreichend groß ist. In diesem Fall werden die Zeilen 16-29 übersprungen. Ansonsten wird in Zeile 17 geprüft, ob es überhaupt schon eine tatsächliche Population gibt. Ist dies nicht der Fall, dann wird die Schrittweite für die Erweiterung gleich der im Parameter angegebenen Größe gesetzt (Zeile 18). Andernfalls wird die Schrittweite mit der Größe der bisherigen tatsächlichen Population festgelegt. In Zeile 22 wird dann die neue Arraygröße berechnet. Diese beträgt ein Vielfaches der Schrittweite, jedenfalls aber ist sie größer als die im Parameter angegebene Größe. Das Array selbst wird in Zeile 24 erzeugt. In Zeile 25 wird das bisherige Array gelöscht. Zeilen 27 und 28 legen fest, daß das neu erzeugte Array das nunmehrige `Actual`-Array sein soll. In Zeile 30 wird festgehalten, daß die Populationsgröße gleich dem im Parameter übergebenen Wert ist.

6.3.5.2 ReSizeVirtualPop(int,int)

Diese Funktion erzeugt das in `Virtual` referenzierte Array neu und vernichtet das alte. Der erste Parameter gibt dabei an, wieviele Einträge in dem neuen Array Platz finden müssen. Diese Funktion erzeugt jedoch stets ein größeres Array, um Pufferplatz für eine einfach handzuhabende Erweiterung der virtuellen Population zu schaffen. Diese Puffergröße wird dynamisch verwaltet und liegt stets in der Größenordnung der bisherigen Population. Die im ersten Parameter angegebene Populationsgröße wird in der Variablen `VirtualPopSize` gespeichert. Der zweite Parameter legt fest, ob der Inhalt des alten Arrays in das neu erzeugte mit übernommen werden soll.

```
    /*-------------------------------------------------------
      void Population::ReSizeVirtualPop(int NewSize, int SaveOld)

      Resizing the virtual population array in steps of the
 5    current size of the Array. Depending on SaveOld, the content
      of the old virtual population array is saved or not.
      -------------------------------------------------------
    */
    void Population::ReSizeVirtualPop(int NewSize, int SaveOld)
10  {
            StartNode** NewVirtualPop;
            int NewArraySize;
            int SizingSteps;
            int i;
15
            if(NewSize>VirtualPopSize)
            {
                    if(!VirtualPopSize)
                    {
20                          if(!ActualPopSize)
                                    SizingSteps = NewSize;
                            else
                                    SizingSteps = ActualPopSize;
                    }
25                  else
                            SizingSteps = VirtualPopSize;

                    NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;
                    NewVirtualPop = new StartNode*[NewArraySize];
30
                    if(SaveOld)
```

```
                    for(i=0;i<VirtualPopSize;i++)
                        NewVirtualPop[i] = Virtual[i];

35              delete [] Virtual;
                Virtual = NewVirtualPop;
                VirtualPopArraySize = NewArraySize;
            }
            VirtualPopSize = NewSize;
40  }
```

In Zeile 16 wird zunächst geprüft, ob das in `Virtual` referenzierte Array ohnedies ausreichend groß ist. In diesem Fall werden die Zeilen 17-38 übersprungen. Ansonsten wird in Zeile 18 geprüft, ob es überhaupt schon eine virtuelle Population gibt. Ist dies nicht der Fall, dann wird weiter getestet, ob es bereits eine tatsächliche Population gibt (Zeile 20). Trifft dies zu, dann wird die Schrittweite für die Erweiterung des Arrays mit der tatsächlichen Populationsgröße gleichgesetzt, ansonsten mit der im Parameter übergebenen Größe. Gibt es aber bereits eine virtuelle Population, so wird die Schrittweite gleich der Größe der virtuellen Population gesetzt. In Zeile 28 wird dann die neue Arraygröße berechnet. Diese beträgt ein Vielfaches der Schrittweite, jedenfalls aber ist sie größer als die im Parameter angegebene Größe. Das Array selbst wird in Zeile 29 erzeugt. In Zeile 31 wird geprüft, ob der Inhalt des bisherigen Arrays in das neue Array übernommen werden soll. Ist das der Fall, dann wird in der in Zeile 32 beginnenden Schleife der Inhalt des alten Arrays in das neue Array umkopiert (Zeile 33). In Zeile 35 wird das bisherige Array gelöscht. Zeilen 36 und 37 legen fest, daß das neu erzeugte Array das nunmehrige `Virtual`-Array sein soll. In Zeile 39 wird festgehalten, daß die virtuelle Populationsgröße gleich dem im Parameter übergebenen Wert ist.

6.3.5.3 ReSizeMatingList(int)

Diese Funktion erzeugt das in `MatingList` referenzierte Array neu und vernichtet das alte. Der Parameter gibt dabei an, wieviele Einträge in dem neuen Array Platz finden müssen. Diese Funktion erzeugt jedoch stets ein größeres Array, um Pufferplatz für eine einfach handzuhabende Erweiterung der MatingList zu schaffen. Diese Puffergröße wird dynamisch verwaltet und liegt stets in der Größenordnung der bisherigen MatingList. Die im Parameter angegebene Populationsgröße wird in der Variablen `ActualPopSize` gespeichert.

```
    /*-------------------------------------------------------
      void Population::ReSizeMatingList(int NewSize)

      Resizing the MatingList-array in steps of the
 5    current size of the array.
      -------------------------------------------------------
    */
    void Population::ReSizeMatingList(int NewSize)
    {
10          MatingListEntry* NewMatingList;
            int NewArraySize;
            int SizingSteps;
```

```
             if(NewSize>MatingListArraySize)
15           {
                     if(!MatingListNoOfEntries)
                             SizingSteps = NewSize;
                     else
                             SizingSteps = MatingListNoOfEntries;
20
                     NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;

                     NewMatingList = new MatingListEntry [NewArraySize];
                     delete [] MatingList;
25
                     MatingList = NewMatingList;
                     MatingListArraySize = NewArraySize;
             }
             MatingListNoOfEntries = NewSize;
30   }
```

In Zeile 14 wird zunächst geprüft, ob das in `MatingList` referenzierte Array ohnedies ausreichend groß ist. In diesem Fall werden die Zeilen 15-28 übersprungen. Ansonsten wird in Zeile 17 geprüft, ob es überhaupt schon Einträge in der MatingList gibt. Ist dies nicht der Fall, dann wird die Schrittweite für die Erweiterung gleich der im Parameter angegebenen Größe gesetzt (Zeile 18). Andernfalls wird die Schrittweite mit der Anzahl der bisherigen Einträge festgelegt. In Zeile 21 wird dann die neue Arraygröße berechnet. Diese beträgt ein Vielfaches der Schrittweite, jedenfalls aber ist sie größer als die im Parameter angegebene Größe. Das Array selbst wird in Zeile 23 erzeugt. In Zeile 24 wird das bisherige Array gelöscht. Zeilen 26 und 27 legen fest, daß das neu erzeugte Array das nunmehrige `MatingList`-Array sein soll. In Zeile 29 wird festgehalten, daß die Anzahl der Einträge in der neuen Mating-List gleich dem im Parameter übergebenen Wert ist.

6.4 Die Farm

Die Klasse `Farm` beinhaltet in ihrem Datenteil die aktuelle und mehrere Vorgängerpopulationen, sowie die Grammatik (`Language`) der in dieser Population enthaltenen Ableitungsbäume. Sie stellt programmintern alle genetischen Operatoren zur Verfügung, die sich auf mehrere Generationen beziehen (z.B. Kreuzung oder Fitness-Skalierung über mehrere Generationen), und bietet dem Anwender eine komfortable Schnittstelle zur einfachen Anwendung sämtlicher genetischer Operatoren (im Wesentlichen die `Set...()`-Funktionen und die `NextStep()`-Anweisung. Hinzu kommen einige sehr nützliche Zusatzfunktionen die z.B. das Speichern und Laden der Farm in eine Datei ermöglichen.

6.4.1 Die Datenstruktur

Die im Datenteil der Klasse `Farm` enthaltenen Variablen haben folgende Bedeutung:

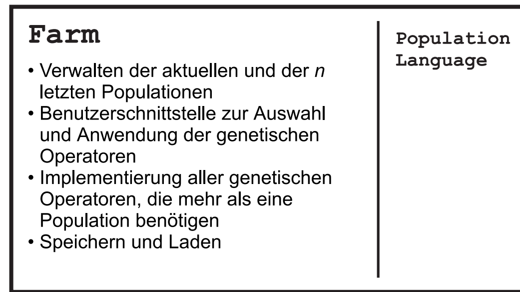


Abbildung 30: Die Klasse `Farm`

- `L`: Beinhaltet die Grammatik (ein Objekt der Klasse `Language`) der in den Populationen dieser Farm enthaltenenen Individuen.
- `Current`: Ist ein Pointer auf die momentan aktuelle Population.
- `ThePopulation`: Referenziert ein Array, welches die Populationen beinhaltet.
- `PopulationIndex`: Gibt den Index der aktuellen Population in diesem Array an.
- `UnivProb`: Gibt an, ob die Initialpopulation gleichverteilt (`EXACTUNIF`), gleichverteilt nach Größe (`SIZEUNIV`) oder nicht gleichverteilt (`NONUNIF`) war.
- `History`: Gibt an, wieviele Vorgängergenerationen der aktuellen Population gespeichert sind.
- `MaxDerivDepth`: Gibt die momentan zulässige maximale Ableitungstiefe an.
- `SmallestSubTreeOfVirtualPop`: Gibt die kleinste Ableitungstiefe, mit der ein Unterbaum noch in die virtuelle Population aufgenommen wird, an. Ein Spezialfall ist der Eintrag von null: In diesem Fall werden überhaupt keine Unterableitungsbäume in die virtuelle Population mit aufgenommen.
- `LinearRankSelectParameter`: Beinhaltet den a_{min}-Parameter für die nächste auszuführende Linear-Rank-Selektion (siehe Kapitel 6.3.4.2).
- `Sampling`: Gibt die anzuwendende Sampling-Methode an (`STOCHASTIC_UNIVERSAL` oder `STOCH_SAMPL_WITH_REPL`).
- `NoOfMates`: Gibt die Länge der im nächsten Sampling-Schritt zu erstellenden MatingList an.
- `Mating`: Gibt den anzuwendenen Mating-Operator an (`RANDOM` oder `RANDOM_PERMUTATION`).
- `Selection`: Gibt das anzuwendende Selektionsverfahren an (`PROPORTIONAL` oder `LINEAR_RANK`).

- `NoOfCrossOverPoints`: Gibt die Anzahl der Kreuzungspunkte für die nächste Kreuzung an.
- `CrossOverProbability`: Gibt an, mit welcher Wahrscheinlichkeit die Kreuzung durchzuführen ist.
- `MutationPropability`: Gibt an, mit welcher Wahrscheinlichkeit der Mutationsoperator anzuwenden ist.
- `SaveBest`: Gibt an, ob das beste Individuum der Vorgeneration der neu erzeugten Population nach der Mutation hinzugefügt werden soll.
- `DynamicFitnessScalingParameter`: Gibt den Parameter für die dynamische Fitness-Skalierung an (siehe Kapitel 6.4.5.2).
- `Goal`: Gibt an, ob es sich bei der Optimierungsaufgabe um eine Maximierungsaufgabe (`MAXIMIZE`) oder eine Minimierungsaufgabe handelt (`MINIMIZE`).
- `double (*FitnessAdjustmentFunction) (double)`: Referenziert die Funktion, mit der die Fitnessadjustierung durchgeführt werden soll. Kann auf `NULL` gesetzt werden, wenn keine Fitnessadjustierung durchgeführt werden soll.
- `double (*SelectionHeuristic) (double)`: Referenziert eine Funktion, die als Selektionsheuristik verwendet wird. Kann auf `NULL` gesetzt werden, wenn keine Selektionsheuristik angewendet werden soll.
- `SelectionHeuristicBasis`: Gibt an, auf welche Werte sich die Selektionsheuristik bezieht: Auf die Ableitungstiefen (`TREESIZE`) oder auf die Suchraumgrößen der Knotensymbole (`SEARCHSPACESIZE`).

6.4.2 Erzeugung und Vernichtung der Farm

6.4.2.1 Farm()

Der Leerkonstruktor sollte unter normalen Umständen vom Anwender nicht aufgerufen werden, und wurde nur der Vollständigkeit halber implementiert.

```
    /*-------------------------------------------------------
      Farm::Farm()

      Empty farm constructor.
 5    -------------------------------------------------------
    */
    Farm::Farm()
    {
            History = 1;
10          ThePopulation = new Population[2];
            PopulationIndex = 0;
            Current = ThePopulation;
            MaxDerivDepth = 16;
            UnifProb = NONUNIF;
15          SetDefaultMethods();
    }
```

In Zeile 9 wird festgelegt, daß maximal eine Generation zurückgerechnet werden kann. In Zeile 10 wird sodann das durch `ThePopulation` referenzierte, zwei Leerpopulationen beinhaltende, Array erzeugt. Die aktuelle Population wird zugewiesen (Zeilen 11 und 12) und die maximal zulässige Ableitungstiefe mit 16 begrenzt (Zeile 13). In Zeile 14 wird festgehalten, daß keine gleichverteilte Population erzeugt wurde. Schließlich werden in Zeile 15 unter Zuhilfenahme der Funktion `SetDefaultMethods` die Standardoperatoren eingestellt.

6.4.2.2 Farm(int, int, int, char*, int)

Dieser Konstruktor soll vom Anwender bei der Erzeugung einer neuen Farm verwendet werden. Im ersten Parameter kann die Größe der Anfangspopulation angegeben werden. Der zweite Parameter gibt an, wieviele Vorgänger-Generationen gleichzeitig im Speicher zu halten sind. Der dritte Parameter indiziert die Art der Initialisierung: gleichverteilt (`EXACTUNIF`), gleichverteilt nach Größe (`SIZEUNIV`) oder nicht gleichverteilt (`NONUNIF`). Im vierten Parameter ist der Name der Grammatikdefinitionsdatei (siehe Kapitel 5.1) anzugeben. Der letzte Parameter schließlich gibt die kleinste Ableitungstiefe, mit der ein Unterableitungsbaum noch in die virtuelle Population aufgenommen wird, an. Wird an dieser Stelle null angegeben, so werden überhaupt keine Unterableitungsbäume in die virtuelle Population aufgenommen. Alle nicht definierten Parameter (z.B. die zu wählenden genetischen Operatoren) werden auf Standardwerte eingestellt.

```
    /*-------------------------------------------------------
      Farm::Farm(int psize, int hist, int uniform, char* filename, int mdd)

      Constructor for new farm.
 5
      psize .... gives the initital (actual) population size
      hist  .... gives the number of cached old populations
      uniform .. initial population NONUNIF, SIZEUNIF or EXACTUNIF
      filename   name of language definition file (BNF format)
10    mdd ...... maximium derivation depth of the smallest s
                 subtree in the virtual population
      -------------------------------------------------------
    */
    Farm::Farm(int psize, int hist, int uniform, char* filename, int mdd)
15  {
            int i;

            History = ( (hist<1) ? 1 : hist );
            mdd = ( (mdd<1) ? 1 : mdd);
20
            i = L.Load(filename);

            if(!i)
            {
25                  cout << "ERROR while loading language definition file!" << endl << endl;
                    History = 1;
                    ThePopulation = new Population[2];
                    PopulationIndex = 0;
                    Current = ThePopulation;
30                  MaxDerivDepth = 16;
                    UnifProb = NONUNIF;
                    SetDefaultMethods();
            }
```

```
            else
35          {
                    PopulationIndex = 0;
                    MaxDerivDepth = mdd;
                    if(uniform==EXACTUNIF)
                    {
40                          cerr << "Exact uniform initialization not yet implemented.";
                            cerr << endl << "Uniform size heuristic initialisation ";
                            cerr << "instead." << endl;
                            UnifProb = SIZEUNIF;
                    }
45                  else
                            UnifProb = uniform;

                    ThePopulation =
                    new Population[History+1](&L,psize,MaxDerivDepth);
50
                    if(UnifProb == SIZEUNIF)
                            ThePopulation[0].
                            GenerateAlmostUniformActualPop(&L,psize,MaxDerivDepth);

55                  Current = ThePopulation;
                    SetDefaultMethods();
            }
    }
```

In Zeile 17 wird zunächst geprüft, ob der Wert des zweiten Parameters die zulässige untere Grenze von eins unterschreitet, und gegebenenfalls auf eins korrigiert. Zeile 18 macht das gleiche mit dem Wert des letzten Parameters. Sodann wird in Zeile 21 versucht, unter Zuhilfenahme der Funktion `Language::Load(char*)` die im vierten Parameter angegebene Sprachdefinitionsdatei zu laden. In Zeile 23 wird getestet, ob dieser Versuch fehlgeschlagen ist. Sollte dies der Fall sein, dann wird in Zeile 25 eine entsprechende Fehlermeldung ausgegeben, und die Farm in den Zeilen 26-32 genauso wie im Leerkonstruktor initialisiert. Hat es beim Laden der Sprachdefinitionsdatei keine Probleme gegeben, so fährt das Programm in Zeile 36 fort. Da wird zunächst die erste der erzeugten Populationen zur aktuellen Population erklärt und die maximale Ableitungstiefe mit dem im letzten Parameter gegebenen Wert festgelegt (Zeile 37). In Zeile 38 wird getestet, ob im dritten Parameter eine gleichverteilte Ausgangspopulation gefordert wurde. Ist dies der Fall, dann wird in den Zeilen 40-42 eine Meldung ausgegeben, die anzeigt, daß diese Art der Initialisierung noch nicht implementiert wurde, und es wird als Initialisierungsmethode die in Kapitel 4.1.2 vorgestellte Heuristik festgelegt (Zeile 43). Ansonsten wird die im dritten Parameter angegebene Initialisierungsmethode festgelegt (Zeile 46). Sodann wird ein Array mit einer entsprechenden Anzahl von Populationen erzeugt und zum `ThePopulation`-Array erklärt (Zeilen 48 und 49). In Zeile 51 wird danach geprüft, ob eine Initialisierung der Ausgangspopulation nach der in Kapitel 4.1.2 beschriebenen Heuristik erwünscht ist. In diesem Fall wird die erste Population im durch `ThePopulation` referenzierten Array unter Zuhilfenahme der Funktion `Population::GenerateAlmostUniformActualPop(...)` neu initialisiert (Zeile 52 und 53). In Zeile 55 wird der Pointer der aktuellen Population auf die Ausgangspopulation gesetzt, und in Zeile 56 werden schließlich unter Zuhilfenahme der Funktion `SetDefaultMethods` die Standardmethoden und Standardparameter eingestellt.

6.4.2.3 Der Destruktor von Farm()

Der Destruktor vernichtet alle im Speicher gehaltenen Populationen.

```
    /*-------------------------------------------------------
      Farm::~Farm()

      Destructor:
 5    Deleting all population of this farm.
      -------------------------------------------------------
    */
    Farm::~Farm()
    {
10          if(ThePopulation!=NULL)
                    delete [] ThePopulation;
    }
```

In Zeile 10 wird geprüft, ob es überhaupt Populationen in dieser Farm gibt (Regelfall), und sodann wird der durch diese Populationen belegte Speicher in Zeile 11 freigegeben.

6.4.3 Speichern und Laden

6.4.3.1 Save(char*)

Diese Funktion speichert die gesamte Farm mit allen Populationen auf eine Datei mit dem im Parameter angegebenen Namen. Das Dateiformat ist in Kapitel 5.3.6 erläutert.

```
    /*------------------------------------------
      int Farm::Save(char* FileName)

      Saves the whole Population and the Grammar
 5    -----------------------------------------
    */
    int Farm::Save(char* FileName)
    {
            ofstream OutputFile;
10          int i;
            int j;
            int k;
            int CurrentGeneration;
            int LastGeneration;
15          int Parent1;
            int Parent2;
            int Best;
            int Worst;

20          if(Current==NULL)
            {
                    cerr << "Error Farm::Save ... No current population to save!" << endl;
                    return(FALSE);
            }
25
            if(FileName==NULL)
            {
                    cerr << "Error Farm::Save ... No filename!" << endl;
                    return(FALSE);
```

```
30          }

            OutputFile.open(FileName);

            if(!OutputFile)
35          {
                    cerr << "Error Farm::Save ... Unable to create file ";
                    cerr  << FileName << endl;
                    return(FALSE);
            }
40
            // FILE HEADER

            CurrentGeneration = PopulationIndex;
            OutputFile << FILEFORMAT << '#';
45          OutputFile << History << '#';
            OutputFile << UnifProb << '#';
            OutputFile << MaxDerivDepth << '#' << endl;

            for(j=0;j<=History;j++)
50          {
                    // ACTUAL POPULATION

                    OutputFile << 'P' << endl;
                    OutputFile << ThePopulation[CurrentGeneration].ActualPopSize << '#';
55
                    OutputFile <<
                    ThePopulation[CurrentGeneration].SmallestSubTreeOfVirtualPop << '#';

                    OutputFile << ThePopulation[CurrentGeneration].ParentsAvailable << '#';
60
                    Best = 0;
                    Worst = 0;

                    if(ThePopulation[CurrentGeneration].BestIndividual!=NULL)
65                  {
                            for(k=ThePopulation[CurrentGeneration].VirtualPopSize-1;k>=0;k--)
                            {
                                    if(ThePopulation[CurrentGeneration].Virtual[k] ==
                                    ThePopulation[CurrentGeneration].BestIndividual)
70                                          Best = k+1;

                                    if(ThePopulation[CurrentGeneration].Virtual[k] ==
                                    ThePopulation[CurrentGeneration].WorstIndividual)
                                            Worst = k+1;
75                          }
                    }

                    OutputFile << Best << '#' << Worst << '#';

80                  OutputFile << setprecision(LOADFLPOINTBFF-10) <<
                    ThePopulation[CurrentGeneration].AdjustedFitnessSum << "#" << endl;

                    LastGeneration =
                    (CurrentGeneration==0 ? History : CurrentGeneration-1);
85
                    for(i=0;i<ThePopulation[CurrentGeneration].ActualPopSize;i++)
                    {
                            if(ThePopulation[CurrentGeneration].
                            Actual[i].Parent1==NULL)
90                                  Parent1 = 0;
                            else
                            {
                                    Parent1 = 0;
                                    k = ThePopulation[CurrentGeneration].VirtualPopSize-1;
95
                                    while(k>=0 &&
```

```
                        ThePopulation[CurrentGeneration].ParentsAvailable)
                        {
                                if(
100                             ThePopulation[LastGeneration].Actual+k ==
                                ThePopulation[CurrentGeneration].Actual[i].Parent1)
                                {
                                        Parent1 = k+1;
                                        break;
105                             }
                                k--;
                        }
                }

110             if(ThePopulation[CurrentGeneration].
                Actual[i].Parent2==NULL)
                        Parent2 = 0;
                else
                {
115                     Parent2 = 0;
                        k = ThePopulation[CurrentGeneration].VirtualPopSize-1;

                        while(k>=0 &&
                        ThePopulation[CurrentGeneration].ParentsAvailable)
120                     {
                                if(
                                ThePopulation[LastGeneration].Actual+k ==
                                ThePopulation[CurrentGeneration].Actual[i].Parent2)
                                {
125                                     Parent2 = k+1;
                                        break;
                                }
                                k--;
                        }
130             }

                OutputFile << '!' <<
                ThePopulation[CurrentGeneration].Actual[i];

135             OutputFile << '#';
                OutputFile << Parent1 << '#' << Parent2 << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Actual[i].Mutated;
140
                OutputFile << '#' << endl;
        }

        // VIRTUAL POPULATION PARAMETERS
145
        OutputFile << 'V' << endl;

        for(k=0;k<ThePopulation[CurrentGeneration].VirtualPopSize;k++)
        {
150             OutputFile << '!';

                OutputFile << setprecision(LOADFLPOINTBFF-10) <<
                ThePopulation[CurrentGeneration].Virtual[k]->RawFitness << '#';

160             OutputFile <<
                ThePopulation[CurrentGeneration].Virtual[k]->
                StandardizedFitness << '#';

                OutputFile <<
165             ThePopulation[CurrentGeneration].Virtual[k]->
                AdjustedFitness << '#';

                OutputFile <<
```

```
                        ThePopulation[CurrentGeneration].Virtual[k]->
170                     NormalizedFitness << '#';

                        OutputFile <<
                        ThePopulation[CurrentGeneration].Virtual[k]->TargSamplRate << '#';

175                     OutputFile << endl;
                    }

                    CurrentGeneration =
                    (CurrentGeneration==0 ? History : CurrentGeneration-1);
180
                }

                // the grammar....

185             OutputFile << 'L' << endl;
                OutputFile << L;
                OutputFile.close();
                return(TRUE);
            }
```

In Zeile 20 wird zunächst geprüft, ob es in dieser Farm überhaupt zu speichernde Populationen gibt. Sollte dies nicht der Fall sein, dann wird diese Funktion, unter Ausgabe einer entsprechenden Fehlermeldung (Zeile 22), mit `FALSE` als Rückgabewert abgebrochen (Zeile 23). Dann wird in Zeile 26 getestet, ob der Parameter nicht ein `NULL`-String ist. Auch in diesem Fall wird die Funktion, unter Ausgabe einer entsprechenden Fehlermeldung (Zeile 38), mit `FALSE` als Rückgabewert abgebrochen (Zeile 29). In Zeile 32 wird versucht, die Datei mit dem im Parameter angegebenen Namen zu eröffnen. Schlägt dieser Versuch fehl (Zeile 24), dann wird ebenfalls eine entsprechende Fehlermeldung ausgegeben (Zeile 36 und 37), und die Funktion unter Rückgabe von `FALSE`, abgebrochen (Zeile 38). Der eigentliche Speichervorgang beginnt in Zeile 41 mit dem File-Header:

Zunächst wird in der lokalen Variablen `CurrentGeneration` der Index der aktuellen Population gespeichert (Zeile 43). Dann wird in den Zeilen 44-47 der Fileheader in die Datei geschrieben. Der Reihe nach werden geschrieben:

- das Fileformat (momentan 1),
- die Anzahl der im Speicher gehaltenen Vorgängergenerationen,
- die Art der Initialisierung,
- die momentan maximal zulässige Ableitungstiefe.

Alle Werte werden mit einem Nummernsymbol (`#`) abgeschlossen, der ganze Fileheader wird mit einem Zeilenendezeichen beendet (Zeile 47). Danach werden in der in Zeile 49 beginnenden Schleife alle Populationen, beginnend mit der aktuellen Population, durchlaufen. In Zeile 51 beginnt die Ausgabe der tatsächlichen Population, deren Anfang durch die Ausgabe eines "P", gefolgt von einem Zeilenendezeichen gekennzeichnet wird (Zeile 53). Danach folgen, jeweils abgeschlossen durch ein Nummernsymbol (`#`):

- die Größe der tatsächlichen Population (Zeile 54),
- der kleinste Unterableitungsbaum, der noch in den virtuellen Teil dieser Population aufzunehmen ist (Zeile 56 und 57),
- das Flag, das angibt, ob die Elterngeneration dieser Population verfügbar ist (Zeile 59),
- die Nummer des besten Individuums des virtuellen Teils der Population (Zeile 78),
- die Nummer des schlechtesten Individuums des virtuellen Teils der Population (ebenfalls Zeile 78),
- die Summe der adjustierten Fitnesswerte aller im virtuellen Teil der Population vermerkten Individuen (Zeilen 80 und 81).

Abgeschlossen wird dieser Header von einem Zeilenendezeichen (Zeile 81). Die Nummer des besten und des schlechtesten Individuums wird in der in Zeile 66 beginnenden Schleife an Hand des `BestIndividual` und des `WorstIndividual`-Pointers festgestellt. In dieser Schleife wird die ganze virtuelle Population durchlaufen, und jeder Eintrag mit den beiden Pointern verglichen (Zeilen 68-69 und 72-73). Ist der entsprechende Eintrag gefunden, wird dessen Index in der temporären Variable `Best` bzw. `Worst` gesichert (Zeile 70 und 74). In den Zeilen 83 und 84 wird der Index der Elterngeneration der momentan zu speichernden Generation errechnet.

Nach dem Schreiben des Populationsheaders wird in der in Zeile 86 beginnenden Schleife die gesamte tatsächliche Population durchlaufen. Für jedes Individuum werden folgende Werte, jeweils abgeschlossen mit einem Nummernzeichen (`#`) in die Datei geschrieben:

- das Individuum in seiner kurzen Stringrepräsentation (Zeilen 132 und 133),
- die Nummer des ersten Elternteils in der Vorgängergeneration (null für unbekannt) - Zeilen 135 und 136,
- die Nummer des zweiten Elternteils in der Vorgängergeneration (null für unbekannt) - ebenfalls Zeilen 135 und 136,
- das Flag, das angibt, ob dieses Individuum einer Mutation unterzogen wurde (Zeilen 138 und 139).

Eingeleitet wird jedes Individuum mit einem Rufzeichen (Zeile 132), und beendet mit einem Zeilenendezeichen (Zeile 141). Nach der tatsächlichen Population folgt in Zeile 146 die Ausgabe des Headers des virtuellen Teils der Population, der einfach aus einem

"V", gefolgt von einem Zeilenendezeichen besteht. In der in Zeile 148 beginnenden Schleife wird die gesamte virtuelle Population durchlaufen, und für jedes Individuum die folgenden Werte, jeweils abgeschlossen durch ein Nummernsymbol (#), gespeichert:

- der Rohfitnesswert (Zeilen 157-158),
- die standardisierte Fitness (Zeilen 160-162),
- die adjustierte Fitness (Zeilen 164-166),
- die normalisierte Fitness (Zeilen 168-170),
- die target sampling rate (Zeilen 172-174).

Eingeleitet wird jedes Tupel mit einem Rufzeichen (Zeile 150), und beendet mit einem Zeilenendezeichen (Zeile 175). In Zeile 185 wird schließlich nach Ausgabe aller Populationen der Header für die Grammatikdefinition, bestehend aus einem "L", gefolgt von einem Zeilenendezeichen, ausgegeben. Danach die Grammatik in ihrer BNF (Zeile 186). In Zeile 187 schließlich wird die Datei geschlossen und die Funktion, unter Rückgabe von `TRUE`, beendet (Zeile 188).

6.4.3.2 `Farm(char*)`

Mit Hilfe dieses Konstruktors kann eine, mittels der vorigen Funktion in eine Datei gespeicherte Farm wiederhergestellt (geladen) werden. Es ist dabei zu beachten, daß nur die im Dateiformat beschriebenen Werte wiederhergestellt werden, nicht aber die Informationen über die für den nächsten Generationsschritt anzuwendenden genetischen Operatoren, Selektionsheuristiken, etc. Das bedeutet, daß diese Angaben mit Hilfe der `Farm::Set...()`-Befehle wiederhergestellt werden müssen.

Es ist außerdem wichtig zu wissen, daß zwar die Fitnesswerte der geladenen Vorgängergenerationen unverändert wiederhergestellt werden, die Fitnesswerte der aktuellen Population jedoch bereits aufgrund der ausgewählten Fitnessberechnungsmethoden erfolgt. Dies bedeutet, daß z.B. unmittelbar nach Aufruf dieses Konstruktors die Fitnesswerte der aktuellen Population anhand der Defaulteinstellungen ermittelt werden und von den gespeicherten abweichen können. Ändert man jedoch diese Einstellungen mit Hilfe der `Set...()`-Befehle auf die Einstellungen, die zum Zeitpunkt der Speicherung gültig waren, kann man z.B. eine unterbrochene Simulation fortsetzen, als wäre sie nie unterbrochen worden.

```
   /*-------------------------------------------------------
     Farm::Farm(char* FileName)

     Constructor for loading an old farm saved to disk
5    by Farm::Save(char* FileName).
```

```
       Selection, crossover etc methods are not specified
       in the file and set to default values.
       -------------------------------------------------------
10    */
      Farm::Farm(char* FileName)
      {
              int i;
              int j;
15            int k;
              int x;
              ifstream InputFile;
              char CurrentCharacter;
              int LoadingOK;
20            char* StringBuffer;
              char* FloatingNumberBuffer;
              char* NewStringBuffer;
              char** DummyEndPtr;
              int StringBufferSize;
25            int Best;
              int Worst;
              int FileFormat;

              StringBuffer = new char[CHRBUFSIZE];
30            StringBufferSize = CHRBUFSIZE;
              FloatingNumberBuffer = new char[LOADFLPOINTBFF];

              ThePopulation = NULL;
              Current = NULL;
35            SetDefaultMethods();
              LoadingOK = TRUE;

              if(FileName==NULL)
              {
40                    LoadingOK = FALSE;
                      cerr << "Error loading farm! No filename!" << endl;
              }

              if(LoadingOK)
45            {
                      i = L.Load(FileName);
                      if(!i)
                      {
                              LoadingOK = FALSE;
50                            cerr << "Invalid File Format!" << endl;
                      }
              }

              if(LoadingOK)
55            {
                      InputFile.open(FileName);

                      if(!InputFile)
                      {
60                            LoadingOK = FALSE;
                              cerr << "Error loading farm! Invalid filename!" << endl;
                      }
              }

65            if(LoadingOK)
              {


                      // LOADING FILE HEADER PARAMETERS:
70                    // FileFormat, History, UnifProb and MaxDerivDepth

                      for(j=0;j<4;j++)
                      {
```

```
                        i = 0;
75                      InputFile >> CurrentCharacter;
                        FloatingNumberBuffer[i] = '\0';
                        while(!InputFile.eof() && CurrentCharacter!='#')
                        {
                                if(i<LOADFLPOINTBFF)
80                              {
                                        FloatingNumberBuffer[i] = CurrentCharacter;
                                        i++;
                                        FloatingNumberBuffer[i] = '\0';
                                }
85                              InputFile >> CurrentCharacter;
                        }

                        if(CurrentCharacter=='#')
                        {
90                              switch(j)
                                {
                                        case 0:
                                                FileFormat =
                                                int(strtod(FloatingNumberBuffer,
95                                              DummyEndPtr));

                                        case 1:
                                                History =
                                                int(strtod(FloatingNumberBuffer,
100                                             DummyEndPtr));

                                                break;
                                        case 2:
                                                UnifProb =
105                                             int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;
                                        case 3:
110                                             MaxDerivDepth =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;
115                             }
                        }
                        else
                        {
                                LoadingOK = FALSE;
120                             cerr << "Invalid file format!" << endl;
                                break;
                        }
                }
        }
125
        if(LoadingOK)
        {
                k = History;
                ThePopulation = new Population[History+1];
130
                Current = ThePopulation+History;
                PopulationIndex = History;
        }

140     // LOADING THE POPULATIONS

        while(k>=0 && LoadingOK)
        {

145
```

```
                // SEARCHING START OF NEXT ACTUAL POPULATION

                while(!InputFile.eof() && CurrentCharacter!='P')
                        InputFile >> CurrentCharacter;
150
                LoadingOK = (CurrentCharacter=='P');

                if(!LoadingOK)
                        break;
155

                // LOADING POPULATION HEADER PARAMETERS:
                // ActualPopSize, SmallestSubTreeOfVirtualPop, ParentsAvailable
                // BestIndividual, WorstIndividual, AdjustedFitnessSum
160
                for(j=0;j<6;j++)
                {
                        i = 0;
                        InputFile >> CurrentCharacter;
165                     FloatingNumberBuffer[i] = '\0';
                        while(!InputFile.eof() && CurrentCharacter!='#')
                        {
                                if(i<LOADFLPOINTBFF)
                                {
170                                     FloatingNumberBuffer[i] = CurrentCharacter;
                                        i++;
                                        FloatingNumberBuffer[i] = '\0';
                                }
                                InputFile >> CurrentCharacter;
175                     }

                        if(CurrentCharacter=='#')
                        {
                                switch(j)
180                             {
                                        case 0:
                                                ThePopulation[k].ActualPopSize =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));
185
                                                break;

                                        case 1:
                                                ThePopulation[k].
190                                             SmallestSubTreeOfVirtualPop =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;
195
                                        case 2:
                                                ThePopulation[k].ParentsAvailable =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));
200
                                                break;

                                        case 3:
                                                Best =
205                                             int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;

210                                     case 4:
                                                Worst =
                                                int(strtod(FloatingNumberBuffer,
```

```
                                        DummyEndPtr));

215                                     break;

                                case 5:
                                        ThePopulation[k].AdjustedFitnessSum =
                                        strtod(FloatingNumberBuffer,
220                                     DummyEndPtr);
                        }
                }
                else
                {
225                     LoadingOK = FALSE;
                        cerr << "Invalid file format!" << endl;
                        break;
                }
        }
230
        if(!LoadingOK)
                break;


235     // LOADING ACTUAL POPULATION

        ThePopulation[k].Actual =
        new RootNode[ThePopulation[k].ActualPopSize](
        &L, MaxDerivDepth, ThePopulation+k);
240
        for(i=0;i<ThePopulation[k].ActualPopSize;i++)
        {


245             // SEARCHING START OF NEXT ACTUAL POPULATION MEMBER

                while(!InputFile.eof() && CurrentCharacter!='!')
                        InputFile >> CurrentCharacter;

250             // LOADING ACTUAL POPULATION MEMBER

                j = 1;
                StringBuffer[0] = '\0';

255             InputFile >> CurrentCharacter;

                while(!InputFile.eof() &&
                (CurrentCharacter=='0' ||
                CurrentCharacter=='1' ||
260             CurrentCharacter=='2' ||
                CurrentCharacter=='3' ||
                CurrentCharacter=='4' ||
                CurrentCharacter=='5' ||
                CurrentCharacter=='6' ||
265             CurrentCharacter=='7' ||
                CurrentCharacter=='8' ||
                CurrentCharacter=='9' ||
                CurrentCharacter=='.'))
                {
270                     if(j==StringBufferSize)
                        {
                                NewStringBuffer =
                                new char[StringBufferSize + CHRBUFSIZE];
                                for(k=0;k<StringBufferSize;k++)
275                                     NewStringBuffer[k] =
                                        StringBuffer[k];

                                delete [] StringBuffer;
                                StringBuffer = NewStringBuffer;
```

```
280                                 StringBufferSize+=  CHRBUFSIZE;
                            }

                            StringBuffer[j-1] = CurrentCharacter;
                            StringBuffer[j] = '\0';
285                         j++;
                            InputFile >> CurrentCharacter;
                    }

                    LoadingOK =
290                 ThePopulation[k].Actual[i].Set(StringBuffer);

                    if(!LoadingOK)
                            break;

295
                    // LOADING PARAMETERS PARENT1, PARENT2, MUTATED

                    for(j=0;j<3;j++)
                    {
300                         x = 0;
                            InputFile >> CurrentCharacter;
                            FloatingNumberBuffer[x] = '\0';
                            while(!InputFile.eof() && CurrentCharacter!='#')
                            {
305                                 if(k<LOADFLPOINTBFF)
                                    {
                                            FloatingNumberBuffer[x] = CurrentCharacter;
                                            x++;
                                            FloatingNumberBuffer[x] = '\0';
310                                 }
                                    InputFile >> CurrentCharacter;
                            }

                            if(CurrentCharacter=='#')
315                         {
                                    switch(j)
                                    {
                                            case 0:
                                                    // RELATIVE ADRESS OF PARENT1!!!
320                                                 ThePopulation[k].Actual[i].
                                                    Parent1+=
                                                    int(strtod(FloatingNumberBuffer,
                                                    DummyEndPtr));

325                                                 break;

                                            case 1:
                                                    // RELATIVE ADRESS OF PARENT2!!!
                                                    ThePopulation[k].Actual[i].
330                                                 Parent2+=
                                                    int(strtod(FloatingNumberBuffer,
                                                    DummyEndPtr));

                                                    break;
335
                                            case 2:
                                                    ThePopulation[k].Actual[i].
                                                    Mutated =
                                                    int(strtod(FloatingNumberBuffer,
340                                                 DummyEndPtr));

                                                    break;
                                    }
                            }
345                         else
                            {
```

```
                                LoadingOK = FALSE;
                                cerr << "Invalid file format!" << endl;
                                break;
350                     }
                }
        }

        if(!LoadingOK)
355             break;


        // CONVERTING RELATIVE ADRESSES OF PARENT1 AND PARENT2 TO ABSOLUTE

360     if(k<History)
        {
                for(i=0;i<ThePopulation[k+1].ActualPopSize;i++)
                {
                        ThePopulation[k+1].Actual[i].Parent1 =
365                     ThePopulation[k].Actual +
                        int(int(ThePopulation[k+1].Actual[i].Parent1)/
                        sizeof(StartNode)) - 1;

                        ThePopulation[k+1].Actual[i].Parent2 =
370                     ThePopulation[k].Actual +
                        int(int(ThePopulation[k+1].Actual[i].Parent2)/
                        sizeof(StartNode)) - 1;
                }
        }
375
        // BUILDING UP VIRTUAL POPULATION

        if(ThePopulation[k].SmallestSubTreeOfVirtualPop == 0)
                ThePopulation[k].GenerateVirtualPop();
380     else
                ThePopulation[k].GenerateVirtualPopInclSubTrees(
                ThePopulation[k].SmallestSubTreeOfVirtualPop);


385     if(Best==0)
                ThePopulation[k].BestIndividual = NULL;
        else
                ThePopulation[k].BestIndividual =
                ThePopulation[k].Virtual[Best-1];
390
        if(Worst==0)
                ThePopulation[k].WorstIndividual = NULL;
        else
                ThePopulation[k].WorstIndividual =
395             ThePopulation[k].Virtual[Worst-1];



        // SEARCHING START OF VIRTUAL POPULATION
400
        while(!InputFile.eof() && CurrentCharacter!='V')
                InputFile >> CurrentCharacter;


405     // LOADING VIRTUAL POPULATION PARAMETERS:
        // RawFitness, StandardizedFitness, AdjustedFitness,
        // NormalizedFitness, TargSamplRate

        for(i=0;i<ThePopulation[k].VirtualPopSize;i++)
410     {


                // SEARCHING START OF NEXT VIRTUAL POPULATION MEMBER PARAMETER
```

```
    // VECTOR
415
    while(!InputFile.eof() && CurrentCharacter!='!')
            InputFile >> CurrentCharacter;

    // LOADING VIRTUAL POPULATION MEMBER PARAMETER VECTOR
420
    for(j=0;j<5;j++)
    {
            x = 0;
            InputFile >> CurrentCharacter;
425         FloatingNumberBuffer[x] = '\0';
            while(!InputFile.eof() && CurrentCharacter!='#')
            {
                    if(x<LOADFLPOINTBFF)
                    {
430                         FloatingNumberBuffer[x] =
                            CurrentCharacter;

                            x++;

435                         FloatingNumberBuffer[x] = '\0';
                    }
                    InputFile >> CurrentCharacter;
            }

440         if(CurrentCharacter=='#')
            {
                    switch(j)
                    {
                            case 0:
445                                 ThePopulation[k].
                                    Virtual[i]->RawFitness =
                                    float(strtod(FloatingNumberBuffer,
                                    DummyEndPtr));

450                                 break;

                            case 1:
                                    ThePopulation[k].
                                    Virtual[i]->StandardizedFitness =
455                                 float(strtod(FloatingNumberBuffer,
                                    DummyEndPtr));

                                    break;

460                         case 2:
                                    ThePopulation[k].
                                    Virtual[i]->AdjustedFitness =
                                    float(strtod(FloatingNumberBuffer,
                                    DummyEndPtr));
465
                                    break;

                            case 3:
                                    ThePopulation[k].
470                                 Virtual[i]->NormalizedFitness =
                                    float(strtod(FloatingNumberBuffer,
                                    DummyEndPtr));

                                    break;
475
                            case 4:
                                    ThePopulation[k].
                                    Virtual[i]->TargSamplRate =
                                    float(strtod(FloatingNumberBuffer,
480                                 DummyEndPtr));
```

```
                                                break;
                                        }
                                }
485                             else
                                {
                                        LoadingOK = FALSE;
                                        cerr << "Invalid file format!" << endl;
                                        break;
490                             }
                        }

                        if(!LoadingOK)
                                break;
495             }

                k--;
        }

500
        if(LoadingOK)
                SetDefaultMethods();
        else
        {
505             cerr << "Error while loading population!" << endl;

                if(ThePopulation!=NULL)
                        delete [] ThePopulation;

510             History = 1;
                ThePopulation = new Population[2];

                PopulationIndex = 0;
                Current = ThePopulation;
515
                MaxDerivDepth = 16;
                UnifProb = NONUNIF;
                SmallestSubTreeOfVirtualPop = 0;
        }
520
        delete [] StringBuffer;
        delete [] FloatingNumberBuffer;
    }
```

In Zeile 29 wird zunächst ein Buffer für einzulesende Zeichenketten reserviert. Die Größe des Buffers wird dabei vorerst mit `CHRBUFSIZE` festgelegt, kann jedoch in der Funktion, wenn erforderlich, vergrößert werden. `CHRBUFSIZE` wurde im Header-File `ga.h` mit 256 festgelegt. Der in Zeile 31 erzeugte Buffer der Größe `LOADFLPOINTBFF` ist innerhalb der Funktion fix, und dient zur Aufnahme von einzulesenden Fließkommazahlen. `LOADFLPOINTBFF` wurde im Header-File `ga.h` mit 255 festgelegt.

In den Zeilen 33 und 34 wird festgelegt, daß vorerst keine Populationen vorhanden sind. Mit Hilfe der `SetDefaultMethods()`-Funktion werden in Zeile 35 die Standardparameter und Standardoperatoren eingestellt. Das Flag `LoadingOK`, dessen Inhalt den korrekten Fortgang der Ladefunktion darstellt, wird in Zeile 36 initialisiert. Bevor der eigentliche Ladevorgang beginnt, wird zunächst in Zeile 38 geprüft, ob als Parameter nicht ein `NULL`-String übergeben wurde. In diesem Fall wird das `LoadingOK`-Flag auf `FALSE` gestellt (Zeile 40), und eine entsprechende Fehlermeldung ausgegeben (Zeile 41). Andernfalls wird in Zeile 46 versucht, aus der im Parameter angegebenen Datei

die Grammatikdefinition mit Hilfe der Funktion `Language::Load(char*)` einzulesen. Mißlingt dies (Zeile 47), so wird das `LoadingOK`-Flag auf `FALSE` gestellt (Zeile 49) und eine entsprechende Fehlermeldung ausgegeben (Zeile 50). Andernfalls wird in Zeile 56 versucht, die im Parameter angegebene Datei zum Lesen zu öffnen. Sollte dies mißlingen (Zeile 58), so wird das `LoadingOK`-Flag ebenfalls auf `FALSE` gestellt (Zeile 60), und eine entsprechende Fehlermeldung ausgegeben (Zeile 61). Andernfalls wird in Zeile 69 mit dem eigentlichen Ladevorgang begonnen.

Zunächst werden in der in Zeile 72 beginnenden Schleife die vier Zahlenwerte des Fileheaders eingelesen. Jede Zahl wird Ziffer für Ziffer in der in Zeile 76 beginnenden Schleife in den vorgesehenen Buffer eingelesen, bis entweder ein Nummernzeichen (`#`) eingelesen wird, oder das Ende der Datei erreicht ist. Der Vergleich in Zeile 79 verhindert einen Bufferüberlauf. Nach jeder eingelesenen Zahl wird in Zeile 88 überprüft, ob zuletzt das abschließende Nummernzeichen eingelesen wurde. Ist dies nicht der Fall, dann handelt es sich offensichtlich um eine syntaktisch nicht korrekte Datei, und das `LoadingOK`-Flag wird auf `FALSE` gesetzt (Zeile 119). Die Schleife wird nach Ausgabe einer entsprechenden Fehlermeldung (Zeile 120) abgebrochen (Zeile 121). Wurde die Zahl korrekt eingelesen, wird sie mit Hilfe der `switch`-Anweisung in Zeile 90 der entsprechenden Variablen zugewiesen. Die Umwandlung erfolgt dabei mit Hilfe der Funktion `strtod(char*,int)`. Nach dem korrekten Einlesen des Fileheaders wird in Zeile 129 ein entsprechend großes Array zur Aufnahme der einzulesenden Populationen erzeugt. Die letzte Population dieses Arrays wird zur aktuellen Population erklärt (Zeilen 131 und 132).

Für jede einzulesende Population wird nun die in Zeile 142 beginnende Schleife einmal durchlaufen. In der in Zeile 148 beginnenden Schleife wird dabei zunächst die Anfangsmarke "P" gesucht. Das weitere Fortfahren wird vom Finden dieser Anfangsmarke abhängig gemacht (Zeilen 151-154). Sodann werden in der in Zeile 161 beginnenden Schleife die sechs Werte des Populationsheaders eingelesen. Das Einlesen der Zahlen erfolgt in den Zeilen 163-175 genauso wie beim Fileheader (Zeilen 75-86). Nach jeder eingelesenen Zahl wird in Zeile 177 überprüft, ob zuletzt das abschließende Nummernzeichen eingelesen wurde. Ist dies nicht der Fall, dann handelt es sich offensichtlich um eine syntaktisch nicht korrekte Datei, und das `LoadingOK`-Flag wird auf `FALSE` gesetzt (Zeile 225). Die Schleife wird nach Ausgabe einer entsprechenden Fehlermeldung (Zeile 226) abgebrochen (Zeile 227). Wurde die Zahl korrekt eingelesen, dann wird sie mit Hilfe der `switch`-Anweisung in Zeile 129 der entsprechenden Variablen zugewiesen. Der Index des besten und der Index des schlechtesten Individuums werden vorerst in den lokalen Variablen `Best` und `Worst` gesichert.

Nachdem der Populationsheader eingelesen wurde, wird mit dem Laden der tatsächlichen Population begonnen. Dazu wird zunächst ein entsprechend großes Array mit Individuen erzeugt und zum `Actual`-Array der Population erklärt (Zeilen 237-239). In der in Zeile 241 beginnenden Schleife werden dann alle Individuen der Population eingelesen. Dazu wird zunächst mit der in Zeile 247 beginnenden Schleife das nächste Rufzeichen in der Datei gesucht, da dieses den Beginn eines Individuums kennzeichnet.

Danach werden in der in Zeile 257 beginnenden Schleife solange Zeichen in den vorgesehenen Buffer eingelesen, solange es sich bei diesen Zeichen um Ziffern oder einen Punkt handelt. Das erste abweichende Zeichen beendet die Schleife. In Zeile 270 wird dabei ständig überprüft, ob der Buffer bereits voll ist. Sollte dies der Fall sein, dann wird in den Zeilen 272 und 273 ein neuer, um `CHRBUFSIZE` größerer Buffer angelegt, und der Inhalt des alten in den neu angelegten Buffer überspielt (Zeilen 274-276). Der alte Buffer wird vernichtet (Zeile 277), und der neue, größere Buffer zum Stringbuffer erklärt (Zeilen 279 und 280). In den Zeilen 289 und 290 wird schließlich das entsprechende Individuum der momentan einzulesenden Population mit Hife der Funktion `RootNode::Set(char*)` mit dem eingelesenen Individuum überschrieben.

Nach dem Einlesen des Individuums werden in der in Zeile 298 beginnenden Schleife die drei jedem Individuum angehängten Parameter eingelesen. Das Einlesen der Zahlen erfolgt in den Zeilen 301-313 genau so wie beim Fileheader (Zeilen 75-86). Nach jeder eingelesenen Zahl wird in Zeile 315 überprüft, ob zuletzt das abschließende Nummernzeichen eingelesen wurde. Ist dies nicht der Fall, dann handelt es sich offensichtlich um eine syntaktisch nicht korrekte Datei, und das `LoadingOK`-Flag wird auf `FALSE` gesetzt (Zeile 347). Die Schleife wird nach Ausgabe einer entsprechenden Fehlermeldung (Zeile 348) abgebrochen (Zeile 349). Wurde die Zahl korrekt eingelesen, dann wird sie mit Hilfe der `switch`-Anweisung in Zeile 317 der entsprechenden Variablen zugewiesen. Die Pointer `Parent1` und `Parent2` werden dabei im Moment nur als relative Adresse mit Offset null gespeichert (Zeilen 320-323 und 329-332). Dies ist nicht anders möglich, da die Elterngeneration, auf die sich diese Pointer beziehen, noch nicht eingelesen und auch noch nicht erstellt wurde. Erst nach dem Einlesen der gesamten tatsächlichen Population können - wenn es sich nicht um die erste eingelesene Population handelt (Zeile 360) - die relativen Adressen in den `Parent1`- und `Parent2`-Pointern der Individuen der Kinder-Population dieser Population in absolute Adressen umgewandelt werden. Dies geschieht in der in Zeile 362 beginnenden Schleife.

Nach dem Einlesen aller Individuen kann die virtuelle Population aufgebaut werden. Dies geschieht, je nach Wert der zu Beginn eingelesenen Variablen `SmallestSubTreeOfVirtualPop` (Zeile 178) mit Hilfe der Funktion `Population::GenererateVirtualPop()` (Zeile 379) oder der Funktion `Population::GenerateVirtualPopIncSubTrees(unsigned int)` (Zeilen 381 und 382). Nach dem Erstellen der virtuellen Population können die Pointer auf das beste und das schlechteste Individuum anhand der eingelesenen und in den lokalen Variablen `Best` und `Worst` gespeicherten Indices gesetzt werden. Dies geschieht in den Zeilen 385-389 und 391-395.

Danach wird mit dem Einlesen der Werte des virtuellen Teils der Population begonnen. In der in Zeile 401 beginnenden Schleife wird zunächst nach der Anfangsmarke "V" gesucht. Wurde diese gefunden, dann wird die in Zeile 409 beginnende Schleife für jeden Eintrag in der virtuellen Population einmal durchlaufen. Die in der Zeile 416 beginnende Schleife sucht dabei zunächst das nächste Rufzeichen, da dieses den Anfang des Wertetupels kennzeichnet. Die in Zeile 421 beginnende Schleife wird nun fünfmal

durchlaufen, für jeden einzulesenden Parameter einmal. Das Einlesen der Zahlen erfolgt in den Zeilen 425-438 genauso wie beim Fileheader (Zeilen 75-86). Nach jeder eingelesenen Zahl wird in Zeile 440 überprüft, ob zuletzt das abschließende Nummernzeichen eingelesen wurde. Ist dies nicht der Fall, dann handelt es sich offensichtlich um eine syntaktisch nicht korrekte Datei, und das `LoadingOK`-Flag wird auf `FALSE` gesetzt (Zeile 487). Die Schleife wird nach Ausgabe einer entsprechenden Fehlermeldung (Zeile 488) abgebrochen (Zeile 489). Wurde die Zahl korrekt eingelesen, dann wird sie mit Hilfe der `switch`-Anweisung in Zeile 442 der entsprechenden Variablen zugewiesen.

In Zeile 501 wird schließlich an Hand des Flags `LoadingOK` geprüft, ob der Ladevorgang erfolgreich war. Trifft dies zu, werden mit Hilfe der Funktion `SetDefaultMethods()` die Standardoperatoren und Standardparameter eingestellt. Andernfalls wird in der Zeile 505 eine entsprechende Fehlermeldung ausgegeben, eventuell inzwischen erstellte Populationen gelöscht (Zeilen 507 und 508) und die Farm so initialisiert, wie dies auch der Leerkonstruktor tut (Zeilen 510-517). In den Zeilen 521 und 522 wird zuletzt noch der durch die Buffer belegte Speicher freigegeben.

6.4.4 Auswahlfunktionen

6.4.4.1 `SetFitnessScaling(unsigned int, int)`

Mit Hilfe dieser Funktion kann der Anwender die im nächsten Generationsschritt anzuwendende Fitness-Skalierung der standardisierten Fitness sowie das Optimierungsziel bestimmen. Der erste Parameter definiert hierbei, über wieviele Vorgängergenerationen die standardisierte Fitness skaliert werden soll, und der zweite Parameter das Optimierungsziel (`MINIMIZE` oder `MAXIMIZE`).

```
    /*--------------------------------------------------------
      void Farm::SetFitnessScaling(unsigned int NewScalingParameter, int NewGoal)

      Selection of fitness-scaling method:
 5
      NewScalingParameter = 0 ... just the current population
                          = 1 ... current and last
                          = 2 ... current, and last two...
                                  etc.
10
      Goal: MAXIMIZE and MINIMIZE
      --------------------------------------------------------
    */
    void Farm::SetFitnessScaling(unsigned int NewScalingParameter, int NewGoal)
15  {
            if(NewGoal!=Goal || NewScalingParameter!=DynamicFitnessScalingParameter)
            {
                    if(NewGoal==MINIMIZE)
                            Goal = MINIMIZE;
20                  else
                    {
                            Goal = MAXIMIZE;
                            if(NewGoal!=MAXIMIZE)
                            {
25                                  cerr << "Warning! Invalid optimization goal!" << endl;
                                    cerr << "Fitness maximization instead." << endl;
                            }
                    }
```

```
30                  DynamicFitnessScalingParameter = NewScalingParameter;
                    ApplyDynamicFitnessScaling(DynamicFitnessScalingParameter,Goal);

                    if(FitnessAdjustmentFunction!=NULL)
                            AdjustFitnessValues();
35                  else
                            Current->InvertFitnessValues();

                    Current->NormalizeFitnessValues();
            }
40  }
```

In Zeile 16 wird zunächst geprüft, ob sich die Parameter im Vergleich zum letzten Aufruf überhaupt geändert haben. Nur wenn dies der Fall ist, wird in Zeile 18 fortgesetzt. Hier wird getestet, ob das neue Optimierungsziel auf Minimierung der Fitness lautet. Ist dies der Fall, dann wird dies entsprechend vermerkt (Zeile 19). Ansonsten wird Fitnessmaximierung als Ziel angenommen (Zeile 22). In Zeile 23 wird geprüft, ob dies im Parameter auch tatsächlich angefordert wurde. Trifft dies nicht zu, dann wird in den Zeilen 25 und 26 eine entsprechende Fehlermeldung ausgegeben. In Zeile 30 wird der erste Parameter als neuer Fitness-Skalierungsparameter vermerkt und die standardisierte Fitness für alle in der virtuellen Population referenzierten Individuen mit Hilfe der Funktion `ApplyDynamicFitnessScaling(...)` neu berechnet (Zeile 31). Da nun jedoch die adjustierten und die normalisierten Fitnesswerte der Individuen nicht mehr stimmen, müssen diese ebenfalls neu berechnet werden: Wenn eine spezielle Fitnessadjustierung gewünscht ist (Zeile 33), dann werden die adjustierten Fitnesswerte mit Hilfe der Funktion `AdjustFitnessValues()` neu berechnet (Zeile 34), ansonsten mit Hilfe der Funktion `Population::InvertFitnessValues()` (Zeile 36).

Die Neuberechnung der normalisierten Fitnesswerte erfolgt in Zeile 38 mit Hilfe der Funktion `Population::NormalizeFitnessValues`.

6.4.4.2 `SetFitnessAdjustment(double (*) (double))`

Mit Hilfe dieser Funktion kann der Anwender festlegen, ob er eine spezielle Fitnessadjustierung wünscht, und wenn ja, welche Funktion zur Berechnung der adjustierten Fitness herangezogen werden soll. Als Parameter ist, wenn keine spezielle Fitnessadjustierung gewünscht wird, `NULL`, ansonsten ein Funktions-Pointer auf die gewünschte Funktion anzugeben. Bereits ausprogrammiert ist die gebräuchliche Funktion `double One_Divided_By_OnePlusX(double)`.

```
    /*--------------------------------------------------------
      void Farm::SetFitnessAdjustment(double(AdjustmentMethod)(double))

      Setting of fitness adjustment:
 5
      AdjustmentMethod==NULL ... no adjustment
      AdjustmentMethod!=NULL ... pointer to adjustment function
      --------------------------------------------------------
    */
10  void Farm::SetFitnessAdjustment(double (*AdjustmentMethod)(double))
    {
            if(AdjustmentMethod!=FitnessAdjustmentFunction)
```

```
          {
                 FitnessAdjustmentFunction = AdjustmentMethod;
15               if(FitnessAdjustmentFunction!=NULL)
                        AdjustFitnessValues();
                 else
                        Current->InvertFitnessValues();

20               Current->NormalizeFitnessValues();
          }
    }
```

In Zeile 12 wird zunächst geprüft, ob sich der Parameter seit dem letzten Aufruf überhaupt geändert hat. Nur wenn dies zutrifft, wird in Zeile 14 fortgefahren. Hier wird die im Parameter referenzierte Funktion zur nunmehr gültigen Fitness-Adjustierungsfunktion erklärt. Handelt es sich nicht um einen `NULL`-Pointer, dann werden die adjustierten Fitnesswerte aller in der virtuellen Population referenzierten Individuen mit der neuen Adjustierungsfunktion unter Zuhilfenahme von `AdjustFitnessValues()` neu berechnet. Ist aber keine spezielle Fitness-Adjustierung gewünscht, dann werden die adjustierten Fitnesswerte mit Hilfe der Funktion `Population::InvertFitnessValues()` neu berechnet. Da nun die in den Individuen gespeicherten normalisierten Fitnesswerte nicht mehr stimmen, werden auch diese in Zeile 20, unter Zuhilfenahme der Funktion `Population::NormalizeFitnessvalues`, neu berechnet.

6.4.4.3 SetSelection(int)

Mit Hilfe dieser Funktion kann der Anwender die im nächsten Generationsschritt anzuwendende Selektionsmethode auswählen. Als Parameter kann angegeben werden: `PROPORTIONAL` für proportionale Selektion, oder `LINEAR_RANK` für die Linear-Rank-Selection Methode.

```
    /*-------------------------------------------------------
      void Farm::SetSelection(int NewSelectionMethod)

      Setting of new selection methode:
 5    PROPORTIONAL or LINEAR_RANK
      -------------------------------------------------------
    */
    void Farm::SetSelection(int NewSelectionMethod)
    {
10        if(NewSelectionMethod==LINEAR_RANK)
                 Selection = LINEAR_RANK;
          else
          {
                 Selection = PROPORTIONAL;
15
                 if(NewSelectionMethod!=PROPORTIONAL)
                 {
                        cerr << "Warning! Invalid selection method!" << endl;
                        cerr << "Proportional selection instead!" << endl;
20               }
          }
    }
```

In Zeile 10 wird zunächst getestet, ob als Selektionsmethode die Linear-Rank-Methode gewünscht wird. Ist dies der Fall, dann wird dies entsprechend vermerkt (Zeile 11). Ansonsten wird proportionale Selektion angenommen (Zeile 14). In Zeile 16 wird geprüft,

ob dies im Parameter auch tatsächlich angefordert wurde. Trifft dies nicht zu, dann wird in den Zeilen 18 und 19 eine entprechende Fehlermeldung ausgegeben.

6.4.4.4 SetSampling(int, int)

Mit Hilfe dieser Funktion kann der Anwender die im nächsten Generationsschritt anzuwendende Samplingmethode und die Anzahl der zu selektierenden Individuen (somit die Größe der nächsten tatsächlichen Population) auswählen. Der erste Parameter gibt die gewünschte Samplingmethode an. Es können entweder **STOCHASTIC_UNIVERSAL** für die "stochastic universal sampling"-Methode (siehe Kapitel 4.4.2), oder **STOCH_SAMPL_WITH_REPL** für die "stochastic sampling with replacement"-Methode angegeben werden. Der zweite Parameter steht für die Anzahl der zu selektierenden Individuen.

```
    /*------------------------------------------------------
      void Farm::SetSampling(int
      NewSamplingMethod, int NewNoOfMates)

 5    Setting of new sampling method:
      STOCHASTIC_UNIVERSAL or STOCH_SAMPL_WITH_REPL

      NewNoOfMates gives the number of mates, and therefore
      the size of the next actual generation
10    (the virtual population, possibly including subtrees,
      might be larger).
      ------------------------------------------------------
    */
    void Farm::SetSampling(int NewSamplingMethod, int NewNoOfMates)
15  {
            if(NewSamplingMethod==STOCH_SAMPL_WITH_REPL)
                    Sampling = STOCH_SAMPL_WITH_REPL;
            else
            {
20                  Sampling = STOCHASTIC_UNIVERSAL;

                    if(NewSamplingMethode!=STOCHASTIC_UNIVERSAL)
                    {
                            cerr << "Warning! Invalid sampling method!" << endl;
25                          cerr << "Stochastic universal sampling instead." << endl;
                    }
            }

            if(NewNoOfMates<1)
30          {
                    NewNoOfMates = 1;
                    cerr << "Warning! Invalid number of mates (must be >=1)!" << endl;
                    cerr << "Number of mates = 1 instead, " << endl;
            }
35
            NoOfMates = NewNoOfMates;
    }
```

In Zeile 16 wird zunächst getestet, ob als Samplingmethode die "stochastic sampling with replacement"-Methode gewünscht wird. Ist dies der Fall, dann wird dies entsprechend vermerkt (Zeile 17). Ansonsten wird die "stochastic universal sampling"-Methode angenommen (Zeile 20). In Zeile 22 wird geprüft, ob dies im Parameter auch

tatsächlich angefordert wurde. Trifft dies nicht zu, dann wird in den Zeilen 24 und 25 eine entsprechende Fehlermeldung ausgegeben. Schließlich wird in Zeile 29 geprüft, ob der Wert des zweiten Parameters die Untergrenze von 1 unterschreitet. Ist dies der Fall, dann wird eins als Wert für den zweiten Parameter angenommen (Zeile 31), und eine Fehlermeldung ausgegeben (Zeilen 32 und 33). In Zeile 36 wird die Anzahl der zu selektierenden Individuen entsprechend eingestellt.

6.4.4.5 `SetMating(int)`

Mit Hilfe dieser Funktion kann der Anwender das im nächsten Generationsschritt anzuwendende Matingverfahren auswählen. Als Parameter kann angegeben werden: `RANDOM` für Random-Mating, oder `RANDOM_PERMUTATION` für Random-Permutation-Mating.

```
    /*-------------------------------------------------------
      void Farm::SetMating(int NewMatingMethod)

      Setting odf new mating method:
 5    RANDOM or RANDOM_PERMUTATION.
      -------------------------------------------------------
    */
    void Farm::SetMating(int NewMatingMethod)
    {
10          if(NewMatingMethod==RANDOM)
                    Mating = RANDOM;
            else
            {
                    Mating = RANDOM_PERMUTATION;
15
                    if(NewMatingMethod!=RANDOM_PERMUTATION)
                    {
                            cerr << "Warning! Invalid mating method!" << endl;
                            cerr << "Random permutation mating instead." << endl;
20                  }
            }
    }
```

In Zeile 10 wird getestet, ob als Matingverfahren das Random-Mating gewünscht wird. Ist dies der Fall, dann wird dies entsprechend vermerkt (Zeile 11). Ansonsten wird Random-Permutation-Mating angenommen (Zeile 14). In Zeile 16 wird geprüft, ob dies im Parameter auch tatsächlich angefordert wurde. Trifft dies nicht zu, dann wird in den Zeilen 18 und 19 eine entprechende Fehlermeldung ausgegeben.

6.4.4.6 `SetSelectionHeuristic(double (*) (double), int)`

Mit Hilfe dieser Funktion kann der Anwender festlegen, ob bei Kreuzung und Mutation eine spezielle Selektionsheuristik zur Auswahl von Teilbäumen erwünscht ist, und auf welche Werte sich diese Heursitik beziehen soll. Als erster Parameter ist, wenn keine spezielle Selektionsheuristik gewünscht wird, `NULL`, ansonsten ein Funktions-Pointer auf die gewünschte Funktion anzugeben. Bereits ausprogrammiert sind die Funktionen `double pow2(double)`, `double pow3(double)` und `double pow4(double)`. Der zweite Parameter kann die Werte `TREESIZE` oder

SEARCHSPACESIZE annehmen, je nachdem, ob sich die im ersten Parameter angegebene Funktion auf die Ableitungstiefe des Teilbaumes, oder auf die Suchraumgröße des Symbols des Startknotens beziehen soll. Beide Parameter sind optional. Der Defaultwert für den ersten Parameter ist NULL, für den zweiten Parameter TREESIZE.

```
    /*--------------------------------------------------------
      void Farm::SetSelectionHeuristic(double (*NewSelectionHeuristic)(double),
                                       int NewSelectionHeuristicBasis)

 5    NewSelectionHeuristic  ... function pointer to new heuristic
                                 NULL for no heuristic
      NewSelectionHeuristicBasis ... TREESIZE or SEARCHSPACESIZE
      -------------------------------------------------------
    */
10  void Farm::SetSelectionHeuristic(double (*NewSelectionHeuristic)(double),
                                     int NewSelectionHeuristicBasis)
    {
            if(NewSelectionHeuristicBasis!=TREESIZE &&
            NewSelectionHeuristicBasis!=SEARCHSPACESIZE)
15          {
                    cerr << "Warning! Invalid selection-heuristic basis!" << endl;
                    cerr << "Treesize as basisi instead." << endl;
                    NewSelectionHeuristicBasis = TREESIZE;
            }
20
            SelectionHeuristic = NewSelectionHeuristic;
            SelectionHeuristicBasis = NewSelectionHeuristicBasis;
    }
```

In den Zeilen 13 und 14 wird zunächst geprüft, ob der im zweiten Parameter angegebene Wert gültig ist. Sollte dies nicht der Fall sein, dann wird in den Zeilen 16 und 17 eine entsprechende Fehlermeldung ausgegeben, und angenommen, daß sich die im ersten Parameter referenzierte Selektionsheuristik auf die Ableitungstiefe der Teilbäume bezieht (Zeile 18). In den Zeilen 21 und 22 werden die Selektionsheuristik und die Basis der Selektionsheuristik den Angaben gemäß eingestellt.

6.4.4.7 SetCrossOver(int, float)

Mit Hilfe dieser Funktion kann der Anwender die Anzahl der Kreuzungspunkte und die Wahrscheinlichkeit, mit der der Kreuzungsoperator angewendet werden soll, angeben. Die Anzahl der Kreuzungspunkte ist im ersten Parameter anzugeben, die Wahrscheinlichkeit im zweiten Parameter.

```
    /*--------------------------------------------------------
      void Farm::SetCrossOver(int NewNoOfPoints,
                              float NewProbability)

 5    Setting of new cross over method:
      NewNoOfPoints ... new number of cross-over points
      NewProbability ... new cross-over-probability
      --------------------------------------------------------
    */
10  void Farm::SetCrossOver(int NewNoOfPoints,
                            float NewProbability)
    {
```

```
            if(NewNoOfPoints<1)
            {
15                  cerr << "Warning! Invalid number of cross-over points ";
                    cerr << "(must be >=1!" << endl;
                    cerr << "One cross-over point instead." << endl;
                    NewNoOfPoints = 1;
            }
20
            if(NewProbability<0.0)
            {
                    cerr << "Warning! Invalid cross-over probability ";
                    cerr << "(must be >=0)!" << endl;
25                  cerr << "Zero cross-over probability instead." << endl;
                    NewProbability = 0;
            }

            if(NewProbability>1.0)
30          {
                    cerr << "Warning! Invalid cross-over probability ";
                    cerr << "(must be <=1)!" << endl;
                    cerr << "Cross-over probability = 1 instead." << endl;
                    NewProbability = 1;
35          }

            NoOfCrossOverPoints = NewNoOfPoints;
            CrossOverProbability = NewProbability;
    }
```

In Zeile 13 wird zunächst geprüft, ob der erste, die Anzahl der Kreuzungspunkte angebende Parameter die Untergrenze von eins unterschreitet. Ist dies der Fall, dann wird in den Zeilen 15-17 eine Warnung ausgegeben, und eine Ein-Punkt-Kreuzung (*One-Point-Crossover*) angenommen (Zeile 18). In Zeile 21 wird getestet, ob der zweite, die Wahrscheinlichkeit angebende Parameter, die Untergrenze von null unterschreitet. Ist dies der Fall, dann wird in den Zeilen 23-25 eine Warnung ausgegeben, und als Wahrscheinlichkeit für die Anwendung des Kreuzungsoperators null angenommen (Zeile 26). In Zeile 29 wird sodann noch getestet, ob der zweite Parameter die Obergrenze von eins überschreitet. Ist dies der Fall, dann wird in den Zeilen 31-33 eine Warnung ausgegeben, und als Wahrscheinlichkeit für die Anwendung des Kreuzungsoperators eins angenommen. In den Zeilen 37 und 38 werden die neuen Parameter wirksam gemacht.

6.4.4.8 SetMutation(float)

Mit Hilfe des Parameters dieser Funktion kann der Anwender angeben, mit welcher Wahrscheinlichkeit der Mutationsoperator zur Anwendung kommen soll.

```
    /*-------------------------------------------------------
      void Farm::SetMutation(float NewProbability)

      Setting of new mutation method:
 5    NewProbability ... new mutation probability
      -------------------------------------------------------
    */
    void Farm::SetMutation(float NewProbability)
    {
10          if(NewProbability<0.0)
            {
```

```
                    cerr << "Warning! Invalid cross-over probability ";
                    cerr << "(must be >=0)!" << endl;
                    cerr << "Zero cross-over probability instead." << endl;
15                  NewProbability = 0;
            }

            if(NewProbability>1.0)
            {
20                  cerr << "Warning! Invalid cross-over probability ";
                    cerr << "(must be <=1)!" << endl;
                    cerr << "Cross-over probability = 1 instead." << endl;
                    NewProbability = 1;
            }
25
            MutationProbability = NewProbability;
    }
```

In Zeile 10 wird getestet, ob der die Wahrscheinlichkeit angebende Parameter die Untergrenze von null unterschreitet. Ist dies der Fall, dann wird in den Zeilen 12-14 eine Warnung ausgegeben, und als Wahrscheinlichkeit für die Anwendung des Mutations-Operators null angenommen (Zeile 15). In Zeile 18 wird sodann getestet, ob der Parameter die Obergrenze von eins überschreitet. Ist dies der Fall, dann wird in den Zeilen 20-22 eine Warnung ausgegeben, und als Wahrscheinlichkeit für die Anwendung des Mutations-Operators eins angenommen. In der Zeile 26 wird der neue Parameter wirksam gemacht.

6.4.4.9 `SetSaveBest(int)`

Mit Hilfe des Parameters dieser Funktion kann der Anwender angeben, ob eine Elitismusstrategie mit Überleben des besten Individuums erwünscht ist oder nicht.

```
    /*-------------------------------------------------------
      void Farm::SetSaveBest(int NewSaveBestMode)

      Save best inndividual: TRUE or FALSE)
 5    -------------------------------------------------------
    */
    void Farm::SetSaveBest(int NewSaveBestMode)
    {
            SaveBest = NewSaveBestMode;
10  }
```

Der Wert des Parameters wird in Zeile 9 dem Flag `SaveBest` zugewiesen, und dadurch wirksam gemacht.

6.4.4.10 `SetSmallestSubTreeOfVirtualPop(unsigned int)`

Mit Hilfe dieser Funktion kann der Anwender die Ableitungstiefe des kleinsten in die virtuelle Population aufzunehmenden Teilbaumes angeben. Ein Spezialfall ist die Angabe von null als Parameter: In diesem Fall werden nur die Mitglieder der tatsächlichen Population in die virtuelle Population aufgenommen, jedoch keine Teilbäume.

```
    /*-------------------------------------------------------
      void Farm::SetSmallestSubTreeOfVirtualPop(unsigned int NewSmallestTreeSize)

      Setting of smallest subtree to be included to
 5    virtual population.
      -------------------------------------------------------
    */
    void Farm::SetSmallestSubTreeOfVirtualPop(unsigned int NewSmallestTreeSize)
    {
10          if(SmallestSubTreeOfVirtualPop!=NewSmallestTreeSize)
            {
                    SmallestSubTreeOfVirtualPop = NewSmallestTreeSize;

                    if(SmallestSubTreeOfVirtualPop==0)
15                          Current->GenerateVirtualPop();
                    else
                            Current->
                            GenerateVirtualPopInclSubTrees(SmallestSubTreeOfVirtualPop);

20                  ComputeFitnessValues();
            }
    }
```

In Zeile 10 wird zunächst geprüft, ob sich der Wert des Parameters seit dem letzten Aufruf überhaupt verändert hat. Nur wenn dies der Fall ist, wird in Zeile 12 fortgesetzt. Hier wird der Wert des Parameters zur ab nun gültigen kleinsten Teilbaumgröße erklärt. In Zeile 14 wird geprüft, ob der neue Wert null beträgt (Spezialfall). Trifft dies zu, dann wird die virtuelle Population unter Zuhilfenahme der Funktion `Population::GenerateVirtualPop()` neu erstellt (Zeile 15). Andernfalls wird die virtuelle Population in Zeile 17 mit Hilfe der Funktion `Population::GenerateVirtualPopInclSubTrees(int)` neu erstellt. In Zeile 20 wird durch Aufruf der Funktion `ComputeFitnessValues` dafür gesorgt, daß alle in der neu erstellten virtuellen Population referenzierten Individuen Fitnesswerte zugewiesen bekommen.

6.4.4.11 SetMaxDerivDepth(int)

Mit Hilfe dieser Funktion kann der Anwender die maximal zulässige Ableitungstiefe bestimmen.

```
    /*-------------------------------------------------------
      void Farm::SetMaxDerivDepth(int mdd)

      Setting of maximum derivation depth
 5    (must be >=1)
      -------------------------------------------------------
    */
    void Farm::SetMaxDerivDepth(int mdd)
    {
10          mdd = ( (mdd<1) ? 1 : mdd);
            MaxDerivDepth = mdd;
    }
```

In Zeile 10 wird zunächst geprüft, ob unzulässigerweise eine maximale Ableitungstiefe kleiner eins gefordert wird. Sollte dies zutreffen, dann wird die maximal zulässige

Ableitungstiefe mit eins eingestellt. Die neue maximal zulässige Ableitungstiefe wird daraufhin in Zeile 11 zur ab nun wirksamen gemacht.

6.4.5 Fitnessberechnungen

6.4.5.1 StandardizeFitnessValues(int)

Diese Funktion errechnet und speichert die standardisierte Fitness für jedes in der virtuellen Population referenzierte Individuum. Der Parameter gibt dabei an, ob die Rohfitness zu maximieren oder zu minimieren ist. Je nachden ist `MAXIMIZE` oder `MINIMIZE` zu übergeben. Das beste in der virtuellen Population referenzierte Individuum erhält einen standardisierten Fitnesswert von null zugewiesen.

```
    /*------------------------------------------------------
      void Farm::StandardizeFitnessValues(int goal)

      Standardizing the fitness values of every tree of
 5    the current virtual population  in a way that better
      trees get a lower fitness value and the best fitness
      value in the population is 0.

      The variable goal determines, wheter the raw
10    fitness is to be maximized or minimized.
      ------------------------------------------------------
    */
    void Farm::StandardizeFitnessValues(int goal)
    {
15          if(goal!=MAXIMIZE && goal!=MINIMIZE)
            {
                    cerr << "Warning! No valid goal (maximization or mimimization) defined!";
                    cerr << endl << "I choose MINIMIZATION as goal." << endl;
                    goal = MINIMIZE;
20          }

            Current->ComputeBestAndWorstIndividual(goal);
            Current->StandardizeWith(Current->BestIndividual->RawFitness, goal);
    }
```

In Zeile 15 wird zunächst geprüft, ob der Parameter ein gültiges Optimierungsziel beinhaltet. Sollte dies nicht der Fall sein, wird in den Zeilen 17 und 18 eine entsprechende Warnung ausgegeben, und Rohfitness-Minimierung als Ziel angenommen (Zeile 19). In Zeile 22 wird dann mit Hilfe der Funktion `Population::ComputeBestAndWorstIndividual(int)` das beste und das schlechteste Individuum der virtuellen Population bestimmt. Anschließend wird in Zeile 23 mit Hilfe der Funktion `Population::StandardizeWith(float,int)` für alle in der virtuellen Population referenzierten Individuen die standardisierte Fitness ausgerechnet und gespeichert. Der erste Parameter dieser Funktion gibt den Rohfitnesswert an, der gerade eine standardisierte Fitness von null ergeben soll. Der zweite Parameter definiert das Optimierungsziel. Im speziellen Fall wird für den ersten Parameter der Rohfitnesswert des besten Individuums dieser einen Population eingesetzt (Zeile 23).

6.4.5.2 ApplyDynamicFitnessScaling(unsigned int, int)

Diese Funktion errechnet und speichert die standardisierte Fitness für jedes in der aktuellen virtuellen Population referenzierte Individuum. Der erste Parameter gibt an, wieviele Vorgängergenerationen dabei zu berücksichtigen sind. Das beste Individuum in dieser und der angegebenen Anzahl von Vorgängergenerationen entspricht gerade einem standardisierten Fitnesswert von null. Der zweite Parameter gibt an, ob die Rohfitness zu maximieren oder zu minimieren ist. Je nachdem ist `MAXIMIZE` oder `MINIMIZE` zu übergeben.

```
    /*-------------------------------------------------------
      void Farm::ApplyDynamicFitnessScaling(unsigned int n, int goal)

      Standardizing the fitness values of every tree in
 5      a way that better trees get a lower fitness value
      and the best fitness value in the last n
      populations is 0.

      n==0 ... just this population, (as StandardizeFitnessValues)
10    n==1 ... this and last population
      n==2 ... this and the last and the one before last
      ...

      The parameter goal determines, wheter the raw
15    fitness ist to be maximized or minimized.
      -------------------------------------------------------
    */
    void Farm::ApplyDynamicFitnessScaling(unsigned int n, int goal)
    {
20          int i;
            int j;
            float BestFitness;

            if(goal!=MAXIMIZE && goal!=MINIMIZE)
25          {
                    cerr << "Warning! No valid goal (maximization or mimimization) defined!";
                    cerr << endl << "I choose MINIMIZATION as goal." << endl;
                    goal = MINIMIZE;
            }
30
            if(n>History)
            {
                    cerr << "Warning! Dynamic fitness scaling over the last " << n;
                    cerr << " populations can not be performed, "<< endl;
35                  cerr << "because only the last ";
                    cerr << History << " populations are stored." << endl;
                    cerr << "I therefore perform a dynamic fitness scaling over the ";
                    cerr << "last " << History << " populations." << endl << endl;
                    n = History;
40          }

            Current->ComputeBestAndWorstIndividual(goal);

            j = PopulationIndex;
45          BestFitness = Current->BestIndividual->RawFitness;
            i = 0;

            while(i<n)
            {
50                  if(goal==MINIMIZE)
                    {
                            if(ThePopulation[j].BestIndividual->RawFitness<BestFitness)
                                    BestFitness =
```

```
                                        ThePopulation[j].BestIndividual->RawFitness;
55                      }
                        else
                        {
                                if(ThePopulation[j].BestIndividual->RawFitness>BestFitness)
                                        BestFitness =
60                                      ThePopulation[j].BestIndividual->RawFitness;
                        }

                        if(!ThePopulation[j].ParentsAvailable)
                                break;
65
                        j = (j==0 ? History : j-1);
                        i++;
                }

70              Current->StandardizeWith(BestFitness, goal);
        }
```

In Zeile 24 wird zunächst geprüft, ob der zweite Parameter ein gültiges Optimierungsziel beinhaltet. Sollte dies nicht der Fall sein, wird in den Zeilen 27 und 28 eine entsprechende Warnung ausgegeben, und Rohfitness-Minimierung als Ziel angenommen (Zeile 29). Sodann wird in Zeile 31 getestet, ob die im ersten Parameter angegebene Anzahl von Vorgängergenerationen größer ist, als die Anzahl der im Speicher zur Verfügung stehenden Vorgängergenerationen. Sollte dies der Fall sein, dann wird in den Zeilen 33-38 eine entsprechende Warnung ausgegeben, und die Anzahl der zu berücksichtigen Vorgängergenerationen auf die Anzahl der im Speicher zur Verfügung stehenden Vorgängergenerationen beschränkt. In Zeile 42 wird dann zunächst mit Hilfe der Funktion `Population::ComputeBestAndWorstIndividual(int)` das beste und das schlechteste Individuum der aktuellen Population bestimmt. In Zeile 44 wird dann der lokalen Zählvariablen `j` der Index der aktuellen Population zugewiesen und die Fitness des besten Individuums der aktuellen Population vorerst als beste Fitness aller zu berücksichtigenden Generationen angenommen (Zeile 45). In der in Zeile 48 beginnenden Schleife werden sodann, beginnend von der aktuellen, die aktuelle und die `n` Vorgängergenerationen durchlaufen. In jeder Population wird, wenn es sich um eine Minimierungsaufgabe handelt (Zeile 50), verglichen, ob das beste Individuum der betrachteten Population eine kleinere Rohfitness aufweist als die beste bisher gefundene Fitness (Zeile 52), oder, wenn es sich um eine Maximierungsaufgabe handelt (Zeile 56), ob das beste Individuum der betrachteten Population eine größere Rohfitness aufweist als die beste bisher gefundene Fitness (Zeile 58). Sollte dies zutreffen, wird die beste bisher gefundene Fitness auf den Wert der Fitness des betrachteten Individuums aktualisiert (Zeilen 53-54 und 59-60). In Zeile 63 wird festgestellt, ob zur betrachteten Population eine Elterngeneration verfügbar ist. Sollte dies nicht der Fall sein, dann wird die Schleife vorzeitig abgebrochen (Zeile 64). In Zeile 66 wird der Index der im nächsten Schleifendurchlauf zu betrachtenden Vorgängergeneration berechnet. Nach Durchlaufen der Schleife wird für alle in der aktuellen virtuellen Population vermerkten Individuen mit Hilfe der Funktion `Population::StandardizeWith(float, int)` die standardisierte Fitness berechnet und gespeichert, wobei als erster Parameter dieser Funktion die beste gefundene Rohfitness angegeben wird.

6.4.5.3 AdjustFitnessValues()

Diese Funktion errechnet und speichert an Hand der eingestellten Adjustierungsfunktion einen adjustierten Fitnesswert für jedes in der virtuellen Population vermerkte Individuum.

```
   /*-------------------------------------------------------
     void Farm::AdjustFitnessValues()

     Computing the AdjustedFitness for every Tree
 5   in the current virtual population using

     double Farm::(*SelectionHeuristic) (double)

     which has to be assigned before.
10
     Alternative to Current->InvertFitnessValues()
     ------------------------------------------------------
   */
   void Farm::AdjustFitnessValues()
15 {
           int i;

           Current->AdjustedFitnessSum = 0.0;

20         for(i=0;i<Current->VirtualPopSize;i++)
           {
                   Current->Virtual[i]->AdjustedFitness =
                   FitnessAdjustmentFunction(Current->Virtual[i]->StandardizedFitness);

25                 Current->AdjustedFitnessSum+=
                   Current->Virtual[i]->AdjustedFitness;
           }
   }
```

In Zeile 18 wird zunächst die Summe der adjustierten Fitnesswerte der aktuellen Population mit null initialisiert. In der in Zeile 20 beginnenden Schleife werden dann alle in der virtuellen Population vermerkten Individuen durchlaufen. Jedem Individuum wird unter Zuhilfenahme der eingestellten Adjustierungsfunktion eine adjustierte Fitness zugewiesen (Zeilen 22 und 23) und die Summe der adjustierten Fitnesswerte wird um die errechnete adjustierte Fitness des betrachteten Individuums erhöht (Zeilen 25 und 26).

6.4.5.4 ComputeFitnessValues()

Diese Funktion ruft die Funktionen zur Fitnessberechnung unter Berücksichtigung der eingestellten Optionen in der richtigen Reihenfolge der Reihe nach auf.

```
   /*-------------------------------------------------------
     int Farm::ComputeFitnessValues()

     Computing the FitnessValues.
 5   -------------------------------------------------------
   */
   int Farm::ComputeFitnessValues()
```

```
    {
            if(Current==NULL)
10                  return(FALSE);

            if(Current->ActualPopSize==0)
                    return(FALSE);

15          Current->ApplyRawFitness();

            ApplyDynamicFitnessScaling(DynamicFitnessScalingParameter,Goal);

            if(FitnessAdjustmentFunction!=NULL)
20                  AdjustFitnessValues();
            else
                    Current->InvertFitnessValues();

            Current->NormalizeFitnessValues();
25
            return(TRUE);
    }
```

In Zeile 9 wird zunächst geprüft, ob es überhaupt eine aktuelle Population gibt. Ist dies nicht der Fall, wird die Funktion, unter Rückgabe von `FALSE`, vorzeitig abgebrochen (Zeile 10). Gibt es eine aktuelle Population, so wird in Zeile 12 geprüft, ob sie Individuen beinhaltet. Ist dies nicht der Fall, wird die Funktion ebenfalls, unter Rückgabe von `FALSE`, vorzeitig abgebrochen (Zeile 13). Danach werden der Reihe nach aufgerufen:

1. Rohfitnessberechnung (Zeile 15)
2. Fitnessskalierung (Zeile 17)
3. Fitenssadjustierung (Zeilen 19-22)
4. Fitnessnormalisierung (Zeile 24)

Anschließend wird die Funktion, unter Rückgabe von `TRUE`, beendet.

6.4.6 Genetische Operatoren

6.4.6.1 `NPointCrossOver(unsigned int, float, int)`

Diese Funktion erzeugt aus den in der Mating-List referenzierten Individuen die nächste Generation, indem aus jedem Paar mittels Kreuzung ein neues Individuum erzeugt wird. Der erste Parameter gibt dabei die Anzahl der Kreuzungspunkte an. Ein Zwei-Punkt-Crossover wird z.B. ausgeführt, indem nach dem gewöhnlichen Ein- Punkt-Crossover die Kreuzung ein zweites mal durchgeführt und statt dem ersten Partner der erzeugte Ableitungsbaum eingesetzt wird. Der zweite Parameter dieser Funktion gibt an, mit welcher Wahrscheinlichkeit die Kreuzung durchgeführt werden soll. Wird für ein bestimmtes Individuum keine Kreuzung durchgeführt, dann wird der erste Partner des betrachteten Paares der Mating-List ohne Veränderung in die neue

Population kopiert. Der dritte Parameter schließlich bestimmt die Art der Zufallsauswahl der Unterbäume. Wird hier `HEURISTICSELECTION` eingetragen, so kommt die ausgewählte Selektionsheuristik zur Auswahl von Teilbäumen zur Anwendung. Wird `DEFAULTSELECTION` eingetragen, dann kommt keine Heuristik zur Anwendung.

```
     /*----------------------------------------------------
       void Farm::NPointCrossOver(unsigned int NoOfPoints,
                                  float Probability,
                                  int SelectionMethod)
 5
       Creating a new population consisting of indivduals
       breeded out of the pairs in the MateList.

       The parameter Probability gives the probability for
10     actually performing a crossover on the single individual,

       SelectionMethod may be DEFAULTSELECTION or
       HEURISTICSELECTION and is influencing the random
       selection of subtrees.
15     ----------------------------------------------------
     */
     void Farm::NPointCrossOver(unsigned int NoOfPoints,
                                float Probability,
                                int SelectionMethod)
20   {
             int i;
             int j;
             int NewPopulationIndex;
             RootNode* NewPopulation;
25           RootNode TempTree;
             CrossOverParameters SelectedCrossOverParameters;
             StartNode* Tree1;
             static Uniran RandomNumber(RANDNPCROSSSEED);

30           NewPopulationIndex =
             (PopulationIndex==History ? 0 : PopulationIndex+1);


             if(
35           ThePopulation[NewPopulationIndex].ActualPopArraySize<
             Current->MatingListNoOfEntries)
                     ThePopulation[NewPopulationIndex].
                     ReSizeActualPop(Current->MatingListNoOfEntries);

40           NewPopulation = ThePopulation[NewPopulationIndex].Actual;

             for(j=0;j<Current->MatingListNoOfEntries;j++)
             {
                     if(RandomNumber.rand()<=Probability)
45                   {
                             for(i=0;i<NoOfPoints;i++)
                             {
                                     if(i==0)
                                             Tree1 = Current->MatingList[j].Partner1;
50                                   else
                                     {
                                             if(NoOfPoints%2)
                                             {
                                                     if(i%2)
55                                                           Tree1 =
                                                             NewPopulation+j;
                                                     else
                                                             Tree1 = &TempTree;
```

```
 60                                          }
                                             else
                                             {
                                                     if(i%2)
                                                             Tree1 = &TempTree;
 65                                                  else
                                                             Tree1 =
                                                             NewPopulation+j;
                                             }
                                     }
 70
                                     if(SelectionMethod == DEFAULTSELECTION)
                                             SelectedCrossOverParameters =
                                             SelectCrossOverPoints(Tree1,
                                             Current->MatingList[j].Partner2);
 75                                  else
                                             SelectedCrossOverParameters =
                                             SelectCrossOverPointsUsingHeuristic(
                                             Tree1,
                                             Current->MatingList[j].Partner2);
 80


                                    if(SelectedCrossOverParameters.FirstTree!=NULL &&
                                        SelectedCrossOverParameters.CrossOverPoint!=NULL &&
                                        SelectedCrossOverParameters.SecondSubTree!=NULL)
 85                                 {
                                             if(NoOfPoints%2)
                                             {
                                                     if(i%2)
                                                             TempTree.CrossOver(
 90                                                          SelectedCrossOverParameters);
                                                     else
                                                             NewPopulation[j].CrossOver(
                                                             SelectedCrossOverParameters);
                                             }
 95                                          else
                                             {
                                                     if(i%2)
                                                             NewPopulation[j].CrossOver(
                                                             SelectedCrossOverParameters);
100                                                  else
                                                             TempTree.CrossOver(
                                                             SelectedCrossOverParameters);
                                             }

105                                          NewPopulation[j].Parent1 =
                                             Current->MatingList[j].Partner1;

                                             NewPopulation[j].Parent2 =
                                             Current->MatingList[j].Partner2;
110                                  }
                                     else
                                             NewPopulation[j] =
                                             *(Current->MatingList[j].Partner1);
                             }
115                  }
                     else
                             NewPopulation[j] =
                             *(Current->MatingList[j].Partner1);

120                  NewPopulation[j].Home = ThePopulation+NewPopulationIndex;
             }

             ThePopulation[NewPopulationIndex].ActualPopSize =
             Current->MatingListNoOfEntries;
125
             PopulationIndex = NewPopulationIndex;
```

```
            Current = ThePopulation + PopulationIndex;
            Current->Actual = NewPopulation;
            Current->ParentsAvailable = YES;
130
            ThePopulation[(PopulationIndex==History ? 0 : PopulationIndex+1)].
            ParentsAvailable = NO;
    }
```

In den Zeilen 30 und 31 wird zunächst der Index der neu zu erzeugenden Population ausgerechnet. Sodann wird in den Zeilen 34-36 geprüft, ob ausreichend Platz für alle zu erzeugenden Individuen vorhanden ist. Sollte dies nicht der Fall sein, wird die Population mit Hilfe der Funktion `Population::ReSizeActualPop(int)` redimensioniert (Zeilen 37 und 38). Die in Zeile 42 beginnende Schleife wird nun so oft durchlaufen, als es Einträge in der Mating-List gibt. Bei jedem Eintrag wird in Zeile 44 geprüft, ob eine erzeugte Zufallszahl zwischen null und eins kleiner ist, als die im zweiten Parameter angegebene Wahrscheinlichkeit. Nur wenn dies zutrifft, wird in Zeile 46 fortgefahren und die Kreuzung durchgeführt. Ansonsten wird in den Zeilen 116 und 117 lediglich der erste Partner des betrachteten Mating-List-Eintrags mit Hilfe des Zuweisungsoperators in die neue Population hineinkopiert.

Ist jedoch eine Kreuzung durchzuführen, dann wird die in Zeile 47 beginnende Schleife so oft durchlaufen, als im ersten Parameter angegeben wurde. Die Kreuzung wird in weiterer Folge stets zwischen dem in der lokalen Pointer-Variablen `Tree1` referenzierten Individuum und dem zweiten Partner des betrachteten Eintrags der Mating-List durchgeführt. In den Zeilen 48 und 49 wird festgelegt, daß beim ersten Durchlauf der Pointer `Tree1` auf den ersten Partner des betrachteten Eintrags der Mating-List zeigen soll.

Bei jedem weiteren Kreuzungspunkt wird in den Zeilen 52-67 der Pointer `Tree1` auf das im letzen Durchlauf erzeugte Individuum gesetzt. Und dieses wird alternierend in der entsprechenden Stelle der neuen Population (Zeilen 55-56 und 65-66) und in der lokalen Variablen `TempTree` (Zeilen 58 und 64) erwartet.

Wo das zuletzt neu erzeugte Individuum in jedem Durchlauf nun erwartet wird, hängt von der Anzahl der Kreuzungspunkte (Zeile 52), sowie davon ab, ob es sich um einen geraden (Zeilen 54 und 63) oder ungeraden Schleifendurchlauf handelt (Zeilen 57 und 64). In jedem Fall muß nämlich (wenn es mehrere Kreuzungspunkte gibt) beim letzten Durchlauf das zuletzt neu erzeugte Individuum in der lokalen Variablen `TempTree` stehen, damit im nun folgenden, letzten Durchlauf das endgültig neue Individuum in der neuen Population erzeugt werden kann.

In Zeile 71 wird dann in weiterer Folge getestet, welche Art der Selektion der Kreuzungspunkte erwünscht ist. Die Standardselektion wird mit Hilfe der Funktion `SelectCrossOverPoints(...)` durchgeführt (Zeilen 72-74), die Selektion unter Zuhilfenahme der eingestellten Selektionsheuristik mittels der Funktion `SelectCrossOverPointsUsingHeuristic(...)` (Zeilen 76-79).

In den Zeilen 82-84 wird getestet, ob ein Tupel korrekter Kreuzungsparameter gefunden werden konnte. Ist dies nicht der Fall, dann wird in Zeile 112 fortgesetzt, wo der erste Partner des betrachteten Mating-List-Eintrags mit Hilfe des Zuweisungsoperators in die neue Population hineinkopiert wird. Sind die Kreuzungsparameter jedoch korrekt, dann wird in den Zeilen 86-103 die Kreuzung durchgeführt.

Abhängig von der Anzahl der Kreuzungspunkte (Zeile 86) und davon, ob es sich um einen geraden oder einen ungeraden Schleifendurchlauf handelt (Zeilen 88 und 97), wird das Individuum in der lokalen Variablen `TempTree` (Zeilen 90-91 und 100-101), oder in der entsprechenden Stelle der neuen Population neu erzeugt (Zeilen 92-93 und 98-99). Auf jeden Fall muß nämlich beim letzten Durchlauf das endgültige Individuum an der entsprechenden Stelle der neuen Population erzeugt werden.

In den Zeilen 105-109 werden zuletzt für jedes neu erzeugte Individuum die richtigen Elternbezüge gespeichert. Nachdem die neue Population erzeugt wurde, wird noch die Populationsgröße (Zeilen 123 und 124), der Populationsindex (Zeile 126), der Pointer auf die aktuelle Population (Zeilen 127 und 128) sowie das `ParentsAvailable`-Flag (Zeile 129) eingestellt. Das `ParentsAvailable`-Flag der ältesten im Speicher befindlichen Generation wird auf `NO` eingestellt, da die Elterngeneration dieser Population mit der neu erzeugten Population überschrieben wurde.

6.4.6.2 `SelectCrossOverPoints(StartNode*, StartNode*)`

Diese Funktion wählt aus den durch den ersten und den zweiten Parameter referenzierten Bäumen zwei Teilbäume aus, deren Wurzelknoten das gleiche Symbol beinhalten, und die daher ausgetauscht werden können. Es wird darauf geachtet, daß der aus dem zweiten Baum ausgewählte Teilbaum mit dem im ersten Baum ausgewählten Teilbaum ausgetauscht werden kann, ohne daß der erste Baum die maximal zulässige Ableitungstiefe überschreitet.

```
    /*-------------------------------------------------------
      CrossOverParameters Farm::SelectCrossOverPoints(StartNode* TheFirstTree,
                                                StartNode* TheSecondTree)

 5    Random-selection of adequate cross-over-points in
      both trees.
      -------------------------------------------------------
    */

10  CrossOverParameters Farm::SelectCrossOverPoints(StartNode* TheFirstTree,
                                              StartNode* TheSecondTree)

    {
            CrossOverParameters Selected;
15          int NoOfSelectedSubTree1;
            int NoOfSelectedSubTree2;
            static Uniran RandomNumber(SELCROSSSEED);
            int Trial;

20          Selected.FirstTree = TheFirstTree;
            Selected.CrossOverPoint = NULL;
```

```
            Selected.SecondSubTree = NULL;

            Trial = 0;
25          while(Trial<8 && Selected.SecondSubTree==NULL)
            {
                    Trial++;

                    NoOfSelectedSubTree1 =
30                  RandomNumber.dice(TheFirstTree->DerivationDepth);

                    if(NoOfSelectedSubTree1<0)
                            break;
                    else
35                  {

                            Selected.CrossOverPoint =
                            TheFirstTree->SubTree(NoOfSelectedSubTree1);

40                          NoOfSelectedSubTree2 =
                            RandomNumber.dice(TheSecondTree->NoOfSubTrees(
                                              Selected.CrossOverPoint->Symbol,
                                              MaxDerivDepth -
                                              TheFirstTree->DerivationDepth +
45                                            Selected.CrossOverPoint->
                                              DerivationDepth));

                            if(NoOfSelectedSubTree2>=0)
                            {
50                                  Selected.SecondSubTree =
                                    TheSecondTree->SubTree(
                                    Selected.CrossOverPoint->Symbol,
                                    NoOfSelectedSubTree2,
                                    MaxDerivDepth -
55                                  TheFirstTree->DerivationDepth +
                                    Selected.CrossOverPoint->
                                    DerivationDepth);
                            }
                    }
60          }

            return(Selected);
    }
```

In den Zeilen 20-22 wird zunächst die lokale Variable `Selected`, die die ausgewählten Kreuzungspunkte aufnehmen soll, initialisiert. Die in Zeile 25 beginnende Schleife wird solange durchlaufen, bis ein geeigneter Kreuzungspunkt in beiden Bäumen gefunden wurde, höchstens jedoch acht mal. In jedem Durchlauf wird zunächst in den Zeilen 29 und 30 mit Hilfe der Zufallsfunktion `dice` ein Baumknoten, der ein Nichtterminalsymbol beinhaltet aus dem im ersten Parameter referenzierten Baum ausgewählt. Die Zufallsfunktion `dice` liefert eine ganzzahlige Zufallszahl zwischen 0 und dem angegebenen Parameter (exklusive). Wird sie mit null als Parameter aufgerufen, so gibt sie minus eins zurück. Daher kann mit dem Vergleich in Zeile 32 getestet werden, ob kein Teilbaum ausgewählt werden konnte, weil es keine Nichtterminalsymbol-Knoten im durch den ersten Parameter referenzierten Baum gibt. Ist dies der Fall, dann wird die Schleife vorzeitig verlassen (Zeile 33). Ansonsten wird der Startknoten des ausgewählten Teilbaums mit Hilfe der Funktion `StartNode::SubTree(unsigned int)` herausgesucht, und als Kreuzungspunkt im ersten Baum festgelegt (Zeilen 37 und 38). In den Zeilen 40-46 wird sodann aus den in Frage kommenden Knoten des zweiten Baumes (das sind alle

Knoten, die das gleiche Symbol beinhalten, und deren Ableitungstiefe nicht zur Überschreitung der maximal zulässigen Ableitungstiefe führen würde), deren Anzahl mit Hilfe der Funktion `TreeNode::NoOfSubTrees(...)` festgestellt wird, unter Verwendung der Funktion `dice(int)` zufällig einer ausgewählt. In Zeile 48 wird getestet, ob ein Teilbaum ausgewählt werden konnte, und falls dies der Fall ist, dann wird dieser ausgewählte Teilbaum mit Hilfe der Funktion `StartNode::SubTree(SymbTabEntry*, unsigned int, unsigned int)` herausgesucht und zum Kreuzungspunkt in dem durch den zweiten Parameter referenzierten Baum erklärt (Zeilen 50-57). In Zeile 62 wird die Funktion, unter Rückgabe der ausgewählten Kreuzungsparameter, beendet.

6.4.6.3 SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*)

Diese Funktion wählt aus den durch den ersten und den zweiten Parameter referenzierten Bäumen zwei Teilbäume aus, deren Wurzelknoten das gleiche Symbol beinhalten, und die daher ausgetauscht werden können. Es wird darauf geachtet, daß der aus dem zweiten Baum ausgewählte Teilbaum mit dem im ersten Baum ausgewählten Teilbaum ausgetauscht werden kann, ohne daß der erste Baum die maximal zulässige Ableitungstiefe überschreitet. Die Auswahlwahrscheinlichkeit wird mit der eingestellten Selektionsheuristik gewichtet.

```
    /*------------------------------------------------------
      CrossOverParameters Farm::SelectCrossOverPointsUsingHeuristic(
                                                StartNode* TheFirstTree,
                                                StartNode* TheSecondTree)
 5
      Random-selection of adequate cross-over points
      by using SelectionHeuristic and SelectionHeuristicBasis.
      ------------------------------------------------------
    */
10
    CrossOverParameters Farm::SelectCrossOverPointsUsingHeuristic(
                                                StartNode* TheFirstTree,
                                                StartNode* TheSecondTree)
    {
15          TreeNode* TheSubTree;
            SuperFloat* LargeRoulette1Slot;
            SuperFloat MaxLargeRoulette1Slot;
            double* SmallRoulette1Slot;
            double SmallRoulette1Sum;
20          SuperFloat* LargeRoulette2Slot;
            SuperFloat MaxLargeRoulette2Slot;
            double* SmallRoulette2Slot;
            double SmallRoulette2Sum;
            CrossOverParameters Selected;
25          int NoOfSelectedSubTree1;
            int NoOfSelectedSubTree2;
            static Uniran RandomNumber(IMPRSELCROSSSEED);
            int Trial;
            double SelectedSlot;
30          int i;
            int n;

            if(L.ArbCard==NULL)
            {
35                  cout << "computing search space size..." << endl;
```

```
                    L.ComputeArbSearchSpace(MaxDerivDepth);
                    cout << "done." << endl;
            }

40          SmallRoulette1Slot = new double[TheFirstTree->DerivationDepth];

            if(SelectionHeuristicBasis == SEARCHSPACESIZE)
            {
                    LargeRoulette1Slot = new SuperFloat[TheFirstTree->DerivationDepth];
45
                    MaxLargeRoulette1Slot = 0;
                    for(i=0;i<TheFirstTree->DerivationDepth;i++)
                    {
                            TheSubTree = TheFirstTree->SubTree(i);
50
                            LargeRoulette1Slot[i] =
                            TheSubTree->Symbol->ArbCard[TheSubTree->DerivationDepth];

                            if(LargeRoulette1Slot[i]>MaxLargeRoulette1Slot)
55                                  MaxLargeRoulette1Slot = LargeRoulette1Slot[i];
                    }

                    SmallRoulette1Sum = 0.0;
                    if(MaxLargeRoulette1Slot.Exp>32)
60                  {
                            for(i=0;i<TheFirstTree->DerivationDepth;i++)
                            {
                                    SmallRoulette1Slot[i] =
                                    SelectionHeuristic(
65                                  LargeRoulette1Slot[i].Mant*
                                    pow(10,LargeRoulette1Slot[i].Exp-
                                    MaxLargeRoulette1Slot.Exp + 32)) +
                                    SmallRoulette1Sum;

70                                  SmallRoulette1Sum = SmallRoulette1Slot[i];
                            }
                    }
                    else
                    {
75                          for(i=0;i<TheFirstTree->DerivationDepth;i++)
                            {
                                    SmallRoulette1Slot[i] =
                                    SelectionHeuristic(
                                    LargeRoulette1Slot[i].Mant*
80                                  pow(10,LargeRoulette1Slot[i].Exp)) +
                                    SmallRoulette1Sum;

                                    SmallRoulette1Sum = SmallRoulette1Slot[i];
                            }
85                  }
            }

            else

90          {
                    SmallRoulette1Sum = 0.0;
                    for(i=0;i<TheFirstTree->DerivationDepth;i++)
                    {
                            SmallRoulette1Slot[i] =
95                          TheFirstTree->SubTree(i)->DerivationDepth;

                            SmallRoulette1Slot[i] =
                            SelectionHeuristic(SmallRoulette1Slot[i]) +
                            SmallRoulette1Sum;
100
                            SmallRoulette1Sum = SmallRoulette1Slot[i];
                    }
```

```
            }

105         Selected.FirstTree = TheFirstTree;
            Selected.CrossOverPoint = NULL;
            Selected.SecondSubTree = NULL;

            Trial = 0;
110         while(Trial<8 && Selected.SecondSubTree==NULL)
            {
                    Trial++;

                    SelectedSlot =
115                 SmallRoulette1Sum * RandomNumber.rand();

                    i = TheFirstTree->DerivationDepth-2;
                    while(i>=0 && SmallRoulette1Slot[i]>SelectedSlot)
                            i--;
120
                    if(i<=0 && SmallRoulette1Slot[0]>SelectedSlot)
                            NoOfSelectedSubTree1 = 0;
                    else
                            NoOfSelectedSubTree1 = i+1;
125
                    if(TheFirstTree->DerivationDepth==0)
                            break;
                    else
                    {
130                         Selected.CrossOverPoint =
                            TheFirstTree->SubTree(NoOfSelectedSubTree1);

                            n =
                            TheSecondTree->
135                         NoOfSubTrees(Selected.CrossOverPoint->Symbol,
                            MaxDerivDepth - TheFirstTree->DerivationDepth +
                            Selected.CrossOverPoint->DerivationDepth);

                            if(n>0)
140                         {
                                    SmallRoulette2Slot = new double[n];

                                    if(SelectionHeuristicBasis == SEARCHSPACESIZE)
                                    {
145                                         LargeRoulette2Slot = new SuperFloat[n];
                                            MaxLargeRoulette2Slot = 0;
                                            for(i=0;i<n;i++)
                                            {
                                                    TheSubTree =
150                                                 TheSecondTree->SubTree(
                                                    Selected.CrossOverPoint->Symbol,i,
                                                    MaxDerivDepth -
                                                    TheFirstTree->DerivationDepth +
                                                    Selected.CrossOverPoint->
155                                                 DerivationDepth);

                                                    LargeRoulette2Slot[i] =
                                                    TheSubTree->Symbol->
                                                    ArbCard[TheSubTree->DerivationDepth];
160
                                                    if(LargeRoulette2Slot[i]>
                                                    MaxLargeRoulette2Slot)
                                                            MaxLargeRoulette2Slot =
                                                            LargeRoulette2Slot[i];
165                                         }

                                            SmallRoulette2Sum = 0.0;
                                            if(MaxLargeRoulette2Slot.Exp>32)
                                            {
```

```
170                         for(i=0;i<n;i++)
                            {
                                    SmallRoulette2Slot[i] =
                                    SelectionHeuristic(
                                    LargeRoulette2Slot[i].Mant*
175                                 pow(10,LargeRoulette2Slot[i].Exp-
                                    MaxLargeRoulette2Slot.Exp + 32)) +
                                    SmallRoulette2Sum;

                                    SmallRoulette2Sum =
180                                 SmallRoulette2Slot[i];
                            }
                    }

                    else
185
                    {
                            for(i=0;i<n;i++)
                            {
                                    SmallRoulette2Slot[i] =
190                                 SelectionHeuristic(
                                    LargeRoulette2Slot[i].Mant*
                                    pow(10,LargeRoulette2Slot[i].Exp))
                                    + SmallRoulette2Sum;

195                                 SmallRoulette2Sum =
                                    SmallRoulette2Slot[i];
                            }
                    }
            }
200
            else

            {
                    SmallRoulette2Sum = 0.0;
205                 for(i=0;i<n;i++)
                    {
                            TheSubTree =
                            TheSecondTree->SubTree(
                            Selected.CrossOverPoint->Symbol,i,
210                         MaxDerivDepth -
                            TheFirstTree->DerivationDepth +
                            Selected.CrossOverPoint->
                            DerivationDepth);

215                         SmallRoulette2Slot[i] =
                            TheSubTree->DerivationDepth;

                            SmallRoulette2Slot[i] =
                            SelectionHeuristic(SmallRoulette2Slot[i]) +
220                         SmallRoulette2Sum;

                            SmallRoulette2Sum = SmallRoulette2Slot[i];
                    }
            }
225
            SelectedSlot =
            SmallRoulette2Sum * RandomNumber.rand();

            i = n-2;
230         while(i>=0 && SmallRoulette2Slot[i]>SelectedSlot)
                    i--;

            if(i<=0 && SmallRoulette2Slot[0]>SelectedSlot)
                    NoOfSelectedSubTree2 = 0;
235         else
                    NoOfSelectedSubTree2 = i+1;
```

```
                                    Selected.SecondSubTree =
                                    TheSecondTree->SubTree(
240                                 Selected.CrossOverPoint->Symbol,
                                    NoOfSelectedSubTree2,
                                    MaxDerivDepth -
                                    TheFirstTree->DerivationDepth +
                                    Selected.CrossOverPoint->
245                                 DerivationDepth);

                                    delete [] SmallRoulette2Slot;

                                    if(SelectionHeuristicBasis == SEARCHSPACESIZE)
250                                         delete [] LargeRoulette2Slot;
                            }
                    }
            }

255         delete [] SmallRoulette1Slot;

            if(SelectionHeuristicBasis == SEARCHSPACESIZE)
                    delete [] LargeRoulette1Slot;

260         return(Selected);
    }
```

In Zeile 33 wird zunächst geprüft, ob in der Grammatik dieser Farm bereits eine Suchraumtabelle zur Verfügung steht. Sollte dies nicht der Fall sein, dann wird diese, unter Ausgabe einer Meldung (Zeile 35) bis zur maximal zulässigen Ableitungstiefe neu berechnet. Sodann wird in Zeile 40 ein Array von `double`-Zahlen initialisiert, welches das Rouletterad der folgenden Roulette-Wheel-Auswahl von Teilbäumen des im ersten Parameter referenzierten Baumes abbilden soll (siehe dazu auch Abbildung 29). In Zeile 42 wird getestet, ob als Basis für die Selektionsheuristik die Suchraumgrößen der Knotensymbole oder die Ableitungstiefen der Teilbäume herangezogen werden sollen.

Im ersten Fall wird das Programm in Zeile 44 fortgesetzt. Hier wird ein Array von `SuperFloat`-Zahlen erzeugt, welches die oft sehr großen Suchraumwerte aufzunehmen imstande ist. In der in Zeile 47 beginnenden Schleife wird dieses Array mit den Suchraumgrößen der Knotensymbole aller Teilbäume des im ersten Parameter referenzierten Baumes aufgefüllt (Zeilen 51 und 52) und der Maximalwert ermittelt (Zeilen 54 und 55). Die Unterbäume werde mit Hilfe der Funktion `StartNode::SubTree(unsigned int)` herausgesucht (Zeile 49). In Zeile 59 wird sodann getestet, ob der Exponent des Maximalwerts 10^{32} übersteigt (damit soll ein Überlauf der double-Arithmetik verhindert werden). Ist dies der Fall dann werden in der in Zeile 61 beginnenden Schleife die kumulierten Werte des zuvor aufgefüllten Arrays so standardisiert, daß der größte Exponent 10^{32} beträgt, und diese Werte in die ausgewählte Selektionsheuristik eingesetzt (Zeilen 63-68). Die Ergebnisse werden kumuliert in das eigentliche Roulette-Array hineinkopiert. Zugleich wird in Zeile 70 die Summe aller Werte errechnet. Übersteigt aber der Exponent des Maximalwerts 10^{32} nicht, so werden in der in Zeile 75 beginnenden Schleife die kumulierten Werte des zuvor aufgefüllten Arrays ohne Veränderung in das eigentliche Roulett-Array umkopiert (Zeilen 77-81) und die Summe berechnet. Sollen als Basis für die Selektionsheuristik

aber nicht die Suchraumgrößen der Knotensymbole, sondern die Ableitungstiefen der Teilbäume herangezogen werden, dann werden diese in der in Zeile 92 beginnenden Schleife in das Roulett-Array hineinkopiert (Zeilen 94 und 95), in die Heuristik eingesetzt und aufkumuliert (Zeilen 97-99).

In den Zeilen 105-107 wird die lokale Variable `Selected`, die die ausgewählten Kreuzungsparameter aufnehmen soll, initialisiert. Die in Zeile 110 beginnende Schleife wird solange durchlaufen, bis zwei passende Kreuzungspunkte gefunden werden konnten, höchstens jedoch acht mal. In jedem Durchlauf wird zunächst einmal in den Zeilen 114 und 115 mit Hilfe der Zufallsfunktion `Uniran:rand()` ein Punkt auf dem Roulettrad ausgewählt. In der in Zeile 118 beginnenden Schleife wird dann ermittelt, welchem Slot dieser Punkt entspricht. Beginnend von der obersten Grenze des letzten Sektors wird das Roulett-Array in der in Zeile 118 beginnenden Schleife solange durchlaufen, bis eine obere Sektorgrenze gefunden wurde, die kleiner ist als der gewählte Punkt, oder bis das ganze Roulettrad durchlaufen wurde. In der Zeile 124 wird die Nummer des ausgewählten Sektors gespeichert. Die Abfrage in Zeile 121 ist nötig, da die untere Begrenzung des nullten Sektors (die gleich null ist) nicht im durch `RouletteSlots` referenzierten Array abgebildet wird. Abbildung 29 veranschaulicht diese Tatsache.

In Zeile 126 wird sodann gestestet, ob es gelungen ist, einen Kreuzungspunkt im durch den ersten Parameter referenzierten Baum zu finden. Sollte dies nicht der Fall sein, dann wird die Schleife vorzeitig abgebrochen (Zeile 127). Ansonsten wird der Startknoten des ausgewählte Teilbaums mit Hilfe der Funktion `StartNode::SubTree(unsigned int)` herausgesucht, und als Kreuzungspunkt im ersten Baum festgelegt (Zeilen 130 und 131). In den Zeilen 133-137 wird sodann die Anzahl der in Frage kommenden Knoten des zweiten Baumes (das sind alle Knoten, die das gleiche Symbol beinhalten, und deren Ableitungstiefe nicht zur Überschreitung der maximal zulässigen Ableitungstiefe führen würde) mit Hilfe der Funktion `TreeNode::NoOfSubTrees(...)` festgestellt. Gibt es in Frage kommende Knoten (was in Zeile 139 getestet wird), dann wird in Zeile 141 fortgefahren.

In Zeile 143 wird wieder getestet, ob als Basis für die Selektionsheuristik die Suchraumgrößen der Knotensymbole oder die Ableitungstiefen der Teilbäume herangezogen werden sollen. Im ersten Fall wird das Programm in Zeile 145 fortgesetzt. Hier wird wieder ein Array von `SuperFloat`-Zahlen für den zweiten Baum erzeugt. In der in Zeile 148 beginnenden Schleife wird dieses Array mit den Suchraumgrößen der Knotensymbole aller in Frage kommenden Teilbäume des im zweiten Parameter referenzierten Baumes aufgefüllt (Zeilen 158-160) und der Maximalwert ermittelt (Zeilen 161-164). Die Unterbäume werden mit Hilfe der Funktion `StartNode::SubTree(SymbTabEntry*, unsigned int,unsigned int)` herausgesucht (Zeilen 149-155). In Zeile 169 wird sodann getestet, ob der Exponent dieses Maximalwerts 10^{32} übersteigt. Ist dies der Fall, dann werden in der in Zeile 170 beginnenden Schleife die kumulierten Werte des zuvor aufgefüllten Arrays so standardisiert, daß der größte Exponent 10^{32} beträgt (damit soll ein Überlauf der double-Arithmetik verhindert werden) und diese Werte werden in die ausgewählte Selektionsheuristik eingesetzt (Zeilen 172-177). Die Ergebnisse werden

kumuliert in das eigentliche Roulett-Array hineinkopiert. Zugleich wird in den Zeilen 180-181 die Summe aller Werte errechnet. Übersteigt aber der Exponent des Maximalwerts 10^{32} nicht, so werden in der in Zeile 187 beginnenden Schleife die kumulierten Werte des zuvor aufgefüllten Arrays ohne Veränderung in das eigentliche Roulett-Array des zweiten Baums umkopiert (Zeilen 189-193) und die Summe berechnet (Zeilen 195 und 196). Sollen als Basis für die Selektionsheuristik aber nicht die Suchraumgrößen der Knotensymbole, sondern die Ableitungstiefen der Teilbäume herangezogen werden, dann werden diese in der in Zeile 205 beginnenden Schleife in das Roulett-Array des zweiten Baumes hineinkopiert (Zeilen 215 und 216), in die Heuristik eingesetzt und aufkumuliert (Zeilen 218-220).

In jedem Durchlauf wird sodann in den Zeilen 226 und 227 mit Hilfe der Zufallsfunktion `Uniran:rand()` ein Punkt auf dem Roulettrad ausgewählt. In der in Zeile 230 beginnenden Schleife wird dann, wie schon zuvor für den ersten Baum, ermittelt, welchem Slot dieser Punkt enspricht. Der ausgewählte zweite Teilbaum wird in den Zeilen 238-245 mit Hilfe der Funktion `StartNode::SubTree(...)` herausgesucht und zugewiesen. Die erzeugten Arrays für den im zweiten Parameter referenzierten Baum werden in den Zeilen 247-250, die für den im ersten Parameter referenzierten Baum in den Zeilen 255-258 vernichtet. In Zeile 260 wird die Funktion, unter Rückgabe der ermittelten Kreuzungsparameter, beendet.

6.4.6.4 `Mutate(float, int)`

Diese Funktion geht alle Individuen der tatsächlichen Population durch und läßt sie, mit der im ersten Parameter angegebene Wahrscheinlichkeit, mutieren. Der zweite Parameter kann die Werte `DEFAULTSELECTION` oder `HEURISTICSELECTION` annehmen, und gibt an, ob bei der Zufallsauswahl des zu mutierenden Teilbaumes die eingestellte Selektionsheuristik angewendet werden soll.

```
    /*-------------------------------------------------------
      void Farm::Mutate(float Probability, int SelectionMethod)

      Exchanges with the given Probability a randomly
 5    selected subtree with a randomly created one.

      SelectionMethod may be DEFAULTSELECTION or
      HEURISTICSELECTION and is influencing the
      subtree selection.
10    -------------------------------------------------------
    */
    void Farm::Mutate(float Probability, int SelectionMethod)
    {
            int i;
15          RootNode FirstTree;
            RootNode* RandomTree;
            StartNode* Parent1;
            StartNode* Parent2;
            Population* Home;
20          CrossOverParameters Selected;

            static Uniran Mutation(RANDMUTSEED);
            RandomTree = NULL;
```

```
25          for(i=0;i<Current->ActualPopSize;i++)
            {
                    if(Mutation.rand()<=Probability)
                    {
                            Parent1 = Current->Actual[i].Parent1;
30                          Parent2 = Current->Actual[i].Parent2;
                            Home    = Current->Actual[i].Home;

                            FirstTree = Current->Actual[i];
                            RandomTree = new RootNode(&L,MaxDerivDepth,NULL);
35
                            if(SelectionMethod == DEFAULTSELECTION)
                                    Selected =
                                    SelectCrossOverPoints(&FirstTree, RandomTree);
                            else
40                                  Selected =
                                    SelectCrossOverPointsUsingHeuristic(
                                    &FirstTree, RandomTree);

                            if(Selected.FirstTree!=NULL &&
45                          Selected.CrossOverPoint!=NULL &&
                            Selected.SecondSubTree!=NULL)
                            {
                                    Current->Actual[i].CrossOver(Selected);
                                    Current->Actual[i].Mutated = YES;
50                                  Current->Actual[i].Parent1 = Parent1;
                                    Current->Actual[i].Parent2 = Parent2;
                                    Current->Actual[i].Home = Home;
                            }
                    }
55          }

            if(RandomTree!=NULL)
                    delete RandomTree;

60    }
```

In der in Zeile 25 beginnenden Schleife wird die gesamte tatsächliche Population einmal durchlaufen. Für jedes Individuum in der Population wird in Zeile 27 geprüft, ob eine mit Hilfe der Funktion `Uniran::rand()` erzeugte Zufallszahl zwischen null und eins kleiner ist, als die im ersten Parameter angegebene Wahrscheinlichkeit. Trifft dies zu, dann wird in Zeile 29 fortgefahren. Zunächst werden in den Zeilen 29-30 die Pointer auf die Eltern des Individuums und der Pointer auf die Heimpopulation gesichert. In Zeile 33 wird das gerade betrachtete Individuum als erster Partner für die Kreuzung definiert, und in Zeile 34 ein Zufallsindividuum als zweiter Partner. In Zeile 36 wird an Hand des Wertes des zweiten Parameters getestet, ob keine besondere Selektionsheuristik gewünscht ist. Trifft dies zu, dann werden die Kreuzungspunkte mit Hilfe der Funktion `SelectCrossOverPoints(...)` gesucht (Zeilen 37 und 38). Ansonsten wird die Funktion `SelectCrossOverPointsUsingHeuristic(...)` verwendet (Zeilen 40-42). In den Zeilen 44-46 wird geprüft, ob es gelungen ist, passende Kreuzungspunkte zu finden. Ist dies der Fall, dann wird das zu mutierende Individuum in Zeile 48 mit dem Zufallsindividuum an den selektierten Kreuzungspunkten gekreuzt. In Zeile 49 wird vermerkt, daß dieses Individuum einer Mutation unterzogen wurde, und in den Zeilen 50-52 werden die Pointer auf die Eltern und die Heimpopulation wieder restauriert. In den Zeilen 57 und 58 wird zuletzt das temporäre Zufallsindividuum, so vorhanden, vernichtet.

6.4.6.5 SaveBestIndividual()

Diese Funktion kopiert das beste Individuum der letzten Population an zufälliger Stelle in die aktuelle Population.

```
    /*-------------------------------------------------------
      void Farm::SaveBestIndividual()

      Saving the best individual of the last-generation
 5    population at random position into the current
      population.
      -------------------------------------------------------
    */
    void Farm::SaveBestIndividual()
10  {
            static Uniran RandomPosition(RANDSAVEBESTSEED);
            int LastGenerationIndex;
            int SelectedRandomPosition;

15          LastGenerationIndex =
            (PopulationIndex==0 ? History : PopulationIndex-1);

            if(
            Current->ActualPopSize>0 &&
20          ThePopulation[LastGenerationIndex].ActualPopSize>0)
            {
                    SelectedRandomPosition =
                    RandomPosition.dice(Current->ActualPopSize);

25                  Current->Actual[SelectedRandomPosition] =
                    *(ThePopulation[LastGenerationIndex].BestIndividual);
            }
    }
```

In den Zeilen 15 und 16 wird zunächst der Index der letzten Generation berechnet. In den Zeilen 18-20 wird sodann geprüft, ob es überhaupt eine aktuelle und eine Vorgängergeneration gibt. Nur wenn dies der Fall ist, dann wird zuerst in den Zeilen 22-23 eine zufällige Position in der aktuellen Population gewählt, und dann in den Zeilen 25-26 das beste Individuum der Vorgängergeneration mit Hilfe des Zuweisungsoperators in die aktuelle Population kopiert.

6.4.6.6 NextStep()

Diese Funktion ruft alle notwendigen genetischen Operatoren in der richtigen Reihenfolge unter Berücksichtungung der zuvor gewählten Parameter auf, und erzeugt dadurch die nächste Generation von Individuen.

```
    /*-------------------------------------------------------
      void Farm::NextStep()

      Next evolutionary step.
 5    -------------------------------------------------------
    */
```

```
    void Farm::NextStep()
    {
            if(Selection==PROPORTIONAL)
10                  Current->PropSelect();
            else
                    Current->LinearRankSelect(LinearRankSelectParameter);

            if(Sampling==STOCHASTIC_UNIVERSAL)
15                  Current->StochUnivSampl(NoOfMates);
            else
                    Current->StochSamplWithRepl(NoOfMates);

            if(Mating==RANDOM_PERMUTATION)
20                  Current->RandomPermutationMate();
            else
                    Current->RandomMate();

            if(SelectionHeuristic==NULL)
25                  NPointCrossOver(
                    NoOfCrossOverPoints,CrossOverProbability,DEFAULTSELECTION);
            else
                    NPointCrossOver(
                    NoOfCrossOverPoints,CrossOverProbability,HEURISTICSELECTION);
30
            if(SelectionHeuristic==NULL)
                    Mutate(MutationProbability,DEFAULTSELECTION);
            else
                    Mutate(MutationProbability,HEURISTICSELECTION);
35
            if(SaveBest)
                    SaveBestIndividual();

            if(SmallestSubTreeOfVirtualPop==0)
40                  Current->GenerateVirtualPop();
            else
                    Current->
                    GenerateVirtualPopInclSubTrees(SmallestSubTreeOfVirtualPop);

45          ComputeFitnessValues();
    }
```

Der Reihe nach werden folgende Operatoren aufgerufen:

1. **Selektion**: proportional oder “linear rank“ (Zeilen 9-12),

2. **Sampling**: “stochastic universal sampling“ oder “stochastic sampling wirh replacement“ (Zeilen 14-17),

3. **Mating**: “random mating“ oder “random permutation mating“,

4. **Kreuzung**: n-Punkt, mit oder ohne Selektionsheuristik zur Gewichtung der Zufallsauswahl von Teilbäumen (Zeilen 24-29),

5. **Mutation**: mit oder ohne Selektionsheuristik zur Gewichtung der Zufallsauswahl von Teilbäumen (Zeilen 31-34),

6. **Elitismusoperator**: wenn ausgewählt (Zeilen 36 und 37),

7. **Erzeugung der virtuellen Population**: mit oder ohne Aufnahme von Unterableitungsbäumen (Zeilen 39-43),

8. **Fitnessberechnungen**: mit Hilfe der Funktion `ComputeFitnessaValues()`, die die Fitnesswerte entsprechend der eingestellten Optionen berechnet.

6.4.7 Hilfsfunktionen

6.4.7.1 Print()

Diese Funktion druckt, wie im Beispiel auf Seite 63 gezeigt, die gesamte aktuelle Population, inklusive des virtuellen Anteils, aus.

```
    /*--------------------------------------------------------
      void Farm::Print()

      Printing all Trees of the current virtual population
 5    --------------------------------------------------------
    */
    void Farm::Print()
    {
            int i;
10          int NoOfSubTree;
            int NoOfMainTree;

            cout << "FARM: " << endl << endl;
            cout << "Population size: " << Current->ActualPopSize << endl;
15          cout << "Virtual population size: " << Current->VirtualPopSize << endl;

            cout << endl << "POPULATION:" << endl;

            NoOfSubTree = 0;
20          NoOfMainTree = 0;

            if(Current==NULL)
                    cout << "Empty farm!" << endl;
            else
25          {
                    for(i=0;i<Current->VirtualPopSize;i++)
                    {
                            if(Current->Virtual[i]->Root==Current->Virtual[i])
                            {
30                                  NoOfMainTree++;
                                    cout << endl << "Individual " << NoOfMainTree ;
                                    NoOfSubTree = 0;
                            }
                            else
35                          {
                                    NoOfSubTree++;
                                    cout << endl << "       Subtree " << NoOfSubTree;
                            }
                            cout << " raw-fitness: ";
40                          cout << Current->Virtual[i]->RawFitness;
                            cout << " normalized fitness: ";
                            cout << Current->Virtual[i]->NormalizedFitness;
                            cout << endl;

45                          if(Current->Virtual[i]!=NULL)
                            {
```

```
                        if(NoOfSubTree)
                                cout << "       ";
                        Current->Virtual[i]->Print();
50                  }
                    else
                        cout << "No Individual!";
                    cout << endl;
                }
55          }
    }
```

In den Zeilen 13-15 wird zunächst der Kopf, der die Größe der tatsächlichen und der virtuellen Population beinhaltet, ausgegeben. Die in Zeile 19 initialisierte Zählvariable nimmt die Nummer des Individuums der tatsächlichen Population, die in Zeile 20 initialisierte Zählvariable die Nummer des Teilbaumes innerhalb eines Individuums auf. In Zeile 22 wird geprüft, ob es überhaupt eine Population gibt. Ist dies nicht der Fall, so wird in Zeile 23 die Meldung *Empty Farm!* ausgegeben. Andernfalls wird in der in Zeile 26 beginnenden Schleife die virtuelle Population zur Gänze durchlaufen. In Zeile 28 wird für jedes Individuum getestet, ob es ein Teil eines anderen Individuums ist, oder nicht. Ist es das nicht, dann wird in Zeile 30 der Zähler für Individuen der tatsächlichen Population inkrementiert, und in Zeile 32 der Zähler für Teilbäume auf null gesetzt. In Zeile 31 wird die Nummer des Individuums ausgegeben. Handelt es sich jedoch um einen Teilbaum, dann wird in Zeile 36 der Teilbaum-Zähler inkrementiert, und die Nummer des Teilbaumes eingerückt ausgegeben (Zeile 37). Danach folgt in den Zeilen 39-43 die Ausgabe der Rohfitness und der normalisierten Fitness. In Zeile 45 wird getestet, ob der gerade bearbeitete Eintrag der virtuellen Population gültig ist. Handelt es sich (fehlerhafterweise) um einen `NULL`-Zeiger, dann wird in Zeile 52 die Meldung *No individual!* ausgegeben. Andernfalls wird zuerst geprüft, ob es sich um ein Teilindividuum handelt (Zeile 47), und gegebenenfalls eine Einrückung vorgenommen (Zeile 48). In Zeile 49 wird das Individuum, unter Zuhilfenahme der Funktion `TreeNode::Print()` ausgegeben.

6.4.7.2 `SetDefaultMethods()`

Diese Funktion legt die genetischen Standardoperatoren und Standardparameter fest. Diese kommen zur Anwendung, wenn vom Anwender nichts anderes festgelegt wurde.

```
    /*-------------------------------------------------------
      void Farm::SetDefaultMethods()

      Setting of default-selection, crossover, etc.
 5    -------------------------------------------------------
    */
    void Farm::SetDefaultMethods()
    {
            SelectionHeuristic = NULL;
10          SelectionHeuristicBasis = TREESIZE;
            FitnessAdjustmentFunction = One_DividedBy_OnePlusX;
            Selection = PROPORTIONAL;
```

```
            LinearRankSelectParameter = 0.2;
            Sampling = STOCHASTIC_UNIVERSAL;
15          if(Current==NULL)
                   NoOfMates = 0;
            else
                   NoOfMates = Current->ActualPopSize;

20          Mating = RANDOM_PERMUTATION;
            NoOfCrossOverPoints = 1;
            CrossOverProbability = 0.85;
            MutationProbability = 0.05;
            SaveBest = TRUE;
25          SmallestSubTreeOfVirtualPop = 0;
            DynamicFitnessScalingParameter = 0;
            Goal = MAXIMIZE;
            ComputeFitnessValues();
     }
```

Folgende Standardoperatoren und Standardparameter werden festgelegt:

1. **Selektionsheuristik zur Auswahl von Teilbäumen**: keine (Zeile 9),
2. **Basis für eine eventuelle Selektionsheuristik**: Ableitungstiefe der Individuen (Zeile 10),
3. **Fitnessadjustierung**: $\frac{1}{1+f_s}$ (Zeile 11),
4. **Selektion**: proportionale Selektion (Zeile 12),
5. **Parameter a_{min} für die Linear-Rank-Selektion**: 0,2 (Zeile 13),
6. **Sampling**: stochastic universal sampling (Zeile 14),
7. **Anzahl der zu sampelnden Individuen** (zugleich Größe der nächsten Population): gleich der aktuellen tatsächlichen Populationsgröße (Zeilen 15-18),
8. **Mating**: random permutation mating (Zeile 20),
9. **Kreuzung**: One-Point-Crossover mit 85% Wahrscheinlichkeit (Zeilen 21 und 22),
10. **Mutation**: Mit 5% Wahrscheinlichkeit (Zeile 23),
11. **Elitismusstrategie**: aktiviert (Zeile 24),
12. **Virtuelle Population**: besteht nur aus den Individuen der tatsächlichen Population (Zeile 25),
13. **Rohfitness-Skalierung**: Unter Berücksichtigung nur der aktuellen Population (Zeile 26),
14. **Optimierungsziel**: Rohfitnessmaximierung (Zeile 27).

Zuletzt werden in Zeile 27 mit Hilfe der Funktion `ComputeFitnessValues()` die Fitnesswerte aller in der virtuellen Population vermerkten Individuen neu berechnet.

6.5 Zufallszahlengenerator

Bei dem im vorliegenden Programm verwendeten Zufallszahlengenerator handelt es sich um eine vom Autor vorgenommene C++-Adaptierung des in [Law und Kelton, 1991] vorgestellten PASCAL-Zufallszahlengenerators Uniran. Im Datenteil der Klasse `Uniran` befindet sich lediglich die Variable `zrng`, in welcher die letzte Zufallszahl als Ausgangswert für die Berechnung der nächsten Zufallszahl aufbewahrt wird.

6.5.1 Uniran()

Der Leerkonstruktor setzt die `seed` für die erste Zufallszahl auf die mit Hilfe der Funktion `time(NULL)` ermittelte Systemzeit (Zeile 6).

```
    Uniran::Uniran()
    //
    // Constructor: Initializing seed zrng with system-time
    //
 5  {
            zrng=time(NULL);
    }
```

6.5.2 Uniran(int)

Dieser Konstruktor initialisiert bei Angabe einer positiven Zahl die *seed* mit der angegebenen Zahl. Bei Angabe einer negativen Zahl zwischen minus eins und minus hundert wird die *seed* mit einem von hundert vorbereiteten Werten initialisiert. Bei Angabe von null oder einem Wert kleiner als minus hundert wird der Zufallszahlengererator, wie beim Leerkonstruktor, mit der Systemzeit initialisiert.

```
    Uniran::Uniran(int i)
    //         CONSTRUCTOR:
    //         i>0: seed zrng is set to i
    // -100<=i<=-1: seed zrng is set to one of 100 predefined values
 5  //        else: seed zrng is set to system-time
    //
    {
            switch (i)
            {
10          case -1:
                    zrng=1973272912;
                    break;
            case -2:
                    zrng= 281629770;
15                  break;
            case -3:
                    zrng=  20006270;
                    break;
            case -4:
20                  zrng=1280689831;
```

```
                break;
            case -5:
                zrng=2096730329;
                break;
25          case -6:
                zrng=1933576050;
                break;
            case -7:
                zrng= 913566091;
30              break;
            case -8:
                zrng= 246780520;
                break;
            case -9:
35              zrng=1363774876;
                break;
            case -10:
                zrng= 604901985;
                break;
40          case -11:
                zrng=1511192140;
                break;
            case -12:
                zrng=1259851944;
45              break;
            case -13:
                zrng= 824064364;
                break;
            case -14:
50              zrng= 150493284;
                break;
            case -15:
                zrng= 242708531;
                break;
55          case -16:
                zrng=  75253171;
                break;
            case -17:
                zrng=1964472944;
60              break;
            case -18:
                zrng=1202299975;
                break;
            case -19:
65              zrng= 233217322;
                break;
            case -20:
                zrng=1911216000;
                break;
70          case -21:
                zrng= 726370533;
                break;
            case -22:
                zrng= 403498145;
75              break;
            case -23:
                zrng= 993232223;
                break;
            case -24:
80              zrng=1103205531;
                break;
            case -25:
                zrng= 762430696;
                break;
85          case -26:
                zrng=1922803170;
                break;
```

```
            case -27:
                    zrng=1385516923;
 90                 break;
            case -28:
                    zrng=  76271663;
                    break;
            case -29:
 95                 zrng= 413682397;
                    break;
            case -30:
                    zrng= 726466604;
                    break;
100         case -31:
                    zrng= 336157058;
                    break;
            case -32:
                    zrng=1432650381;
105                 break;
            case -33:
                    zrng=1120463904;
                    break;
            case -34:
110                 zrng= 595778810;
                    break;
            case -35:
                    zrng= 877722890;
                    break;
115         case -36:
                    zrng=1046574445;
                    break;
            case -37:
                    zrng=  68911991;
120                 break;
            case -38:
                    zrng=2088367019;
                    break;
            case -39:
125                 zrng= 748545416;
                    break;
            case -40:
                    zrng= 622401386;
                    break;
130         case -41:
                    zrng=2122378830;
                    break;
            case -42:
                    zrng= 640690903;
135                 break;
            case -43:
                    zrng=1774806513;
                    break;
            case -44:
140                 zrng=2132545692;
                    break;
            case -45:
                    zrng=2079249579;
                    break;
145         case -46:
                    zrng=  78130110;
                    break;
            case -47:
                    zrng= 852776735;
150                 break;
            case -48:
                    zrng=1187867272;
                    break;
            case -49:
```

```
155                 zrng=1351423507;
                    break;
            case -50:
                    zrng=1645973084;
                    break;
160         case -51:
                    zrng=1997049139;
                    break;
            case -52:
                    zrng= 922510944;
165                 break;
            case -53:
                    zrng=2045512870;
                    break;
            case -54:
170                 zrng= 898585771;
                    break;
            case -55:
                    zrng= 243649545;
                    break;
175         case -56:
                    zrng=1004818771;
                    break;
            case -57:
                    zrng= 773686062;
180                 break;
            case -58:
                    zrng= 403188473;
                    break;
            case -59:
185                 zrng= 372279877;
                    break;
            case -60:
                    zrng=1901633463;
                    break;
190         case -61:
                    zrng= 498067494;
                    break;
            case -62:
                    zrng=2087759558;
195                 break;
            case -63:
                    zrng= 493157915;
                    break;
            case -64:
200                 zrng= 597104727;
                    break;
            case -65:
                    zrng=1530940798;
                    break;
205         case -66:
                    zrng=1814496276;
                    break;
            case -67:
                    zrng= 536444882;
210                 break;
            case -68:
                    zrng=1663153658;
                    break;
            case -69:
215                 zrng= 855503735;
                    break;
            case -70:
                    zrng=  67784357;
                    break;
220         case -71:
                    zrng=1432404475;
```

```
                    break;
            case -72:
                    zrng= 619691088;
225                 break;
            case -73:
                    zrng= 119025595;
                    break;
            case -74:
230                 zrng= 880802310;
                    break;
            case -75:
                    zrng= 176192644;
                    break;
235         case -76:
                    zrng=1116780070;
                    break;
            case -77:
                    zrng= 277854671;
240                 break;
            case -78:
                    zrng=1366580350;
                    break;
            case -79:
245                 zrng=1142483975;
                    break;
            case -80:
                    zrng=2026948561;
                    break;
250         case -81:
                    zrng=1053920743;
                    break;
            case -82:
                    zrng= 786262391;
255                 break;
            case -83:
                    zrng=1792203830;
                    break;
            case -84:
260                 zrng=1494667770;
                    break;
            case -85:
                    zrng=1923011392;
                    break;
265         case -86:
                    zrng=1433700034;
                    break;
            case -87:
                    zrng=1244184613;
270                 break;
            case -88:
                    zrng=1147297105;
                    break;
            case -89:
275                 zrng= 539712780;
                    break;
            case -90:
                    zrng=1545929719;
                    break;
280         case -91:
                    zrng= 190641742;
                    break;
            case -92:
                    zrng=1645390429;
285                 break;
            case -93:
                    zrng= 264907697;
                    break;
```

```
          case -94:
290               zrng= 620389253;
                  break;
          case -95:
                 zrng=1502074852;
                 break;
295       case -96:
                 zrng= 927711160;
                 break;
          case -97:
                 zrng= 364849192;
300              break;
          case -98:
                 zrng=2049576050;
                 break;
          case -99:
305              zrng= 638580085;
                 break;
          case -100:
                 zrng= 547070247;
                 break;
310       default:
                 zrng=(i>0 ? i : time(NULL));
                 break;
          }
      }
```

6.5.3 rand()

Diese Funktion liefert die nächste Pseudozufallszahl im Bereich zwischen null und eins.

```
     float Uniran::rand()
     //
     // provides next floating-point random-number
     //
 5   {
             int hi15, hi31, low15, lowprd, ovflow, zi, rand;

             zi=zrng;
             hi15=zi/B2E16;
10           lowprd=(zi - hi15 * B2E16) * MULT1;
             low15=lowprd/B2E16;
             hi31=hi15 * MULT1 + low15;
             ovflow=hi31/  B2E15;
             zi=(((lowprd-low15*B2E16)-MODLUS)+(hi31-ovflow*B2E15)*B2E16)+ovflow;
15           if(zi < 0)
                     zi=zi + MODLUS;
             hi15=zi/B2E16;
             lowprd=(zi - hi15 * B2E16) * MULT2;
             low15=lowprd/B2E16;
20           hi31=hi15 * MULT2 + low15;
             ovflow=hi31/B2E15;
             zi=(((lowprd-low15*B2E16)-MODLUS)+(hi31-ovflow*B2E15)*B2E16)+ovflow;
             if(zi < 0)
                     zi=zi + MODLUS;
25           zrng= zi;
             rand=2*(zi/256)+1;
             return(rand / 16777216.0);
     }
```

6.5.4 dice(int)

Diese Funktion liefert eine ganzzahlige Zufallszahl im Bereich zwischen null (inklusive) und der im Parameter angegebenen Zahl (exklusive). Wird diese Funktion mit null aufgerufen, liefert sie immer minus eins zurück.

```
    int Uniran::dice(int i)
    //
    // provides next int random-number 0 <= random-number < i
    //
 5  {
            int r;
            r = int( (i+0.0)*rand() );
            return( (r==i ? i - 1 : r ) );
    }
```

Literatur

[Aho, Sethi und Ullman, 1988] Alfred V. Aho, Ravi Sethi und Jeffry D. Ullman. Compilerbau. Addison-Wesley, Bonn u.a. 1988.

[Baker, 1987] James E. Baker. Reducing bias and inefficiency in the selection algorithms. In [Grefenstette, 1985, S. 101-111].

[Belew and Booker, 1991] Richard K. Belew und Lashon B. Booker (Hrsg.). Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[Duden Informatik, 1993] Hermann Engesser (Hrsg). Duden Informatik: Ein Sachlexikon für Studium und Praxis, 2. Auflage. Brockhaus, Mannheim; Leipzig; Wien, 1993.

[DeJong, 1988] Kenneth A. DeJong. Learning with Genetic Al gorithms: An Overview. Machine Learning 3:121-138, Kluwer Academic Publishers, 1988

[DeJong, 1975] Kenneth A. DeJong. An analysis of the behavior of a class of genetic adaptive systems. Doctoral dissertation, University of Michigan), zitiert in [Goldberg, 1989].

[Geyer-Schulz und Böhm, 1996] Andreas Geyer-Schulz und Walter Böhm. Exact Uniform Initialization For Genetic Programming. Abteilung für Angewandte Informatik, Wirtschaftsuniversität WIen, Augasse 2-6, A-1090 Wien, Österreich. Accepted: FOGA4, San Diego, 1996.

[Geyer-Schulz, 1995] The MIT Beer Distribution Game Revisited: Genetic Machine Learning an Managerial Behavior in a Dynamic Decision Making Experiment; in: F. Herrera, J.L. Verdegay (Eds.), Genetic Algorithms and Soft Computing, Studies in Fuzziness, Physica-Verlag, Heidelberg, to appear.

[Geyer-Schulz, 1995] Andreas Geyer-Schulz. Fuzzy Rule-Based Expert Systems and Genetic Machine Learning (Studies in Fuzziness, Vol. 3), Physica-Verlag, Wien 1995.

[Geyer-Schulz, 1994] Andreas Geyer-Schulz. Fuzzy Rule-Based Expert Systems and Genetic Machine Learning. Habilitationsschrift Wirtschaftsuniversität Wien, 1994.

[Geyer-Schulz, 1992] Andreas Geyer-Schulz. On learning in a fuzzy rule-based expert system, Kybernetika, 28:33-36, 1992.

[Geyer-Schulz, 1989] Andreas Geyer-Schulz. MEMO. APL Quote Quad, 20(2):12-27, Dezember 1989. ACM, New York.

[Goldberg *et al*, 1990] David E. Goldberg, Bradly Korb und Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems, 3(5):493-530, 1990.

[Goldberg, 1989] David E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Weseley, Reading, Massachusetts, 1989.

[Grefenstette und Baker, 1989] John J. Grefenstette und James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In [Schaffer, 1989, S. 112-120].

[Grefenstette, 1986] John J. Grefenstette. Optimization of Contorl Parameters for genetic Algorithms. IEEE Trans. SMC, 16(1), S. 122-128.

[Grefenstette, 1985] John J. Grefenstette (Hrsg.). Proceedings of the First International Conference on Genetic Algorithms and their Applications, Lawrence Erlbaum Associates, Hillsdale, New Jersy, 1985.

[Hoffmeister 1992] F. Hoffmeister und T. Bäck. Genetic Algorithms and Evolution Strategies: Similarities and Differences. Univ. Dortmund, Technical Report Sys.1/92, 1992.

[Holland, 1975] John H. Holland. Adaption in Natural and Artificial Systems. The University of Michigan Press, Ann Arbor, Michigan 1975.

[Holland, 1973] John H. Holland. Genetic algorithms and the optimal allocation of trials. SIAM Journal of Computing 2(2):88-105, Juni 1973.

[Koza, 1992] John R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, Massachusetts, 1992.

[Law und Kelton, 1991] Law, A.M. und Kelton, D.W. Simulation Modeling and Analysis. New York, McGraw-Hill, 1991.

[Levenick 1991] James R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In [Belew and Booker, 1991], S. 123-127.

[Rechenberg, 1973] Ingo Rechenberg. Evolutionsstrategie. Band 15 *problemata*. Friedrich Frommann Verlag (Günter Holzboog KG), Stuttgart, 1973.

[Schaffer, 1989] J. David Schaffer, Hrsg. Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Mateo, California, 1989.

[Schöneburg *et al.*, 1994] Eberhard Schöneburg, Frank Heinzmann und Sven Feddersen. Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in die Theorie und Praxis der simulierten Evolution. Addison-Wesley, Bonn; Paris; Reading, Massachusetts, 1994.

[Sterman, 1989] John D. Sterman. Modeling managerial behaviour: Misperceptions of feedback in a dynamic decision making experiment. Management Science, 35(3): S. 321-339, März 1989.

[Wirth 1986] Niklaus Wirth. Compilerbau: eine Einführung. 2. Auflage. Teubner, Stuttgart 1986.

Abbildungsverzeichnis

1 Funktionsweise des einfachen genetischen Algorithmus 9

2 Kreuzen zweier Strings im einfachen genetischen Algorithmus 11

3 Die BNF der Grammatik L_{XOR} 14

4 Der Ableitungsbaum für `(NOT(OR(D1)(D2)))` 16

5 Der Kreuzungsoperator für Ableitungsbäume 17

6 Die implementierten genetischen Operatoren 19

7 Das Nichtterminalsymbol $<\gamma>$ mit seinen drei möglichen Ableitungen X, Y und Z 22

8 Die Ableitung $<\alpha><\beta><\gamma>$ und ihr Suchraum 24

9 Symbol $<\gamma>$ bekommt die Anweisung, einen Baum der Tiefe $i = 3$ zu erzeugen. 28

10 Ableitung $<\alpha><\beta>$“T“ bekommt die Anweisung, einen Baum der Tiefe $i = 3$ zu erzeugen. 29

11 Roulett-Wheel Selektion 34

12 Stochastic Universal Sampling: Auswahl von 5 Individuen 34

13 Die benutzerrelevanten Klassen 47

14 Die Zusammenhänge zwischen den benutzerrelevanten Klassen 50

15 Bestimmung der Eltern 57

16 Die Klassen zur Grammatikrepräsentation 70

17 Zusammenhang zwischen den Klassen `Language` und `SymbTabEntry` . . 71

18 Die Klassen `SymbTabEntry`, `ProdTabEntry` und `DerivSymbol` 72

19 Die Klasse `Language` 73

20 Die Klasse `SymbTabEntry` 86

21 Die Klasse `ProdTabEntry` 89

22 Die Klasse `SuperFloat` 92

23 Die Klassen `TreeNode`, `StartNode` und `RootNode` 108

24 Die Klasse `RootNode` 110

25 Die Klasse `StartNode` 118

26 Knotenindexierung mit Hilfe der Ableitungstiefe 125

27 Die Klasse `TreeNode` 131

28 Die Klasse `Population` 154

29 Auswahl von Individuen mittels stochastic universal sampling 162

30 Die Klasse `Farm` 181

Tabellenverzeichnis

1 Anzahl von Wörtern in einer Testpopulation n=100 21

2 Suchraumgrößen für L_{XOR} 22

3 Suchraumberechnung für die Ableitung $< \alpha >< \beta >$ 25

4 Suchraumberechnung für die Ableitung $< \alpha >< \beta >< \gamma >$ 25

Listings

A Anhang

A.1 Listings

A.1.1 Die Headerdatei ga.h

```
/*=======================================================
   VIENNA UNIVERSITY OF ECONOMICS GENETIC PROGRAMMING KERNEL

   HEADER FILE

   (C) 1996 Helmut Hoerner
   email: h8850092@wu-wien.ac.at
   FAX: +43-1-9820904
  =======================================================
*/


// Prototypes

#ifndef __GA_H__
#define __GA_H__

#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include<math.h>
#include<stdlib.h>

#include "uniran.h"

extern "C"
{
#include<time.h>
#include<string.h>
#include<stdio.h>
}

#define CHRBUFSIZE 256
#define LOADFLPOINTBFF 255
#define FILEFORMAT 1
#define YES 1
#define NO 0
#define TRUE 1
#define FALSE 0
#define ON 1
#define OFF 0
#define EQUALSTRINGS 0
#define EXACTUNIF 2
#define SIZEUNIF 1
#define NONUNIF 0
#define MINIMIZE 0
#define MAXIMIZE 1
#define DEFAULTSELECTION 0
#define HEURISTICSELECTION 1
#define TREESIZE 0
#define SEARCHSPACESIZE 1
#define LONG 0
#define SHORT 1
#define PROPORTIONAL 0
#define LINEAR_RANK 1
#define STOCHASTIC_UNIVERSAL 0
```

```
#define STOCH_SAMPL_WITH_REPL 1
#define RANDOM_PERMUTATION 0
#define RANDOM 1
#define ACC 50

// SEEDS FOR RANDOM-NUMBER GENERATION

#define RANDTREESEED -1
#define RANDSAMPLSEED -2
#define MATINGDICESEED -3
#define RANDNPCROSSSEED -4
#define SELCROSSSEED -8
#define RANDMUTSEED -5
#define RANDMUTSELECT -6
#define RANDSAVEBESTSEED -7
#define RANDPOPINIT -8
#define STOCHUNIVSAMPLSEED -9
#define PERMUTSEED -10
#define IMPRSELCROSSSEED -11
#define RANDHEUMUTSEED -12


// CLASS AND STRUCT PROTOTYPES

class SuperFloat;
class SuperUnsignedInt;
class TreeNode;
class StartNode;
class RootNode;
class Population;
class Farm;
class Language;
class SymbTabEntry;
class ProdTabEntry;

struct DerivSymbol;
struct CrossOverParameters;
struct MatingListEntry;


// EXTERN FUNCTION PROTOTYPES

double PropSelectionHeuristic(double);
double pow2(double);
double pow3(double);
double pow4(double);
double One_DividedBy_OnePlusX (double);


/* ----------------------------------------------------
   SuperUnsignedInt class prototype

   UNLIMITED LENGTH INTEGER ARITHMETIC
   CRUDE IMPLEMETATION FOR EXPERIMENTAL USE ONLY
   ----------------------------------------------------
*/
class SuperUnsignedInt
{
        friend ProdTabEntry;

private:
        int Length;
        unsigned char* Digit;

public:
        SuperUnsignedInt();
        ~SuperUnsignedInt();
```

```
        int operator=(int);
        int operator=(SuperUnsignedInt);
        int operator=(unsigned int);
        int operator=(long int);
        int operator!=(int);
        int operator+=(SuperUnsignedInt&);
        int operator*=(SuperUnsignedInt&);
        friend ostream& operator <<(ostream&, SuperUnsignedInt&);
};


/* ---------------------------------------------------
   SuperFloat class prototype

   FLOATING POINT ARITHMETIC WITH DOUBLE PRECISION FOR
   VERY, VERY LARGE NUMBERS...
   ---------------------------------------------------
*/
class SuperFloat
{
        friend Population;
        friend Farm;

private:
        long double Mant;
        long int Exp;

public:
        SuperFloat();
        int operator=(int);
        int operator=(float);
        int operator=(double);
        int operator!=(int);
        int operator>(SuperFloat);
        int operator>(int);
        SuperFloat operator--();
        SuperFloat operator+(SuperFloat);
        SuperFloat operator*(SuperFloat);
        friend ostream& operator <<(ostream&, SuperFloat&);
};


/* ---------------------------------------------------
   Language class prototype

   MANAGEMENT OF A SPECIFIC GRAMMAR:
   CONTAINS SYMBOL TABLE
   ---------------------------------------------------
*/
class Language
{
public:
        SymbTabEntry* FirstSymbol;
        SymbTabEntry* LastSymbol;
        SymbTabEntry* StartSymbol;
        SuperFloat* ArbCard;
        int ArbCardSize;
        SuperUnsignedInt* ExactCard;
        int ExactCardSize;
public:
        Language();
        ~Language();
        void Clear();
        SymbTabEntry* Symbol(char*);
        void Print();
        int Load(char*);
        int ComputeExactSearchSpace(int);
```

```
        int ComputeArbSearchSpace(int);
        void PrintExactSearchSpace();
        void PrintArbSearchSpace();
        friend ostream& operator <<(ostream&, Language&);
};

/* ---------------------------------------------------
   SymbTabEntry class prototype

   BUILDING BLOCK OF SYMBOL TABLE IN CLASS LANGUAGE
   REPRESENTING A SYMBOL OF A SPECIFIC GRAMMAR
   CONTAINING A LIST OF POSSIBLE DERIVATIONS
   (A PRODUCTION TABLE)

   SEARCH SPACE SIZE CALCULATIONS
   ---------------------------------------------------
*/
class SymbTabEntry
{
        friend Language;
        friend Farm;

public:
        char* Symbol;
        Language* L;
        ProdTabEntry* DerivList;
        int NoOfDerivs;
        SymbTabEntry* NextEntry;
private:
        SuperUnsignedInt* ExactCard;
        SuperFloat* ArbCard;
        int ExactCardSize;
        int ArbCardSize;

public:
        SymbTabEntry();
        SymbTabEntry(char*, Language*);
        ~SymbTabEntry();
        ProdTabEntry* AddProdTabEntry();
        SuperUnsignedInt* ComputeExactSearchSpace(int);
        SuperFloat* ComputeArbSearchSpace(int);
};

/* ---------------------------------------------------
   DerivSymbol struct prototype

   MAKING CHAINED LISTS OF DERIVATION SYMBOLS
   (I.E. A COMPLETE DERIVATION) VIA
   NextSymbol-POINTER POSSIBLE.
   ---------------------------------------------------
*/
struct DerivSymbol
{
        SymbTabEntry* Symbol;
        DerivSymbol* NextSymbol;
};


/* ---------------------------------------------------
   ProdTabEntry class prototype

   BUILDING BLOCK OF PRODUCTION TABLE IN CLASS SymbTabEntry
   REPRESENTING A DERIVATION OF A SPECIFIC SYMBOL

   SEARCH SPACE SIZE CALCULATIONS
   ---------------------------------------------------
*/
```

```
class ProdTabEntry
{
        friend Language;

public:
        DerivSymbol* FirstSymbol;
        DerivSymbol* LastSymbol;
        int NoOfDerivSymbols;
private:
        SuperUnsignedInt* ExactCard;
        SuperFloat* ArbCard;
        int ExactCardSize;
        int ArbCardSize;

public:
        ProdTabEntry();
        ~ProdTabEntry();
        void Release();
        DerivSymbol* AddDerivSymbol(SymbTabEntry*);
        SuperUnsignedInt* ComputeExactSearchSpace(int);
        SuperFloat* ComputeArbSearchSpace(int);
};


/* ---------------------------------------------------
   CrossOverParameters struct prototype

   A SET OF COMPLETE CROSSOVER-PARAEMETERS
   ---------------------------------------------------
*/
struct CrossOverParameters
{
        StartNode* FirstTree;
        TreeNode* CrossOverPoint;
        TreeNode* SecondSubTree;
};



/* ---------------------------------------------------
   TreeNode class prototype

   NODE OF A DERIVATION TREE
   ---------------------------------------------------
*/
class TreeNode
{

public:
        RootNode* Root;
        SymbTabEntry* Symbol;
        Language* TreeLanguage;
        TreeNode** Derivs;
        int NoOfDerivSymbols;
        TreeNode* PredecessorNode;
        unsigned int DerivationDepth;
        unsigned int TempIndex;
protected:
        int DerivationNo;

public:
        TreeNode();
        TreeNode(SymbTabEntry*, Language*, TreeNode*,  RootNode*);
        ~TreeNode();
        int operator =(TreeNode&);
        void EnhanceWith(TreeNode*);
        void DerivationDepthUpdate();
```

```
        virtual void AddCompleteSubTreesTo(Population*, unsigned int);
        unsigned int NoOfSubTrees(SymbTabEntry*, unsigned int);
        unsigned int Depth();
        void Print();
        friend ostream& operator <<(ostream&, TreeNode&);
        int Evaluate();
        int Evaluate(int);
        int Evaluate(int, int);
        int Evaluate(int, int, int);
        int Evaluate(int, int, int, int);
        float Evaluate(float);
        float Evaluate(float, float);
        float Evaluate(float, float, float);
        float Evaluate(float, float, float, float);
 protected:
        int CompareWith(TreeNode*);
        int GenoTypeStringLength(int);
        int PhenoTypeStringLength();
        void BuildUpGenoTypeStringIn(char*, int&, int);
        void BuildUpPhenoTypeStringIn(char*, int&);
        int EnhanceWithString(char*,int&);
        TreeNode* XEnhanceWith(TreeNode*, TreeNode*, TreeNode*);
        int RandomEnhance(unsigned int, unsigned int&);
        void EnhanceWith(ProdTabEntry*);
};


/* ----------------------------------------------------
   StartNode class prototype

   START NODE OF A DERIVATION TREE
   (I.E: NODE, CONTAINING A START-SYMBOL).
   CAN BE HOLDING FITNESS VALUES
   ----------------------------------------------------
*/
class StartNode : public TreeNode
{

public:
        float RawFitness;
        float StandardizedFitness;
        float AdjustedFitness;
        float NormalizedFitness;
        float TargSamplRate;
protected:
        char* PhenoString;
        char* GenoString;

public:
        StartNode();
        StartNode(Language*, TreeNode*,  RootNode*);
        void AddCompleteSubTreesTo(Population*, unsigned int);
        TreeNode* SubTree(unsigned int);
        TreeNode* SubTree(SymbTabEntry*, unsigned int, unsigned int);
        float ObjFuncVal();
        virtual int operator =(StartNode&);
        int operator ==(StartNode&);
        void PrintParents();
        char* GenoTypeString(int);
        char* PhenoTypeString();
protected:
        void EnhanceWith(StartNode*);
};


/* ----------------------------------------------------
   RootNode class prototype
```

```
   ROOT NODE OF A COMPLETE DERIVATION TREE,
   INCLUDING POINTER TO ITS PARENTS, A FLAG CONTAINING
   THE INFORMATION WETHER A MUTATION HAS OCCURED OR NOT,
   AND A POINTER TO ITS 'HOME'-POPULATION.
   ---------------------------------------------------
*/
class RootNode : public StartNode
{
public:
        StartNode* Parent1;
        StartNode* Parent2;
        int Mutated;
        Population* Home;

public:
        RootNode();
        RootNode(Language*, int, Population*);
        int Set(char*);
        int operator =(StartNode&);
        int CrossOver(CrossOverParameters);
};

/* ---------------------------------------------------
   MatingListEntry struct prototype

   BUILDING BLOCK OF MATING LIST:
   CAN HOLD A POINTER TO TWO PARTNERS,
   SELECTED FOR MATING
   ---------------------------------------------------
*/
struct MatingListEntry
{
        StartNode* Partner1;
        StartNode* Partner2;
};

/* ---------------------------------------------------
   Population class prototype

   HOLDING A CERTAIN NUMBER OF 'INDIVIDUALS'
   (I.E. COMPLETE DERIVATION TREES)
   IN ITS ACTUAL POPULATION.
   THE VIRTUAL PART OF THE POPULATION CAN HOLD POINTERS
   TO EVERY STARTNODE OF THE ACTUAL POPULATION
   (I.E: SUBTREES OF THE POPULATION)
   ---------------------------------------------------
*/
class Population
{
        friend Farm;

public:
        int VirtualPopSize;
        StartNode** Virtual;
        int ActualPopSize;
        RootNode* Actual;
        int ParentsAvailable;
        StartNode* BestIndividual;
        StartNode* WorstIndividual;
        float AdjustedFitnessSum;
private:
        unsigned int SmallestSubTreeOfVirtualPop;
        int VirtualPopArraySize;
        int ActualPopArraySize;
        int MatingListArraySize;
        MatingListEntry* MatingList;
```

```
        int MatingListNoOfEntries;

public:
        Population();
        Population(Language*, int,int);
        ~Population();
        int GenerateAlmostUniformActualPop(Language*, int, int);
        void GenerateVirtualPopInclSubTrees(unsigned int);
        void GenerateVirtualPop();
        void AddTreeToVirtualPop(StartNode* Tree);
        void ComputeBestAndWorstIndividual(int);
        void StandardizeWith(float, int);
        void ApplyRawFitness();
        void InvertFitnessValues();
        void NormalizeFitnessValues();
        void PropSelect();
        void LinearRankSelect(float);
        void StochSamplWithRepl(int);
        void StochUnivSampl(int);
        void RandomMate();
        void RandomPermutationMate();
private:
        void ReSizeActualPop(int);
        void ReSizeVirtualPop(int,int);
        void ReSizeMatingList(int);
};


/* ----------------------------------------------------
   Farm class prototype

   CONTAINING THE LAST History+1 POPULATIONS
   FOR KEEPING TRACK OF PARENTS AND PERFORMING
   GENETIC OPERATOR OVER MORE THAN ONE POULATION.

   OFFERING THE MOST ABSTRACT LEVEL:
   THE NextStep() METHOD.
   ----------------------------------------------------
*/
class Farm
{
public:
        Language L;
        Population* Current;
private:
        int UnifProb;
        int History;
        int MaxDerivDepth;
        unsigned int SmallestSubTreeOfVirtualPop;
        float LinearRankSelectParameter;
        int Sampling;
        int NoOfMates;
        int Mating;
        int Selection;
        int NoOfCrossOverPoints;
        float CrossOverProbability;
        int SaveBest;
        float MutationProbability;
        int DynamicFitnessScalingParameter;
        int Goal;
        int PopulationIndex;
        int SelectionHeuristicBasis;
        Population* ThePopulation;
        double (*SelectionHeuristic) (double);
        double (*FitnessAdjustmentFunction) (double);

public:
```

```
        Farm();
        Farm(int);
        Farm(int, int, int, char*, int);
        Farm(char*);
        ~Farm();
        void NextStep();
        int Save(char*);
        void Print();
        void SetFitnessScaling(unsigned int, int);
        void SetFitnessAdjustment(double (*)(double));
        void SetSelection(int);
        void SetSampling(int,int);
        void SetMating(int);
        void SetSelectionHeuristic(double (*)(double)=NULL, int=TREESIZE);
        void SetCrossOver(int, float);
        void SetMutation(float);
        void SetSaveBest(int);
        void SetSmallestSubTreeOfVirtualPop(unsigned int);
        void SetMaxDerivDepth(int);
private:
        void StandardizeFitnessValues(int);
        void ApplyDynamicFitnessScaling(unsigned int, int);
        void AdjustFitnessValues();
        void NPointCrossOver(unsigned int, float, int);
        CrossOverParameters SelectCrossOverPoints(StartNode*, StartNode*);
        CrossOverParameters SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*);
        void Mutate(float, int);
        void SaveBestIndividual();
        void SetDefaultMethods();
        int ComputeFitnessValues();
};

#endif
```

A.1.2 Die Datei farm.cc

```
// File: farm.cc

#include "ga.h"

/*=======================================================
   IMPLEMENTATION OF CLASSES

   Population
   Farm

   AND EXTERN FUNCTION DEFINITIONS FOR SELECTION
   HEURISTIC AND FITNESS INVERSION
  =======================================================
*/


/*-------------------------------------------------------
  double ProportionalSelection(double x)

  Can be assigned to
  Farm::double (*SelectionHeuristic) (double)
  using
  Farm::SetSelectionHeuristic(double (*)(double), int)

  A possible selection heuristic to be used by
  Farm::SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*)
  -------------------------------------------------------
*/
double PropSelectionHeuristic(double x)
{
        return(x);
}


/*-------------------------------------------------------
  double pow2(double x)

  Can be assigned to
  Farm::double (*SelectionHeuristic) (double)
  using
  Farm::SetSelectionHeuristic(double (*)(double), int)

  A possible selection heuristic to be used by
  Farm::SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*)
  -------------------------------------------------------
*/
double pow2(double x)
{
        return(x*x);
}


/*-------------------------------------------------------
  double pow3(double x)

  Can be assigned to
  Farm::double (*SelectionHeuristic) (double)
  using
  Farm::SetSelectionHeuristic(double (*)(double), int)

  A possible selection heuristic to be used by
  Farm::SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*)
  -------------------------------------------------------
```

```
*/
double pow3(double x)
{
        return(x*x*x);
}


/*--------------------------------------------------------
  double pow4(double x)

  Can be assigned to
  Farm::double (*SelectionHeuristic) (double)
  using
  Farm::SetSelectionHeuristic(double (*)(double), int)

  A possible selection heuristic, to be used by
  Farm::SelectCrossOverPointsUsingHeuristic(StartNode*, StartNode*)
  -------------------------------------------------------
*/
double pow4(double x)
{
        return(x*x*x*x);
}


/*--------------------------------------------------------
  double One_DividedBy_OnePlusX(double x)

  Can be assigned to
  Farm:: double (*FitnessAdjustmentFunction) (double)
  using
  Farm::SetFitnessAdjustment(double (*)(double))

  A possible fitness adjustment function, to be used by
  Farm::AdjustFitnessValues()
  --------------------------------------------------------
*/
double One_DividedBy_OnePlusX(double x)
{
        return(1.0/(1.0+x));
}




/*=======================================================
  The Population class....
  ======================================================
*/


/*--------------------------------------------------------
  Population::Population()

  Constructor: Creating an empty Population
  --------------------------------------------------------
*/
Population::Population()
{
        ActualPopSize = 0;
        ActualPopArraySize = 0;
        Actual = NULL;
        VirtualPopSize = 0;
        VirtualPopArraySize = 0;
        Virtual = NULL;
        ParentsAvailable = NO;
        AdjustedFitnessSum = 0;
```

```
        MatingList = NULL;
        MatingListArraySize = 0;
        MatingListNoOfEntries = 0;
        SmallestSubTreeOfVirtualPop = 0;
}


/*----------------------------------------------------------
  Population::Population(Language* L, int PopSize, int MaxDepth)

  Constructor: Creating a new Population of language L
  with PopSize Trees, each with maximum derivation depth
  MaxDepth
  ----------------------------------------------------------
*/
Population::Population(Language* L, int PopSize, int MaxDepth)
{
        ActualPopSize = PopSize;
        ActualPopArraySize = PopSize;
        VirtualPopSize = 0;
        VirtualPopArraySize = 0;
        Virtual = NULL;
        SmallestSubTreeOfVirtualPop = 0;

        Actual = new RootNode[PopSize](L, MaxDepth, this);

        GenerateVirtualPop();
        ParentsAvailable = NO;
        AdjustedFitnessSum = 0;
        MatingList = new MatingListEntry[VirtualPopSize];
        MatingListArraySize = VirtualPopSize;
        MatingListNoOfEntries = VirtualPopSize;
}


/*----------------------------------------------------------
  Population::~Population()

  Destructor: Deleting the actual and the virtual
  population and the mating-list.
  ----------------------------------------------------------
*/
Population::~Population()
{
        if(ActualPopArraySize>0)
                delete [] Actual;

        if(VirtualPopArraySize>0)
                delete [] Virtual;

        if(MatingListArraySize>0)
                delete [] MatingList;
}


/*----------------------------------------------------------
  Population::GenerateAlmostUniformActualPop(Language* L, int PopSize, int Depth)

  Generating an more or less uniform initital population
  using the heuristic due to Andreas Geyer-Schulz
  ----------------------------------------------------------
*/
int Population::GenerateAlmostUniformActualPop(Language* L, int PopSize,  int Depth)
{
        static Uniran SpinningWheel(RANDPOPINIT);
        SuperFloat* RouletteSlot;
        SuperFloat* NoOfAvailableIndividuals;
```

```
SuperFloat SelectedSlot;
SuperFloat Steps;
int* TargetNumberOfIndividuals;
RootNode* TempTree;
int i;
int j;
SuperFloat k;

if(L->ArbCard==NULL)
{
        cout << "Computing search space size up to n = " << Depth;
        cout << "..." << endl;
        L->ComputeArbSearchSpace(Depth);
        cout << "done." << endl;
}
else
        if(L->ArbCardSize<Depth)
        {
                cout << "Computing search space size up to n = " << Depth;
                L->ComputeArbSearchSpace(Depth);
                cout << "done." << endl;
        }

cout << "Generating population..." << endl;

if(Actual!=NULL)
        delete [] Actual;

ActualPopSize = PopSize;
ActualPopArraySize = PopSize;
Actual = new RootNode[PopSize];


// Now the roulette-wheel construction...

RouletteSlot = new SuperFloat[Depth+1];
NoOfAvailableIndividuals = new SuperFloat[Depth+1];
TargetNumberOfIndividuals = new int[Depth+1];

k = 0.0;
for(i=0;i<=Depth;i++)
{
        RouletteSlot[i] = L->ArbCard[i] + k;
        k = RouletteSlot[i];
        NoOfAvailableIndividuals[i] = L->ArbCard[i];
        TargetNumberOfIndividuals[i] = 0;
}


// stochastic universal sampling

Steps = 1.0/PopSize;
k = k*Steps;
SelectedSlot = SpinningWheel.rand();
SelectedSlot = k * SelectedSlot;
for(i=0;i<PopSize;i++)
{
        j = Depth-1;
        while(j>=0 && RouletteSlot[j]>SelectedSlot)
                j--;

        if(j<=0 && RouletteSlot[0]>SelectedSlot)
                TargetNumberOfIndividuals[0]++;
        else
                TargetNumberOfIndividuals[j+1]++;

        SelectedSlot = SelectedSlot + k;
```

```
        }

        // Production of Individuals

        i = 0;
        while(i<PopSize)
        {
                TempTree = new RootNode(L, Depth, this);
                if(TargetNumberOfIndividuals[TempTree->DerivationDepth]>0)
                {
                        if(NoOfAvailableIndividuals[TempTree->DerivationDepth].Mant>0)
                        {
                                j=0;
                                while(j<i && !(Actual[j]==*TempTree))
                                        j++;

                                if(Actual[j]==*TempTree)
                                        delete TempTree;
                                else
                                {
                                        Actual[i] = *TempTree;
                                        i++;
                                        TargetNumberOfIndividuals[TempTree->DerivationDepth]--;
                                        NoOfAvailableIndividuals[TempTree->DerivationDepth]--;
                                }
                        }
                        else
                        {
                                Actual[i] = *TempTree;
                                i++;
                                TargetNumberOfIndividuals[TempTree->DerivationDepth]--;
                        }
                }
                else
                        delete TempTree;
        }

        delete [] RouletteSlot;

        GenerateVirtualPop();
        ParentsAvailable = NO;
        AdjustedFitnessSum = 0;

        MatingList = new MatingListEntry[VirtualPopSize];
        MatingListArraySize = VirtualPopSize;
        MatingListNoOfEntries = VirtualPopSize;

        cout << "done..." << endl;

        return(TRUE);
}


/*--------------------------------------------------------
  void Population::GenerateVirtualPop()

  Generating a virtual population (pointers to the trees
  of the real population), not including the subtrees
  --------------------------------------------------------
*/
void Population::GenerateVirtualPop()
{
        int i;

        SmallestSubTreeOfVirtualPop = 0;

        if(VirtualPopArraySize<ActualPopSize)
```

```
                ReSizeVirtualPop(ActualPopSize,NO);
        else
                VirtualPopSize = ActualPopSize;

        for(i=0;i<ActualPopSize;i++)
                Virtual[i] = Actual+i;
}


/*--------------------------------------------------------
  void Population::GenerateVirtualPopInclSubTrees(unsigned int MinDepth)

  Generating a virtual population (pointers to the trees
  of the real population), including all subtrees larger
  than MinDepth.
  --------------------------------------------------------
*/
void Population::GenerateVirtualPopInclSubTrees(unsigned int MinDepth)
{

        int i;

        VirtualPopSize = 0;
        SmallestSubTreeOfVirtualPop = MinDepth;

        for(i=0;i<ActualPopSize;i++)
                Actual[i].AddCompleteSubTreesTo(this,MinDepth);
}


/*--------------------------------------------------------
  void Population::AddTreeToVirtualPop(StartNode* Tree)

  Adding a tree to the virtual population.
  --------------------------------------------------------
*/
void Population::AddTreeToVirtualPop(StartNode* Tree)
{
        if(Tree!=NULL)
        {
                if(VirtualPopSize>=VirtualPopArraySize)
                        ReSizeVirtualPop(VirtualPopSize+1,YES);
                else
                        VirtualPopSize++;

                Virtual[VirtualPopSize-1] = Tree;
        }
}


/*--------------------------------------------------------
  void Population::ReSizeActualPop(int NewSize)

  Resizing the actual population array in steps of the
  current size of the array, not saving the content
  of the old array.
  --------------------------------------------------------
*/
void Population::ReSizeActualPop(int NewSize)
{
        RootNode* NewActualPop;
        int NewArraySize;
        int SizingSteps;

        if(NewSize>ActualPopSize)
        {
                if(!ActualPopSize)
```

```
                        SizingSteps = NewSize;
                else
                        SizingSteps = ActualPopSize;

                NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;

                NewActualPop = new RootNode[NewArraySize];
                delete [] Actual;

                Actual = NewActualPop;
                ActualPopArraySize = NewArraySize;
        }
        ActualPopSize = NewSize;
}



/*--------------------------------------------------------
  void Population::ReSizeVirtualPop(int NewSize, int SaveOld)

  Resizing the virtual population array in steps of the
  current size of the Array. Depending on SaveOld, the content
  of the old virtual population array is saved or not.
  --------------------------------------------------------
*/
void Population::ReSizeVirtualPop(int NewSize, int SaveOld)
{
        StartNode** NewVirtualPop;
        int NewArraySize;
        int SizingSteps;
        int i;

        if(NewSize>VirtualPopSize)
        {
                if(!VirtualPopSize)
                {
                        if(!ActualPopSize)
                                SizingSteps = NewSize;
                        else
                                SizingSteps = ActualPopSize;
                }
                else
                        SizingSteps = VirtualPopSize;

                NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;
                NewVirtualPop = new StartNode*[NewArraySize];

                if(SaveOld)
                        for(i=0;i<VirtualPopSize;i++)
                                NewVirtualPop[i] = Virtual[i];

                delete [] Virtual;
                Virtual = NewVirtualPop;
                VirtualPopArraySize = NewArraySize;
        }
        VirtualPopSize = NewSize;
}


/*--------------------------------------------------------
  void Population::ReSizeMatingList(int NewSize)

  Resizing the MatingList-array in steps of the
  current size of the array.
  --------------------------------------------------------
*/
void Population::ReSizeMatingList(int NewSize)
```

```
{
        MatingListEntry* NewMatingList;
        int NewArraySize;
        int SizingSteps;

        if(NewSize>MatingListArraySize)
        {
                if(!MatingListNoOfEntries)
                        SizingSteps = NewSize;
                else
                        SizingSteps = MatingListNoOfEntries;

                NewArraySize = (NewSize/SizingSteps+1)*SizingSteps;

                NewMatingList = new MatingListEntry [NewArraySize];
                delete [] MatingList;

                MatingList = NewMatingList;
                MatingListArraySize = NewArraySize;
        }
        MatingListNoOfEntries = NewSize;
}


/*--------------------------------------------------------
  void Population::ComputeBestAndWorstIndividual(int goal)

  Computing the best and worst individual of this
  population depending on the goal (MAXIMIZE or MINIMIZE)
  --------------------------------------------------------
*/
void Population::ComputeBestAndWorstIndividual(int goal)
{
        int i;

        if(VirtualPopSize!=0)
        {
                BestIndividual = Virtual[0];
                WorstIndividual = BestIndividual;

                if(goal==MINIMIZE)
                {
                        for(i=0;i<VirtualPopSize;i++)
                        {
                                if(Virtual[i]->RawFitness<BestIndividual->RawFitness)
                                        BestIndividual = Virtual[i];

                                if(Virtual[i]->RawFitness>WorstIndividual->RawFitness)
                                        WorstIndividual = Virtual[i];
                        }
                }

                else

                {
                        for(i=0;i<VirtualPopSize;i++)
                        {
                                if(Virtual[i]->RawFitness>BestIndividual->RawFitness)
                                        BestIndividual = Virtual[i];

                                if(Virtual[i]->RawFitness<WorstIndividual->RawFitness)
                                        WorstIndividual = Virtual[i];
                        }
                }
        }
}
```

```
/*-------------------------------------------------------
  Population::void StandardizeWith(float BestRawFitness, int goal)

  Computing the StandardizedFitness for each individual of
  the virtual population, considering the goal (MINIMIZE
  or MAXIMIZE).

  Better individuals get a lower fitness value and the
  lowest possible standardized fitness value of zero is
  assigned to  the individual with RawFitness ==
  BestRawFitness.
  -------------------------------------------------------
*/
void Population::StandardizeWith(float BestRawFitness, int goal)
{
        int i;

        if(goal==MINIMIZE)
        {
                for(i=0;i<VirtualPopSize;i++)
                        Virtual[i]->StandardizedFitness =
                        Virtual[i]->RawFitness - BestRawFitness;
        }
        else
        {
                for(i=0;i<VirtualPopSize;i++)
                        Virtual[i]->StandardizedFitness =
                        BestRawFitness - Virtual[i]->RawFitness;
        }
}


/*-------------------------------------------------------
  void Population::ApplyRawFitness()

  Computing the raw fitness value (objective function value)
  for all trees in the virtual population.
  -------------------------------------------------------
*/
void Population::ApplyRawFitness()
{
        int i;

        for(i=0;i<VirtualPopSize;i++)
                Virtual[i]->RawFitness = Virtual[i]->ObjFuncVal();
}


/*-------------------------------------------------------
  void Population::InvertFitnessValues()

  Computing the AdjustedFitness for every tree in the
  virtual population, in a way that better trees achieve
  a higher adjusted fitness value. The worst tree in the
  population gets a fitness vaule of zero.
  The sum over the adjusted fitness values is stored in
  Population::AdjustedFitnessSum

  Alternative to Farm::AdjustFitnessValues()

  StandardizedFitness must have been already computed.
  ------------------------------------------------------
*/
void Population::InvertFitnessValues()
{
        int i;
```

```
        AdjustedFitnessSum = 0.0;
        for(i=0;i<VirtualPopSize;i++)
        {
                Virtual[i]->AdjustedFitness =
                WorstIndividual->StandardizedFitness -
                Virtual[i]->StandardizedFitness;

                AdjustedFitnessSum+=
                Virtual[i]->AdjustedFitness;
        }
}


/*---------------------------------------------------------
  void Population::NormalizeFitnessValues()

  Computing the NormalizedFitness for every tree in the
  virtual population, in a way that the sum over all
  fitness values equals 1.

  If the AdjustedFitnessSum is zero,
  every individual receives a NormalizedFitness value
  of 1/PopSize.

  AdjustedFitness and AdjustedFitnessSum must have been
  already computed.
  -------------------------------------------------------
*/

void Population::NormalizeFitnessValues()
{
        int i;

        if(AdjustedFitnessSum!=0)
                for(i=0;i<VirtualPopSize;i++)
                        Virtual[i]->NormalizedFitness =
                        Virtual[i]->AdjustedFitness /
                        AdjustedFitnessSum;
        else
                for(i=0;i<VirtualPopSize;i++)
                        Virtual[i]->NormalizedFitness =
                        1.0/VirtualPopSize;
}


/*---------------------------------------------------------
  void Population::PropSelect()

  Computing a target sampling rate for every tree
  of the virtual population in proportion to the stored
  normalized fitness values.

  Alternative to LinearRankSelect()

  NormalizedFitness must have been already computed.
  -------------------------------------------------------
*/
void Population::PropSelect()
{
        int i;
        float AverageFitness;

        AverageFitness = 1.0 / VirtualPopSize;

        for(i=0;i<VirtualPopSize;i++)
                Virtual[i]->TargSamplRate =
```

```
                Virtual[i]->NormalizedFitness / AverageFitness;
}


/*-------------------------------------------------------
  void Population::LinearRankSelect(float amin)

  Calculating a target sampling rate for every tree
  of the virtual population according to its rank.

  amin gives the target sampling rate for the
  worst individual in the virtual population and
  can be used for regulating the selection pressure.

  0 <= amin <= 1

  amin = 0 ... high selection pressure
  amin = 1 ... no meaningful selection at all

  Alternative to PropSelect()
  -------------------------------------------------------
*/
void Population::LinearRankSelect(float amin)
{
        int i;
        int j;
        int Worst;
        float amax;
        StartNode* Temp;
        StartNode** Ranked;

        if(amin<0.0)
                amin = 0.0;

        if(amin>1.0)
                amin = 1.0;

        amax = 2.0 - amin;
        Ranked = new StartNode*[VirtualPopSize];

        // ranking the individuals...

        for(i=0;i<VirtualPopSize;i++)
                Ranked[i] = Virtual[i];

        for(i=0;i<VirtualPopSize-1;i++)
        {
                Worst = i;

                for(j=i+1;j<VirtualPopSize;j++)
                        if(Ranked[j]->NormalizedFitness<
                        Ranked[Worst]->NormalizedFitness)
                                Worst = j;

                Temp = Ranked[i];
                Ranked[i] = Ranked[Worst];
                Ranked[Worst] = Temp;
        }

        // .. and computing the tsr according to the rank

        if(VirtualPopSize>1)
                for(i=0;i<VirtualPopSize;i++)
                        Ranked[i]->TargSamplRate =
                        amin + (amax - amin)*i/(VirtualPopSize-1);
        else
                if(VirtualPopSize>0)
```

```
                        Ranked[0]->TargSamplRate = 1.0;

        delete [] Ranked;
}


/*-------------------------------------------------------
  void Population::StochSamplWithRepl(int n)

  Fills the Partner1 entries of the MatingList.

  Selection method is the "roulette-wheel-method".
  The size of the roulette-wheel-slots depends on
  the stored target sampling rate (TargSamplRate),
  which must have been calculated before.
  -------------------------------------------------------
*/
void Population::StochSamplWithRepl(int n)
{

        int i;
        int j;
        float* RouletteSlot;
        float k;
        float SelectedSlot;
        static Uniran SpinningWheel(RANDSAMPLSEED);

        if(n<1)
        {
                n = VirtualPopSize;
                cerr << "Error! Stochastic sampling with replacement called with n=" << n;
                cerr << "." << endl << "n will be ";
                cerr << VirtualPopSize << " (current ";
                cerr << "population size) instead." << endl;
        }

        if(n>MatingListArraySize)
                ReSizeMatingList(n);
        else
                MatingListNoOfEntries = n;

        RouletteSlot = new float[VirtualPopSize];

        k = 0.0;
        for(i=0;i<VirtualPopSize;i++)
        {
                RouletteSlot[i] =
                Virtual[i]->TargSamplRate + k;
                k = RouletteSlot[i];
        }

        for(i=0;i<n;i++)
        {

                SelectedSlot = SpinningWheel.rand()*k;

                j = VirtualPopSize-2;
                while(j>=0 && RouletteSlot[j]>SelectedSlot)
                        j--;

                if(j<=0 && RouletteSlot[0]>SelectedSlot)
                        MatingList[i].Partner1 = Virtual[0];
                else
                        MatingList[i].Partner1 = Virtual[j+1];
        }

        delete [] RouletteSlot;
```

```
}


/*-----------------------------------------------------
  void Population::StochUnivSampl(int n)

  Fills the Partner1 entries of the MatingList
  with treepointers,
  refering to randomly selected trees with the
  "stochastic universal sampling method", based
  on the target sampling reate (TargSamplRate).

  The array is created at runtime. If n exceedes
  the largest array so far (ArraySize), the old
  one is deleted, and a new one created.
  -----------------------------------------------------
*/
void Population::StochUnivSampl(int n)
{

        int i;
        int j;
        float* RouletteSlot;
        float k;
        float SelectedSlot;
        static Uniran SpinningWheel(STOCHUNIVSAMPLSEED);

        if(n<1)
        {
                n = VirtualPopSize;
                cerr << "Error! Stochastic universal sampling with replacement
                cerr << called with n=" << n;
                cerr << "." << endl << "n will be ";
                cerr << VirtualPopSize << " (current ";
        cerr << "population size) instead." << endl;
        }

        if(n>MatingListArraySize)
                ReSizeMatingList(n);
        else
                MatingListNoOfEntries = n;

        RouletteSlot = new float[VirtualPopSize];

        k = 0.0;
        for(i=0;i<VirtualPopSize;i++)
        {
                RouletteSlot[i] =
                Virtual[i]->TargSamplRate + k;
                k = RouletteSlot[i];
        }

        k/= n;
        SelectedSlot = SpinningWheel.rand()*k;

        for(i=0;i<n;i++)
        {
                j = VirtualPopSize-2;
                while(j>=0 && RouletteSlot[j]>SelectedSlot)
                        j--;

                if(j<=0 && RouletteSlot[0]>SelectedSlot)
                        MatingList[i].Partner1 = Virtual[0];
                else
                        MatingList[i].Partner1 = Virtual[j+1];
```

```
		SelectedSlot+=k;
	}

	delete [] RouletteSlot;
}


/*-------------------------------------------------------
  void Population::RandomMate()

  Selects a Partner2 for every Partner1 in the
  MatingList randomly.
  -------------------------------------------------------
*/
void Population::RandomMate()
{
	int i;
	static Uniran Mating(MATINGDICESEED);

	for(i=0;i<MatingListNoOfEntries;i++)
		MatingList[i].Partner2 =
		MatingList[Mating.dice(MatingListNoOfEntries)].Partner1;
}


/*-------------------------------------------------------
  void Population::RandomPermutationMate()

  Selects a Partner2 for every Partner1 in the
  MatingList by random permutation
  -------------------------------------------------------
*/
void Population::RandomPermutationMate()
{
	int i;
	int j;
	StartNode* Temp;
	static Uniran Mating(PERMUTSEED);

	for(i=0;i<MatingListNoOfEntries;i++)
		MatingList[i].Partner2 =
		MatingList[i].Partner1;

	for(i=0;i<MatingListNoOfEntries-1;i++)
	{
		j = i + Mating.dice(MatingListNoOfEntries-i);
		Temp = MatingList[i].Partner2;
		MatingList[i].Partner2 = MatingList[j].Partner2;
		MatingList[j].Partner2 = Temp;
	}
}



/*=========================================================
  And now the Farm....
  =========================================================
*/


/*-------------------------------------------------------
  Farm::Farm()

  Empty farm constructor.
  -------------------------------------------------------
*/
Farm::Farm()
```

```
{
        History = 1;
        ThePopulation = new Population[2];
        PopulationIndex = 0;
        Current = ThePopulation;
        MaxDerivDepth = 16;
        UnifProb = NONUNIF;
        SetDefaultMethods();
}


/*--------------------------------------------------------
  Farm::Farm(int psize, int hist, int uniform, char* filename, int mdd)

  Constructor for new farm.

  psize .... gives the initital (actual) population size
  hist  .... gives the number of cached old populations
  uniform .. initial population NONUNIF, SIZEUNIF or EXACTUNIF
  filename   name of language definition file (BNF format)
  mdd ...... maximium derivation depth of the smallest s
             subtree in the virtual population
  --------------------------------------------------------
*/
Farm::Farm(int psize, int hist, int uniform, char* filename, int mdd)
{
        int i;

        History = ( (hist<1) ? 1 : hist );
        mdd = ( (mdd<1) ? 1 : mdd);

        i = L.Load(filename);

        if(!i)
        {
                cout << "ERROR while loading language definition file!" << endl << endl;
                History = 1;
                ThePopulation = new Population[2];
                PopulationIndex = 0;
                Current = ThePopulation;
                MaxDerivDepth = 16;
                UnifProb = NONUNIF;
                SetDefaultMethods();
        }
        else
        {
                PopulationIndex = 0;
                MaxDerivDepth = mdd;
                if(uniform==EXACTUNIF)
                {
                        cerr << "Exact uniform initialization not yet implemented.";
                        cerr << endl << "Uniform size heuristic initialisation ";
                        cerr << "instead." << endl;
                        UnifProb = SIZEUNIF;
                }
                else
                        UnifProb = uniform;

                ThePopulation =
                new Population[History+1](&L,psize,MaxDerivDepth);

                if(UnifProb == SIZEUNIF)
                        ThePopulation[0].
                        GenerateAlmostUniformActualPop(&L,psize,MaxDerivDepth);

                Current = ThePopulation;
                SetDefaultMethods();
```

```
        }
}


/*-------------------------------------------------------
  Farm::Farm(char* FileName)

  Constructor for loading an old farm, saved to disk
  by  Farm::Save(char* FileName).

  Selection, crossover etc methods are not part
  of the file and set to default values.
  -------------------------------------------------------
*/
Farm::Farm(char* FileName)
{
        int i;
        int j;
        int k;
        int x;
        ifstream InputFile;
        char CurrentCharacter;
        int LoadingOK;
        char* StringBuffer;
        char* FloatingNumberBuffer;
        char* NewStringBuffer;
        char** DummyEndPtr;
        int StringBufferSize;
        int Best;
        int Worst;
        int FileFormat;

        StringBuffer = new char[CHRBUFSIZE];
        StringBufferSize = CHRBUFSIZE;
        FloatingNumberBuffer = new char[LOADFLPOINTBFF];

        ThePopulation = NULL;
        Current = NULL;
        SetDefaultMethods();
        LoadingOK = TRUE;

        if(FileName==NULL)
        {
                LoadingOK = FALSE;
                cerr << "Error loading farm! No filename!" << endl;
        }

        if(LoadingOK)
        {
                i = L.Load(FileName);
                if(!i)
                {
                        LoadingOK = FALSE;
                        cerr << "Invalid File Format!" << endl;
                }
        }

        if(LoadingOK)
        {
                InputFile.open(FileName);

                if(!InputFile)
                {
                        LoadingOK = FALSE;
                        cerr << "Error loading farm! Invalid filename!" << endl;
                }
        }
```

```
if(LoadingOK)
{

        // LOADING FILE HEADER PARAMETERS:
        // FileFormat, History, UnifProb and MaxDerivDepth

        for(j=0;j<4;j++)
        {
                i = 0;
                InputFile >> CurrentCharacter;
                FloatingNumberBuffer[i] = '\0';
                while(!InputFile.eof() && CurrentCharacter!='#')
                {
                        if(i<LOADFLPOINTBFF)
                        {
                                FloatingNumberBuffer[i] = CurrentCharacter;
                                i++;
                                FloatingNumberBuffer[i] = '\0';
                        }
                        InputFile >> CurrentCharacter;
                }

                if(CurrentCharacter=='#')
                {
                        switch(j)
                        {
                                case 0:
                                        FileFormat =
                                        int(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                case 1:
                                        History =
                                        int(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;
                                case 2:
                                        UnifProb =
                                        int(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;
                                case 3:
                                        MaxDerivDepth =
                                        int(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;
                        }
                }
                else
                {
                        LoadingOK = FALSE;
                        cerr << "Invalid file format!" << endl;
                        break;
                }
        }
}

if(LoadingOK)
{
        k = History;
        ThePopulation = new Population[History+1];
```

```
                Current = ThePopulation+History;
                PopulationIndex = History;
        }

        // LOADING THE POPULATIONS

        while(k>=0 && LoadingOK)
        {


                // SEARCHING START OF NEXT ACTUAL POPULATION

                while(!InputFile.eof() && CurrentCharacter!='P')
                        InputFile >> CurrentCharacter;

                LoadingOK = (CurrentCharacter=='P');

                if(!LoadingOK)
                        break;


                // LOADING POPULATION HEADER PARAMETERS:
                // ActualPopSize, SmallestSubTreeOfVirtualPop, ParentsAvailable
                // BestIndividual, WorstIndividual, AdjustedFitnessSum

                for(j=0;j<6;j++)
                {
                        i = 0;
                        InputFile >> CurrentCharacter;
                        FloatingNumberBuffer[i] = '\0';
                        while(!InputFile.eof() && CurrentCharacter!='#')
                        {
                                if(i<LOADFLPOINTBFF)
                                {
                                        FloatingNumberBuffer[i] = CurrentCharacter;
                                        i++;
                                        FloatingNumberBuffer[i] = '\0';
                                }
                                InputFile >> CurrentCharacter;
                        }

                        if(CurrentCharacter=='#')
                        {
                                switch(j)
                                {
                                        case 0:
                                                ThePopulation[k].ActualPopSize =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;

                                        case 1:
                                                ThePopulation[k].
                                                SmallestSubTreeOfVirtualPop =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;

                                        case 2:
                                                ThePopulation[k].ParentsAvailable =
                                                int(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;
```

```
                    case 3:
                            Best =
                            int(strtod(FloatingNumberBuffer,
                            DummyEndPtr));

                            break;

                    case 4:
                            Worst =
                            int(strtod(FloatingNumberBuffer,
                            DummyEndPtr));

                            break;

                    case 5:
                            ThePopulation[k].AdjustedFitnessSum =
                            strtod(FloatingNumberBuffer,
                            DummyEndPtr);
            }
    }
    else
    {
            LoadingOK = FALSE;
            cerr << "Invalid file format!" << endl;
            break;
    }
}

if(!LoadingOK)
        break;


// LOADING ACTUAL POPULATION

ThePopulation[k].Actual =
new RootNode[ThePopulation[k].ActualPopSize](
&L, MaxDerivDepth, ThePopulation+k);

for(i=0;i<ThePopulation[k].ActualPopSize;i++)
{


        // SEARCHING START OF NEXT ACTUAL POPULATION MEMBER

        while(!InputFile.eof() && CurrentCharacter!='!')
                InputFile >> CurrentCharacter;

        // LOADING ACTUAL POPULATION MEMBER

        j = 1;
        StringBuffer[0] = '\0';

        InputFile >> CurrentCharacter;

        while(!InputFile.eof() &&
        (CurrentCharacter=='0' ||
        CurrentCharacter=='1' ||
        CurrentCharacter=='2' ||
        CurrentCharacter=='3' ||
        CurrentCharacter=='4' ||
        CurrentCharacter=='5' ||
        CurrentCharacter=='6' ||
        CurrentCharacter=='7' ||
        CurrentCharacter=='8' ||
        CurrentCharacter=='9' ||
        CurrentCharacter=='.'))
        {
```

```
        if(j==StringBufferSize)
        {
                NewStringBuffer =
                new char[StringBufferSize + CHRBUFSIZE];
                for(k=0;k<StringBufferSize;k++)
                        NewStringBuffer[k] =
                        StringBuffer[k];

                delete [] StringBuffer;
                StringBuffer = NewStringBuffer;
                StringBufferSize+=  CHRBUFSIZE;
        }

        StringBuffer[j-1] = CurrentCharacter;
        StringBuffer[j] = '\0';
        j++;
        InputFile >> CurrentCharacter;
}

LoadingOK =
ThePopulation[k].Actual[i].Set(StringBuffer);

if(!LoadingOK)
        break;


// LOADING PARAMETERS PARENT1, PARENT2, MUTATED

for(j=0;j<3;j++)
{
        x = 0;
        InputFile >> CurrentCharacter;
        FloatingNumberBuffer[x] = '\0';
        while(!InputFile.eof() && CurrentCharacter!='#')
        {
                if(k<LOADFLPOINTBFF)
                {
                        FloatingNumberBuffer[x] = CurrentCharacter;
                        x++;
                        FloatingNumberBuffer[x] = '\0';
                }
                InputFile >> CurrentCharacter;
        }

        if(CurrentCharacter=='#')
        {
                switch(j)
                {
                        case 0:
                                // RELATIVE ADRESS OF PARENT1!!!
                                ThePopulation[k].Actual[i].
                                Parent1+=
                                int(strtod(FloatingNumberBuffer,
                                DummyEndPtr));

                                break;

                        case 1:
                                // RELATIVE ADRESS OF PARENT2!!!
                                ThePopulation[k].Actual[i].
                                Parent2+=
                                int(strtod(FloatingNumberBuffer,
                                DummyEndPtr));

                                break;

                        case 2:
```

```
                                        ThePopulation[k].Actual[i].
                                        Mutated =
                                        int(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;
                        }
                }
                else
                {
                        LoadingOK = FALSE;
                        cerr << "Invalid file format!" << endl;
                        break;
                }
        }
}

if(!LoadingOK)
        break;


// CONVERTING RELATIVE ADRESSES OF PARENT1 AND PARENT2 TO ABSOLUTE

if(k<History)
{
        for(i=0;i<ThePopulation[k+1].ActualPopSize;i++)
        {
                ThePopulation[k+1].Actual[i].Parent1 =
                ThePopulation[k].Actual +
                int(int(ThePopulation[k+1].Actual[i].Parent1)/
                sizeof(StartNode)) - 1;

                ThePopulation[k+1].Actual[i].Parent2 =
                ThePopulation[k].Actual +
                int(int(ThePopulation[k+1].Actual[i].Parent2)/
                sizeof(StartNode)) - 1;
        }
}

// BUILDING UP VIRTUAL POPULATION

if(ThePopulation[k].SmallestSubTreeOfVirtualPop == 0)
        ThePopulation[k].GenerateVirtualPop();
else
        ThePopulation[k].GenerateVirtualPopInclSubTrees(
        ThePopulation[k].SmallestSubTreeOfVirtualPop);


if(Best==0)
        ThePopulation[k].BestIndividual = NULL;
else
        ThePopulation[k].BestIndividual =
        ThePopulation[k].Virtual[Best-1];

if(Worst==0)
        ThePopulation[k].WorstIndividual = NULL;
else
        ThePopulation[k].WorstIndividual =
        ThePopulation[k].Virtual[Worst-1];



// SEARCHING START OF VIRTUAL POPULATION

while(!InputFile.eof() && CurrentCharacter!='V')
        InputFile >> CurrentCharacter;
```

```
// LOADING VIRTUAL POPULATION PARAMETERS:
// RawFitness, StandardizedFitness, AdjustedFitness,
// NormalizedFitness, TargSamplRate

for(i=0;i<ThePopulation[k].VirtualPopSize;i++)
{


        // SEARCHING START OF NEXT VIRTUAL POPULATION MEMBER PARAMETER VECTOR

        while(!InputFile.eof() && CurrentCharacter!='!')
                InputFile >> CurrentCharacter;

        // LOADING VIRTUAL POPULATION MEMBER PARAMETER VECTOR

        for(j=0;j<5;j++)
        {
                x = 0;
                InputFile >> CurrentCharacter;
                FloatingNumberBuffer[x] = '\0';
                while(!InputFile.eof() && CurrentCharacter!='#')
                {
                        if(x<LOADFLPOINTBFF)
                        {
                                FloatingNumberBuffer[x] =
                                CurrentCharacter;

                                x++;

                                FloatingNumberBuffer[x] = '\0';
                        }
                        InputFile >> CurrentCharacter;
                }

                if(CurrentCharacter=='#')
                {
                        switch(j)
                        {
                                case 0:
                                        ThePopulation[k].
                                        Virtual[i]->RawFitness =
                                        float(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;

                                case 1:
                                        ThePopulation[k].
                                        Virtual[i]->StandardizedFitness =
                                        float(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;

                                case 2:
                                        ThePopulation[k].
                                        Virtual[i]->AdjustedFitness =
                                        float(strtod(FloatingNumberBuffer,
                                        DummyEndPtr));

                                        break;

                                case 3:
                                        ThePopulation[k].
                                        Virtual[i]->NormalizedFitness =
                                        float(strtod(FloatingNumberBuffer,
```

```
                                                DummyEndPtr));

                                                break;

                                        case 4:
                                                ThePopulation[k].
                                                Virtual[i]->TargSamplRate =
                                                float(strtod(FloatingNumberBuffer,
                                                DummyEndPtr));

                                                break;
                                }
                        }
                        else
                        {
                                LoadingOK = FALSE;
                                cerr << "Invalid file format!" << endl;
                                break;
                        }
                }

                if(!LoadingOK)
                        break;
        }

        k--;
    }


    if(LoadingOK)
        SetDefaultMethods();
    else
    {
        cerr << "Error while loading population!" << endl;

        if(ThePopulation!=NULL)
            delete [] ThePopulation;

        History = 1;
        ThePopulation = new Population[2];

        PopulationIndex = 0;
        Current = ThePopulation;

        MaxDerivDepth = 16;
        UnifProb = NONUNIF;
        SmallestSubTreeOfVirtualPop = 0;
    }

    delete [] StringBuffer;
    delete [] FloatingNumberBuffer;
}


/*-------------------------------------------------------
  Farm::~Farm()

  Destructor:
  Deleting all population of this farm.
  -------------------------------------------------------
*/
Farm::~Farm()
{
    if(ThePopulation!=NULL)
        delete [] ThePopulation;
}
```

```
/*-------------------------------------------------------
  void Farm::StandardizeFitnessValues(int goal)

  Standardizing the fitness values of every tree of
  the current virtual population  in a way that better
  trees get a lower fitness value and the best fitness
  value in the population is 0.

  The variable goal determines, wheter the raw
  fitness is to be maximized or minimized.
  -------------------------------------------------------
*/
void Farm::StandardizeFitnessValues(int goal)
{
        if(goal!=MAXIMIZE && goal!=MINIMIZE)
        {
                cerr << "Warning! No valid goal (maximization or mimimization) defined!";
                cerr << endl << "I choose MINIMIZATION as goal." << endl;
                goal = MINIMIZE;
        }

        Current->ComputeBestAndWorstIndividual(goal);
        Current->StandardizeWith(Current->BestIndividual->RawFitness, goal);
}


/*-------------------------------------------------------
  void Farm::ApplyDynamicFitnessScaling(unsigned int n, int goal)

  Standardizing the fitness values of every tree in
  a way that better trees get a lower fitness value
  and the best fitness value in the last n
  populations is 0.

  n==0 ... just this population, (as StandardizeFitnessValues)
  n==1 ... this and last population
  n==2 ... this and the last and the one before last
  ...

  The parameter goal determines, wheter the raw
  fitness ist to be maximized or minimized.
  -----------------------------------------------------
*/
void Farm::ApplyDynamicFitnessScaling(unsigned int n, int goal)
{
        int i;
        int j;
        float BestFitness;

        if(goal!=MAXIMIZE && goal!=MINIMIZE)
        {
                cerr << "Warning! No valid goal (maximization or mimimization) defined!";
                cerr << endl << "I choose MINIMIZATION as goal." << endl;
                goal = MINIMIZE;
        }

        if(n>History)
        {
                cerr << "Warning! Dynamic fitness scaling over the last " << n;
                cerr << " populations can not be performed, "<< endl;
                cerr << "because only the last ";
                cerr << History << " populations are stored." << endl;
                cerr << "I therefore perform a dynamic fitness scaling over the ";
                cerr << "last " << History << " populations." << endl << endl;
                n = History;
        }
```

```
        Current->ComputeBestAndWorstIndividual(goal);

        j = PopulationIndex;
        BestFitness = Current->BestIndividual->RawFitness;
        i = 0;

        while(i<n)
        {
                if(goal==MINIMIZE)
                {
                        if(ThePopulation[j].BestIndividual->RawFitness<BestFitness)
                                BestFitness =
                                ThePopulation[j].BestIndividual->RawFitness;
                }
                else
                {
                        if(ThePopulation[j].BestIndividual->RawFitness>BestFitness)
                                BestFitness =
                                ThePopulation[j].BestIndividual->RawFitness;
                }

                if(!ThePopulation[j].ParentsAvailable)
                        break;

                j = (j==0 ? History : j-1);
                i++;
        }

        Current->StandardizeWith(BestFitness, goal);
}


/*----------------------------------------------------------
  void Farm::Print()

  Printing out all Trees of the current virtual population
  ----------------------------------------------------------
*/
void Farm::Print()
{
        int i;
        int NoOfSubTree;
        int NoOfMainTree;

        cout << "FARM: " << endl << endl;
        cout << "Population size: " << Current->ActualPopSize << endl;
        cout << "Virtual population size: " << Current->VirtualPopSize << endl;

        cout << endl << "POPULATION:" << endl;

        NoOfSubTree = 0;
        NoOfMainTree = 0;

        if(Current==NULL)
                cout << "Empty farm!" << endl;
        else
        {
                for(i=0;i<Current->VirtualPopSize;i++)
                {
                        if(Current->Virtual[i]->Root==Current->Virtual[i])
                        {
                                NoOfMainTree++;
                                cout << endl << "Individual " << NoOfMainTree ;
                                NoOfSubTree = 0;
                        }
                        else
```

```
                        {
                                NoOfSubTree++;
                                cout << endl << "       Subtree " << NoOfSubTree;
                        }
                        cout << " raw-fitness: ";
                        cout << Current->Virtual[i]->RawFitness;
                        cout << " normalized fitness: ";
                        cout << Current->Virtual[i]->NormalizedFitness;
                        cout << endl;

                        if(Current->Virtual[i]!=NULL)
                        {
                                if(NoOfSubTree)
                                        cout << "       ";
                                Current->Virtual[i]->Print();
                        }
                        else
                                cout << "No Individual!";
                        cout << endl;
                }
        }
}


/*------------------------------------------------------
  void Farm::NPointCrossOver(unsigned int NoOfPoints,
                             float Probability,
                             int SelectionMethod)

  Creating a new population, consisting of indivduals
  breeded out of the pairs in the MateList.

  Probabiltity gives the probability for actually
  performing a crossover on the single individual,

  SelectionMethod may be DEFAULTSELECTION or
  HEURISTICSELECTION and ist influencing the random
  selection of subtrees.
  ------------------------------------------------------
*/
void Farm::NPointCrossOver(unsigned int NoOfPoints,
                           float Probability,
                           int SelectionMethod)
{
        int i;
        int j;
        int NewPopulationIndex;
        RootNode* NewPopulation;
        RootNode TempTree;
        CrossOverParameters SelectedCrossOverParameters;
        StartNode* Tree1;
        static Uniran RandomNumber(RANDNPCROSSSEED);

        NewPopulationIndex =
        (PopulationIndex==History ? 0 : PopulationIndex+1);


        if(
        ThePopulation[NewPopulationIndex].ActualPopArraySize<
        Current->MatingListNoOfEntries)
                ThePopulation[NewPopulationIndex].
                ReSizeActualPop(Current->MatingListNoOfEntries);

        NewPopulation = ThePopulation[NewPopulationIndex].Actual;

        for(j=0;j<Current->MatingListNoOfEntries;j++)
        {
```

```
if(RandomNumber.rand()<=Probability)
{
        for(i=0;i<NoOfPoints;i++)
        {
                if(i==0)
                        Tree1 = Current->MatingList[j].Partner1;
                else
                {
                        if(NoOfPoints%2)
                        {
                                if(i%2)
                                        Tree1 =
                                        NewPopulation+j;
                                else
                                        Tree1 = &TempTree;

                        }
                        else
                        {
                                if(i%2)
                                        Tree1 = &TempTree;
                                else
                                        Tree1 =
                                        NewPopulation+j;
                        }
                }

                if(SelectionMethod == DEFAULTSELECTION)
                        SelectedCrossOverParameters =
                        SelectCrossOverPoints(Tree1,
                        Current->MatingList[j].Partner2);
                else
                        SelectedCrossOverParameters =
                        SelectCrossOverPointsUsingHeuristic(
                        Tree1,
                        Current->MatingList[j].Partner2);


                if(SelectedCrossOverParameters.FirstTree!=NULL &&
                   SelectedCrossOverParameters.CrossOverPoint!=NULL &&
                   SelectedCrossOverParameters.SecondSubTree!=NULL)
                {
                        if(NoOfPoints%2)
                        {
                                if(i%2)
                                        TempTree.CrossOver(
                                        SelectedCrossOverParameters);
                                else
                                        NewPopulation[j].CrossOver(
                                        SelectedCrossOverParameters);
                        }
                        else
                        {
                                if(i%2)
                                        NewPopulation[j].CrossOver(
                                        SelectedCrossOverParameters);
                                else
                                        TempTree.CrossOver(
                                        SelectedCrossOverParameters);
                        }

                        NewPopulation[j].Parent1 =
                        Current->MatingList[j].Partner1;

                        NewPopulation[j].Parent2 =
                        Current->MatingList[j].Partner2;
                }
```

```
                                else
                                        NewPopulation[j] =
                                        *(Current->MatingList[j].Partner1);
                        }
                }
                else
                        NewPopulation[j] =
                        *(Current->MatingList[j].Partner1);

                NewPopulation[j].Home = ThePopulation+NewPopulationIndex;
        }

        ThePopulation[NewPopulationIndex].ActualPopSize =
        Current->MatingListNoOfEntries;

        PopulationIndex = NewPopulationIndex;
        Current = ThePopulation + PopulationIndex;
        Current->Actual = NewPopulation;
        Current->ParentsAvailable = YES;

        ThePopulation[(PopulationIndex==History ? 0 : PopulationIndex+1)].
        ParentsAvailable = NO;
}


/*---------------------------------------------------------
  CrossOverParameters Farm::SelectCrossOverPoints(StartNode* TheFirstTree,
                                                 StartNode* TheSecondTree)

  Random-selection of adequate cross-over-points in
  both trees.
  ---------------------------------------------------------
*/

CrossOverParameters Farm::SelectCrossOverPoints(StartNode* TheFirstTree,
                                               StartNode* TheSecondTree)

{
        CrossOverParameters Selected;
        int NoOfSelectedSubTree1;
        int NoOfSelectedSubTree2;
        static Uniran RandomNumber(SELCROSSSEED);
        int Trial;

        Selected.FirstTree = TheFirstTree;
        Selected.CrossOverPoint = NULL;
        Selected.SecondSubTree = NULL;

        Trial = 0;
        while(Trial<8 && Selected.SecondSubTree==NULL)
        {
                Trial++;

                NoOfSelectedSubTree1 =
                RandomNumber.dice(TheFirstTree->DerivationDepth);

                if(NoOfSelectedSubTree1<0)
                        break;
                else
                {

                        Selected.CrossOverPoint =
                        TheFirstTree->SubTree(NoOfSelectedSubTree1);

                        NoOfSelectedSubTree2 =
                        RandomNumber.dice(TheSecondTree->NoOfSubTrees(
                                          Selected.CrossOverPoint->Symbol,
```

```
                                          MaxDerivDepth -
                                          TheFirstTree->DerivationDepth +
                                          Selected.CrossOverPoint->
                                          DerivationDepth));

                        if(NoOfSelectedSubTree2>=0)
                        {
                                Selected.SecondSubTree =
                                TheSecondTree->SubTree(
                                Selected.CrossOverPoint->Symbol,
                                NoOfSelectedSubTree2,
                                MaxDerivDepth -
                                TheFirstTree->DerivationDepth +
                                Selected.CrossOverPoint->
                                DerivationDepth);
                        }
                }
        }

        return(Selected);
}


/*-------------------------------------------------------
  CrossOverParameters Farm::SelectCrossOverPointsUsingHeuristic(
                                              StartNode* TheFirstTree,
                                              StartNode* TheSecondTree)

  Random-selection of adequate cross-over points
  by using SelectionHeuristic and SelectionHeuristicBasis.
  -------------------------------------------------------
*/

CrossOverParameters Farm::SelectCrossOverPointsUsingHeuristic(
                                              StartNode* TheFirstTree,
                                              StartNode* TheSecondTree)
{
        TreeNode* TheSubTree;
        SuperFloat* LargeRoulette1Slot;
        SuperFloat MaxLargeRoulette1Slot;
        double* SmallRoulette1Slot;
        double SmallRoulette1Sum;
        SuperFloat* LargeRoulette2Slot;
        SuperFloat MaxLargeRoulette2Slot;
        double* SmallRoulette2Slot;
        double SmallRoulette2Sum;
        CrossOverParameters Selected;
        int NoOfSelectedSubTree1;
        int NoOfSelectedSubTree2;
        static Uniran RandomNumber(IMPRSELCROSSSEED);
        int Trial;
        double SelectedSlot;
        int i;
        int n;

        if(L.ArbCard==NULL)
        {
                cout << "computing search space size..." << endl;
                L.ComputeArbSearchSpace(MaxDerivDepth);
                cout << "done." << endl;
        }

        SmallRoulette1Slot = new double[TheFirstTree->DerivationDepth];

        if(SelectionHeuristicBasis == SEARCHSPACESIZE)
        {
                LargeRoulette1Slot = new SuperFloat[TheFirstTree->DerivationDepth];
```

```
        MaxLargeRoulette1Slot = 0;
        for(i=0;i<TheFirstTree->DerivationDepth;i++)
        {
                TheSubTree = TheFirstTree->SubTree(i);

                LargeRoulette1Slot[i] =
                TheSubTree->Symbol->ArbCard[TheSubTree->DerivationDepth];

                if(LargeRoulette1Slot[i]>MaxLargeRoulette1Slot)
                        MaxLargeRoulette1Slot = LargeRoulette1Slot[i];
        }

        SmallRoulette1Sum = 0.0;
        if(MaxLargeRoulette1Slot.Exp>32)
        {
                for(i=0;i<TheFirstTree->DerivationDepth;i++)
                {
                        SmallRoulette1Slot[i] =
                        SelectionHeuristic(
                        LargeRoulette1Slot[i].Mant*
                        pow(10,LargeRoulette1Slot[i].Exp-
                        MaxLargeRoulette1Slot.Exp + 32)) +
                        SmallRoulette1Sum;

                        SmallRoulette1Sum = SmallRoulette1Slot[i];
                }
        }
        else
        {
                for(i=0;i<TheFirstTree->DerivationDepth;i++)
                {
                        SmallRoulette1Slot[i] =
                        SelectionHeuristic(
                        LargeRoulette1Slot[i].Mant*
                        pow(10,LargeRoulette1Slot[i].Exp)) +
                        SmallRoulette1Sum;

                        SmallRoulette1Sum = SmallRoulette1Slot[i];
                }
        }
}

else

{
        SmallRoulette1Sum = 0.0;
        for(i=0;i<TheFirstTree->DerivationDepth;i++)
        {
                SmallRoulette1Slot[i] =
                TheFirstTree->SubTree(i)->DerivationDepth;

                SmallRoulette1Slot[i] =
                SelectionHeuristic(SmallRoulette1Slot[i]) +
                SmallRoulette1Sum;

                SmallRoulette1Sum = SmallRoulette1Slot[i];
        }
}

Selected.FirstTree = TheFirstTree;
Selected.CrossOverPoint = NULL;
Selected.SecondSubTree = NULL;

Trial = 0;
while(Trial<8 && Selected.SecondSubTree==NULL)
{
```

```
Trial++;

SelectedSlot =
SmallRoulette1Sum * RandomNumber.rand();

i = TheFirstTree->DerivationDepth-2;
while(i>=0 && SmallRoulette1Slot[i]>SelectedSlot)
        i--;

if(i<=0 && SmallRoulette1Slot[0]>SelectedSlot)
        NoOfSelectedSubTree1 = 0;
else
        NoOfSelectedSubTree1 = i+1;

if(TheFirstTree->DerivationDepth==0)
        break;
else
{
        Selected.CrossOverPoint =
        TheFirstTree->SubTree(NoOfSelectedSubTree1);

        n =
        TheSecondTree->
        NoOfSubTrees(Selected.CrossOverPoint->Symbol,
        MaxDerivDepth - TheFirstTree->DerivationDepth +
        Selected.CrossOverPoint->DerivationDepth);

        if(n>0)
        {
                SmallRoulette2Slot = new double[n];

                if(SelectionHeuristicBasis == SEARCHSPACESIZE)
                {
                        LargeRoulette2Slot = new SuperFloat[n];
                        MaxLargeRoulette2Slot = 0;
                        for(i=0;i<n;i++)
                        {
                                TheSubTree =
                                TheSecondTree->SubTree(
                                Selected.CrossOverPoint->Symbol,i,
                                MaxDerivDepth -
                                TheFirstTree->DerivationDepth +
                                Selected.CrossOverPoint->
                                DerivationDepth);

                                LargeRoulette2Slot[i] =
                                TheSubTree->Symbol->
                                ArbCard[TheSubTree->DerivationDepth];

                                if(
                                LargeRoulette2Slot[i]>MaxLargeRoulette2Slot)
                                        MaxLargeRoulette2Slot =
                                        LargeRoulette2Slot[i];
                        }

                        SmallRoulette2Sum = 0.0;
                        if(MaxLargeRoulette2Slot.Exp>32)
                        {
                                for(i=0;i<n;i++)
                                {
                                        SmallRoulette2Slot[i] =
                                        SelectionHeuristic(
                                        LargeRoulette2Slot[i].Mant*
                                        pow(10,LargeRoulette2Slot[i].Exp-
                                        MaxLargeRoulette2Slot.Exp + 32)) +
                                        SmallRoulette2Sum;
```

```
                                SmallRoulette2Sum =
                                SmallRoulette2Slot[i];
                        }
                }

                else

                {
                        for(i=0;i<n;i++)
                        {
                                SmallRoulette2Slot[i] =
                                SelectionHeuristic(
                                LargeRoulette2Slot[i].Mant*
                                pow(10,LargeRoulette2Slot[i].Exp)) +
                                SmallRoulette2Sum;

                                SmallRoulette2Sum =
                                SmallRoulette2Slot[i];
                        }
                }
        }

        else

        {
                SmallRoulette2Sum = 0.0;
                for(i=0;i<n;i++)
                {
                        TheSubTree =
                        TheSecondTree->SubTree(
                        Selected.CrossOverPoint->Symbol,i,
                        MaxDerivDepth -
                        TheFirstTree->DerivationDepth +
                        Selected.CrossOverPoint->
                        DerivationDepth);

                        SmallRoulette2Slot[i] =
                        TheSubTree->DerivationDepth;

                        SmallRoulette2Slot[i] =
                        SelectionHeuristic(SmallRoulette2Slot[i]) +
                        SmallRoulette2Sum;

                        SmallRoulette2Sum = SmallRoulette2Slot[i];
                }
        }

        SelectedSlot =
        SmallRoulette2Sum * RandomNumber.rand();

        i = n-2;
        while(i>=0 && SmallRoulette2Slot[i]>SelectedSlot)
                i--;

        if(i<=0 && SmallRoulette2Slot[0]>SelectedSlot)
                NoOfSelectedSubTree2 = 0;
        else
                NoOfSelectedSubTree2 = i+1;

        Selected.SecondSubTree =
        TheSecondTree->SubTree(
        Selected.CrossOverPoint->Symbol,
        NoOfSelectedSubTree2,
        MaxDerivDepth -
        TheFirstTree->DerivationDepth +
        Selected.CrossOverPoint->
        DerivationDepth);
```

```
                                delete [] SmallRoulette2Slot;

                                if(SelectionHeuristicBasis == SEARCHSPACESIZE)
                                        delete [] LargeRoulette2Slot;
                        }
                }
        }

        delete [] SmallRoulette1Slot;

        if(SelectionHeuristicBasis == SEARCHSPACESIZE)
                delete [] LargeRoulette1Slot;

        return(Selected);
}



/*---------------------------------------------------------
  void Farm::Mutate(float Probability, int SelectionMethod)

  Exchanges with the given Probability a randomly
  selected subtree with a randomly created one.

  SelectionMethod may be DEFAULTSELECTION or
  HEURISTICSELECTION and is influencing the
  subtree selection.
  ---------------------------------------------------------
*/
void Farm::Mutate(float Probability, int SelectionMethod)
{
        int i;
        RootNode FirstTree;
        RootNode* RandomTree;
        StartNode* Parent1;
        StartNode* Parent2;
        Population* Home;
        CrossOverParameters Selected;

        static Uniran Mutation(RANDMUTSEED);
        RandomTree = NULL;

        for(i=0;i<Current->ActualPopSize;i++)
        {
                if(Mutation.rand()<=Probability)
                {
                        Parent1 = Current->Actual[i].Parent1;
                        Parent2 = Current->Actual[i].Parent2;
                        Home    = Current->Actual[i].Home;

                        FirstTree = Current->Actual[i];
                        RandomTree = new RootNode(&L,MaxDerivDepth,NULL);

                        if(SelectionMethod == DEFAULTSELECTION)
                                Selected =
                                SelectCrossOverPoints(&FirstTree, RandomTree);
                        else
                                Selected =
                                SelectCrossOverPointsUsingHeuristic(
                                &FirstTree, RandomTree);

                        if(Selected.FirstTree!=NULL &&
                        Selected.CrossOverPoint!=NULL &&
                        Selected.SecondSubTree!=NULL)
                        {
                                Current->Actual[i].CrossOver(Selected);
```

```
                                Current->Actual[i].Mutated = YES;
                                Current->Actual[i].Parent1 = Parent1;
                                Current->Actual[i].Parent2 = Parent2;
                                Current->Actual[i].Home = Home;
                        }
                }
        }

        if(RandomTree!=NULL)
                delete RandomTree;

}


/*-----------------------------------------------------------
  void Farm::SaveBestIndividual()

  Saving the best individual of the last-generation
  population at random position into the current
  population.
  -----------------------------------------------------------
*/
void Farm::SaveBestIndividual()
{
        static Uniran RandomPosition(RANDSAVEBESTSEED);
        int LastGenerationIndex;
        int SelectedRandomPosition;

        LastGenerationIndex =
        (PopulationIndex==0 ? History : PopulationIndex-1);

        if(
        Current->ActualPopSize>0 &&
        ThePopulation[LastGenerationIndex].ActualPopSize>0)
        {
                SelectedRandomPosition =
                RandomPosition.dice(Current->ActualPopSize);

                Current->Actual[SelectedRandomPosition] =
                *(ThePopulation[LastGenerationIndex].BestIndividual);
        }
}


/*-----------------------------------------------------------
  void Farm::AdjustFitnessValues()

  Computing the AdjustedFitness for every Tree
  in the current virtual population, using

  double Farm::(*SelectionHeuristic) (double)

  which has to be assigned before.

  Alternative to Current->InvertFitnessValues()
  ----------------------------------------------------------
*/
void Farm::AdjustFitnessValues()
{
        int i;

        Current->AdjustedFitnessSum = 0.0;

        for(i=0;i<Current->VirtualPopSize;i++)
        {
                Current->Virtual[i]->AdjustedFitness =
                FitnessAdjustmentFunction(Current->Virtual[i]->StandardizedFitness);
```

```
                Current->AdjustedFitnessSum+=
                Current->Virtual[i]->AdjustedFitness;
        }
}


/*-------------------------------------------
  int Farm::Save(char* FileName)

  Saves the whole Population and the Grammar
  ------------------------------------------
*/
int Farm::Save(char* FileName)
{
        ofstream OutputFile;
        int i;
        int j;
        int k;
        int CurrentGeneration;
        int LastGeneration;
        int Parent1;
        int Parent2;
        int Best;
        int Worst;

        if(Current==NULL)
        {
                cerr << "Error Farm::Save ... No current population to save!" << endl;
                return(FALSE);
        }

        if(FileName==NULL)
        {
                cerr << "Error Farm::Save ... No filename!" << endl;
                return(FALSE);
        }

        OutputFile.open(FileName);

        if(!OutputFile)
        {
                cerr << "Error Farm::Save ... Unable to create file ";
                cerr  << FileName << endl;
                return(FALSE);
        }

        // FILE HEADER

        CurrentGeneration = PopulationIndex;
        OutputFile << FILEFORMAT << '#';
        OutputFile << History << '#';
        OutputFile << UnifProb << '#';
        OutputFile << MaxDerivDepth << '#' << endl;

        for(j=0;j<=History;j++)
        {
                // ACTUAL POPULATION

                OutputFile << 'P' << endl;
                OutputFile << ThePopulation[CurrentGeneration].ActualPopSize << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].SmallestSubTreeOfVirtualPop << '#';

                OutputFile << ThePopulation[CurrentGeneration].ParentsAvailable << '#';
```

```
Best = 0;
Worst = 0;

if(ThePopulation[CurrentGeneration].BestIndividual!=NULL)
{
        for(k=ThePopulation[CurrentGeneration].VirtualPopSize-1;k>=0;k--)
        {
                if(ThePopulation[CurrentGeneration].Virtual[k] ==
                ThePopulation[CurrentGeneration].BestIndividual)
                        Best = k+1;

                if(ThePopulation[CurrentGeneration].Virtual[k] ==
                ThePopulation[CurrentGeneration].WorstIndividual)
                        Worst = k+1;
        }
}

OutputFile << Best << '#' << Worst << '#';

OutputFile << setprecision(LOADFLPOINTBFF-10) <<
ThePopulation[CurrentGeneration].AdjustedFitnessSum << "#" << endl;

LastGeneration =
(CurrentGeneration==0 ? History : CurrentGeneration-1);

for(i=0;i<ThePopulation[CurrentGeneration].ActualPopSize;i++)
{
        if(ThePopulation[CurrentGeneration].
        Actual[i].Parent1==NULL)
                Parent1 = 0;
        else
        {
                Parent1 = 0;
                k = ThePopulation[CurrentGeneration].VirtualPopSize-1;

                while(k>=0 &&
                ThePopulation[CurrentGeneration].ParentsAvailable)
                {
                        if(
                        ThePopulation[LastGeneration].Actual+k ==
                        ThePopulation[CurrentGeneration].Actual[i].Parent1)
                        {
                                Parent1 = k+1;
                                break;
                        }
                        k--;
                }
        }

        if(ThePopulation[CurrentGeneration].
        Actual[i].Parent2==NULL)
                Parent2 = 0;
        else
        {
                Parent2 = 0;
                k = ThePopulation[CurrentGeneration].VirtualPopSize-1;

                while(k>=0 &&
                ThePopulation[CurrentGeneration].ParentsAvailable)
                {
                        if(
                        ThePopulation[LastGeneration].Actual+k ==
                        ThePopulation[CurrentGeneration].Actual[i].Parent2)
                        {
                                Parent2 = k+1;
                                break;
                        }
```

```
                                k--;
                        }
                }

                OutputFile << '!' <<
                ThePopulation[CurrentGeneration].Actual[i];

                OutputFile << '#';
                OutputFile << Parent1 << '#' << Parent2 << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Actual[i].Mutated;

                OutputFile << '#' << endl;
        }

        // VIRTUAL POPULATION PARAMETERS

        OutputFile << 'V' << endl;

        for(k=0;k<ThePopulation[CurrentGeneration].VirtualPopSize;k++)
        {
                OutputFile << '!';

                OutputFile << setprecision(LOADFLPOINTBFF-10) <<
                ThePopulation[CurrentGeneration].Virtual[k]->RawFitness << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Virtual[k]->StandardizedFitness << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Virtual[k]->AdjustedFitness << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Virtual[k]->NormalizedFitness << '#';

                OutputFile <<
                ThePopulation[CurrentGeneration].Virtual[k]->TargSamplRate << '#';

                OutputFile << endl;
        }

        CurrentGeneration =
        (CurrentGeneration==0 ? History : CurrentGeneration-1);

    }

    // the grammar....

    OutputFile << 'L' << endl;
    OutputFile << L;
    OutputFile.close();
    return(TRUE);
}


/*----------------------------------------------------------
  void Farm::SetDefaultMethods()

  Setting of default-selection, crossover, etc.
  ----------------------------------------------------------
*/
void Farm::SetDefaultMethods()
{
    SelectionHeuristic = NULL;
    SelectionHeuristicBasis = TREESIZE;
    FitnessAdjustmentFunction = One_DividedBy_OnePlusX;
```

```
        Selection = PROPORTIONAL;
        LinearRankSelectParameter = 0.2;
        Sampling = STOCHASTIC_UNIVERSAL;
        if(Current==NULL)
                NoOfMates = 0;
        else
                NoOfMates = Current->ActualPopSize;

        Mating = RANDOM_PERMUTATION;
        NoOfCrossOverPoints = 1;
        CrossOverProbability = 0.85;
        MutationProbability = 0.05;
        SaveBest = TRUE;
        SmallestSubTreeOfVirtualPop = 0;
        DynamicFitnessScalingParameter = 0;
        Goal = MAXIMIZE;
        ComputeFitnessValues();
}


/*-------------------------------------------------------
  void Farm::NextStep()

  Next evolutionary step.
  -------------------------------------------------------
*/
void Farm::NextStep()
{
        if(Selection==PROPORTIONAL)
                Current->PropSelect();
        else
                Current->LinearRankSelect(LinearRankSelectParameter);

        if(Sampling==STOCHASTIC_UNIVERSAL)
                Current->StochUnivSampl(NoOfMates);
        else
                Current->StochSamplWithRepl(NoOfMates);

        if(Mating==RANDOM_PERMUTATION)
                Current->RandomPermutationMate();
        else
                Current->RandomMate();

        if(SelectionHeuristic==NULL)
                NPointCrossOver(
                NoOfCrossOverPoints,CrossOverProbability,DEFAULTSELECTION);
        else
                NPointCrossOver(
                NoOfCrossOverPoints,CrossOverProbability,HEURISTICSELECTION);

        if(SelectionHeuristic==NULL)
                Mutate(MutationProbability,DEFAULTSELECTION);
        else
                Mutate(MutationProbability,HEURISTICSELECTION);

        if(SaveBest)
                SaveBestIndividual();

        if(SmallestSubTreeOfVirtualPop==0)
                Current->GenerateVirtualPop();
        else
                Current->
                GenerateVirtualPopInclSubTrees(SmallestSubTreeOfVirtualPop);

        ComputeFitnessValues();
}
```

```
/*------------------------------------------------------
  int Farm::ComputeFitnessValues()

  Computing the FitnessValues.
  ------------------------------------------------------
*/
int Farm::ComputeFitnessValues()
{
	if(Current==NULL)
		return(FALSE);

	if(Current->ActualPopSize==0)
		return(FALSE);

	Current->ApplyRawFitness();

	ApplyDynamicFitnessScaling(DynamicFitnessScalingParameter,Goal);

	if(FitnessAdjustmentFunction!=NULL)
		AdjustFitnessValues();
	else
		Current->InvertFitnessValues();

	Current->NormalizeFitnessValues();

	return(TRUE);
}


/*------------------------------------------------------
  void Farm::SetFitnessScaling(unsigned int NewScalingParameter, int NewGoal)

  Selection of fitness-scaling method:

  NewScalingParameter = 0 ... just the current population
                      = 1 ... current and last
                      = 2 ... current, and last two...
                              etc.

  Goal: MAXIMIZE and MINIMIZE
  ------------------------------------------------------
*/
void Farm::SetFitnessScaling(unsigned int NewScalingParameter, int NewGoal)
{
	if(NewGoal!=Goal || NewScalingParameter!=DynamicFitnessScalingParameter)
	{
		if(NewGoal==MINIMIZE)
			Goal = MINIMIZE;
		else
		{
			Goal = MAXIMIZE;
			if(NewGoal!=MAXIMIZE)
			{
				cerr << "Warning! Invalid optimization goal!" << endl;
				cerr << "Fitness maximization instead." << endl;
			}
		}

		DynamicFitnessScalingParameter = NewScalingParameter;
		ApplyDynamicFitnessScaling(DynamicFitnessScalingParameter,Goal);

		if(FitnessAdjustmentFunction!=NULL)
			AdjustFitnessValues();
		else
			Current->InvertFitnessValues();
```

```
                Current->NormalizeFitnessValues();
        }
}


/*----------------------------------------------------------
  void Farm::SetFitnessAdjustment(double(AdjustmentMethod)(double))

  Setting of fitness adjustment:

  AdjustmentMethod==NULL ... no adjustment
  AdjustmentMethod!=NULL ... pointer to adjustment function
  ---------------------------------------------------------
*/
void Farm::SetFitnessAdjustment(double (*AdjustmentMethod)(double))
{
        if(AdjustmentMethod!=FitnessAdjustmentFunction)
        {
                FitnessAdjustmentFunction = AdjustmentMethod;
                if(FitnessAdjustmentFunction!=NULL)
                        AdjustFitnessValues();
                else
                        Current->InvertFitnessValues();

                Current->NormalizeFitnessValues();
        }
}

/*----------------------------------------------------------
  void Farm::SetSelection(int NewSelectionMethod)

  Setting of new selection method:
  PROPORTIONAL or LINEAR_RANK
  ---------------------------------------------------------
*/
void Farm::SetSelection(int NewSelectionMethod)
{
        if(NewSelectionMethod==LINEAR_RANK)
                Selection = LINEAR_RANK;
        else
        {
                Selection = PROPORTIONAL;

                if(NewSelectionMethod!=PROPORTIONAL)
                {
                        cerr << "Warning! Invalid selection method!" << endl;
                        cerr << "Proportional selection instead!" << endl;
                }
        }
}


/*----------------------------------------------------------
  void Farm::SetSampling(int
  NewSamplingMethod, int NewNoOfMates)

  Setting of new sampling method:
  STOCHASTIC_UNIVERSAL or STOCH_SAMPL_WITH_REPL

  NewNoOfMates gives the number of mates, and therefore
  the size of the next actual generation
  (the virtual population, possibly including subtrees,
  might be larger).
  ---------------------------------------------------------
*/
void Farm::SetSampling(int NewSamplingMethod, int NewNoOfMates)
{
```

```
        if(NewSamplingMethod==STOCH_SAMPL_WITH_REPL)
                Sampling = STOCH_SAMPL_WITH_REPL;
        else
        {
                Sampling = STOCHASTIC_UNIVERSAL;

                if(NewSamplingMethod!=STOCHASTIC_UNIVERSAL)
                {
                        cerr << "Warning! Invalid sampling method!" << endl;
                        cerr << "Stochastic universal sampling instead." << endl;
                }
        }

        if(NewNoOfMates<1)
        {
                NewNoOfMates = 1;
                cerr << "Warning! Invalid number of mates (must be >=1)!" << endl;
                cerr << "Number of mates = 1 instead, " << endl;
        }

        NoOfMates = NewNoOfMates;
}


/*---------------------------------------------------------
  void Farm::SetMating(int NewMatingMethod)

  Setting odf new mating method:
  RANDOM or RANDOM_PERMUTATION.
  ---------------------------------------------------------
*/
void Farm::SetMating(int NewMatingMethod)
{
        if(NewMatingMethod==RANDOM)
                Mating = RANDOM;
        else
        {
                Mating = RANDOM_PERMUTATION;

                if(NewMatingMethod!=RANDOM_PERMUTATION)
                {
                        cerr << "Warning! Invalid mating method!" << endl;
                        cerr << "Random permutation mating instead." << endl;
                }
        }
}


/*---------------------------------------------------------
  void Farm::SetSelectionHeuristic(double (*NewSelectionHeuristic)(double),
                                   int NewSelectionHeuristicBasis)

  NewSelectionHeuristic  ... function pointer to new heuristic
                             NULL for no heuristic
  NewSelectionHeuristicBasis ... TREESIZE or SEARCHSPACESIZE
  --------------------------------------------------------
*/
void Farm::SetSelectionHeuristic(double (*NewSelectionHeuristic)(double),
                                 int NewSelectionHeuristicBasis)
{
        if(NewSelectionHeuristicBasis!=TREESIZE &&
        NewSelectionHeuristicBasis!=SEARCHSPACESIZE)
        {
                cerr << "Warning! Invalid selection-heuristic basis!" << endl;
                cerr << "Treesize as basisi instead." << endl;
                NewSelectionHeuristicBasis = TREESIZE;
        }
```

```
        SelectionHeuristic = NewSelectionHeuristic;
        SelectionHeuristicBasis = NewSelectionHeuristicBasis;
}


/*---------------------------------------------------------
  void Farm::SetCrossOver(int NewNoOfPoints,
                          float NewProbability,)

  Setting of new cross over method:
  NewNoOfPoints ... new number of cross-over points
  NewProbability ... new cross-over-probability
  ---------------------------------------------------------
*/
void Farm::SetCrossOver(int NewNoOfPoints,
                       float NewProbability)
{
        if(NewNoOfPoints<1)
        {
                cerr << "Warning! Invalid number of cross-over points ";
                cerr << "(must be >=1!" << endl;
                cerr << "One cross-over point instead." << endl;
                NewNoOfPoints = 1;
        }

        if(NewProbability<0.0)
        {
                cerr << "Warning! Invalid cross-over probability ";
                cerr << "(must be >=0)!" << endl;
                cerr << "Zero cross-over probability instead." << endl;
                NewProbability = 0;
        }

        if(NewProbability>1.0)
        {
                cerr << "Warning! Invalid cross-over probability ";
                cerr << "(must be <=1)!" << endl;
                cerr << "Cross-over probability = 1 instead." << endl;
                NewProbability = 1;
        }

        NoOfCrossOverPoints = NewNoOfPoints;
        CrossOverProbability = NewProbability;
}


/*---------------------------------------------------------
  void Farm::SetMutation(float NewProbability)

  Setting of new mutation method:
  NewProbability ... new mutation probability
  ---------------------------------------------------------
*/
void Farm::SetMutation(float NewProbability)
{
        if(NewProbability<0.0)
        {
                cerr << "Warning! Invalid cross-over probability ";
                cerr << "(must be >=0)!" << endl;
                cerr << "Zero cross-over probability instead." << endl;
                NewProbability = 0;
        }

        if(NewProbability>1.0)
        {
                cerr << "Warning! Invalid cross-over probability ";
```

```
                cerr << "(must be <=1)!" << endl;
                cerr << "Cross-over probability = 1 instead." << endl;
                NewProbability = 1;
        }

        MutationProbability = NewProbability;
}


/*-------------------------------------------------------
  void Farm::SetSaveBest(int NewSaveBestMode)

  Save best inndividual: TRUE or FALSE)
  -------------------------------------------------------
*/
void Farm::SetSaveBest(int NewSaveBestMode)
{
        SaveBest = NewSaveBestMode;
}


/*-------------------------------------------------------
  void Farm::SetSmallestSubTreeOfVirtualPop(unsigned int NewSmallestTreeSize)

  Setting of smallest subtree to be included to
  virtual population.
  -------------------------------------------------------
*/
void Farm::SetSmallestSubTreeOfVirtualPop(unsigned int NewSmallestTreeSize)
{
        if(SmallestSubTreeOfVirtualPop!=NewSmallestTreeSize)
        {
                SmallestSubTreeOfVirtualPop = NewSmallestTreeSize;

                if(SmallestSubTreeOfVirtualPop==0)
                        Current->GenerateVirtualPop();
                else
                        Current->
                        GenerateVirtualPopInclSubTrees(SmallestSubTreeOfVirtualPop);

                ComputeFitnessValues();
        }
}


/*-------------------------------------------------------
  void Farm::SetMaxDerivDepth(int mdd)

  Setting of maximum derivation depth
  (must be >=1)
  -------------------------------------------------------
*/
void Farm::SetMaxDerivDepth(int mdd)
{
        mdd = ( (mdd<1) ? 1 : mdd);
        MaxDerivDepth = mdd;
}
```

A.1.3 Die Datei tree.cc

```
// File: tree.cc

#include "ga.h"



/*========================================================
   IMPLEMENTATION OF CLASSES

   RootNode  : public StartNode
   StartNode : public TreeNode
   TreeNode

   FORMING THE BUILDING BLOCKS FOR DERIVATION TREES
  ========================================================
*/



/*========================================================
  Class RootNode...
  ========================================================
*/


/*--------------------------------------------------------
   RootNode::RootNode()

   Constructor: Creating an empty tree with no parents and
   no home-population
  --------------------------------------------------------
*/
RootNode::RootNode()
{
        Parent1 = NULL;
        Parent2 = NULL;
        Mutated = NO;
        Home = NULL;
        Root = this;
        PredecessorNode = NULL;
}


/*--------------------------------------------------------
  RootNode::RootNode(Language* L,
                     int MaxDerivDepth,
                     Population* HomePopulation)

  Constructor: Creating a random-tree of the given
  language L with the given maxium derivation depth,
  part of the population, referred by the HomePopulation-
  pointer.
  -------------------------------------------------------
*/
RootNode::RootNode(Language* L, int MaxDerivDepth, Population* HomePopulation)
{
        int i;
        unsigned int NoOfNonTerminalNodes;
        Root = this;
        Parent1 = NULL;
        Parent2 = NULL;
        Mutated = NO;
        Home = HomePopulation;
```

```
        PredecessorNode = NULL;
        DerivationDepth = 1;
        TempIndex = 1;
        TreeLanguage = L;
        RawFitness = 0;
        StandardizedFitness = 0;
        AdjustedFitness = 0;
        NormalizedFitness = 0;
        TargSamplRate = 0;
        PhenoString = NULL;
        GenoString = NULL;
        Symbol = L->StartSymbol;
        Derivs = NULL;
        NoOfDerivSymbols = 0;
        DerivationNo = 0;
        TreeLanguage = L;

        while(TRUE)
        {
                NoOfNonTerminalNodes = 1;
                if(RandomEnhance(MaxDerivDepth, NoOfNonTerminalNodes))
                        break;
                else
                {
                        for(i=0;i<NoOfDerivSymbols;i++)
                                delete Derivs[i];

                        delete [] Derivs;
                }
        }
}


/*--------------------------------------------------------
  int RootNode::operator =(StartNode& TheOtherTree)

  New assignement operator =

  Deleting, if necessary, the complete subtreetree,
  starting from this RootNode, and building up a copy of
  the right tree.
  This is done by using the methode
  void TreeNode::EnhanceWith(TreeNode*)

  TheOtherTree is stored as parent. Must be alterd after
  assignement if not OK. Home population is home population
  of TheOtherTree. Fitness values and tsr are copied also.
  --------------------------------------------------------
*/
int RootNode::operator =(StartNode& TheOtherTree)
{
        int i;

        if(NoOfDerivSymbols>0)
        {
                for(i=0;i<NoOfDerivSymbols;i++)
                        delete Derivs[i];

                delete [] Derivs ;
        }

        Parent1 = &TheOtherTree;
        Parent2 = NULL;
        Mutated = NO;
        Home = TheOtherTree.Root->Home;
        Root = this;
        DerivationDepth = TheOtherTree.DerivationDepth;
```

```
        TempIndex = TheOtherTree.TempIndex;
        PredecessorNode = NULL;
        RawFitness = TheOtherTree.RawFitness;
        StandardizedFitness = TheOtherTree.StandardizedFitness;
        AdjustedFitness = TheOtherTree.AdjustedFitness;
        NormalizedFitness = TheOtherTree.NormalizedFitness;
        TargSamplRate = TheOtherTree.TargSamplRate;
        TreeLanguage = TheOtherTree.TreeLanguage;
        Symbol = TheOtherTree.Symbol;

        EnhanceWith(&TheOtherTree);

        return(TRUE);
}


/*---------------------------------------------------------
  int RootNode::Set(char* TheString)

  Assignement from genotype-string TheString
  (StartNode::GenoTypeString(SHORT)-representation)
  to this tree. Returning FALSE in case of an invalid
  string.
  -------------------------------------------------------
*/
int RootNode::Set(char* TheString)
{
        int EnhancementOK;
        int Position;
        RootNode NewTree;

        if(TheString==NULL)
                return(FALSE);

        NewTree.TreeLanguage = TreeLanguage;
        NewTree.Home = Home;
        NewTree.Symbol = TreeLanguage->StartSymbol;

        Position = 0;
        EnhancementOK = NewTree.EnhanceWithString(TheString,Position);

        if(EnhancementOK)
        {
                *this = NewTree;
                Parent1 = NULL;
        }
        else
                cerr << "Genotype-string error at Postion " << Position+1 << endl;

        return(EnhancementOK);
}


/*---------------------------------------------------------
  int RootNode::CrossOver(CrossOverParameters Selected)

  Deleting, if necessary, the complete existing tree,
  and building up a copy of Selected.FirstTree, until
  Selected.CrossOverPoint occurs. The copying from
  this point on is done from Selected.SecondTree.

  This is done by use of the methode
  void TreeNode::XEnhanceWith(TreeNode*, TreeNode*, TreeNode*)

  Assignement of parents and home-population included.
  -------------------------------------------------------
*/
```

```
int RootNode::CrossOver(CrossOverParameters Selected)
{
        int i;

        if(Selected.CrossOverPoint==NULL||Selected.SecondSubTree==NULL)
        {
                cerr << "ERROR while Tree::Crossover. No treepointer (NULL)!" << endl;
                return(FALSE);
        }
        else
        {
                if(Selected.CrossOverPoint->Symbol!=Selected.SecondSubTree->Symbol)
                {
                        cerr << "ERROR while StartNode::Crossover. ";
                        cerr << "Subtrees start with different symbols!" << endl;
                        return(FALSE);
                }
                else
                {
                        if(NoOfDerivSymbols>0)
                        {
                                for(i=0;i<NoOfDerivSymbols;i++)
                                        delete Derivs[i];

                                delete [] Derivs;
                        }

                        Parent1 = Selected.FirstTree;
                        Parent2 = Selected.SecondSubTree->Root;
                        Mutated = NO;
                        Root = this;
                        PredecessorNode = NULL;
                        Home =  Selected.SecondSubTree->Root->Home;
                        RawFitness = 0;
                        StandardizedFitness = 0;
                        AdjustedFitness = 0;
                        NormalizedFitness = 0;
                        TargSamplRate = 0;
                        TreeLanguage = Selected.FirstTree->TreeLanguage;
                        Symbol = TreeLanguage->StartSymbol;
                        DerivationDepth = Selected.FirstTree->DerivationDepth;
                        TempIndex = 0;

                        XEnhanceWith(Selected.FirstTree,
                                     Selected.CrossOverPoint,
                                     Selected.SecondSubTree)->
                        DerivationDepthUpdate();
                }
        }
        return(TRUE);
}



/*=========================================================
  Now the Start-Node definition...
  =========================================================
*/


/*--------------------------------------------------------
   StartNode::StartNode()

   Constructor: Creating an empty StartNode
  --------------------------------------------------------
*/
```

```
StartNode::StartNode()
{
        RawFitness = 0;
        StandardizedFitness = 0;
        AdjustedFitness = 0;
        NormalizedFitness = 0;
        TargSamplRate = 0;
        PhenoString = NULL;
        GenoString = NULL;
        DerivationDepth = 1;
}


/*-------------------------------------------------------
   StartNode::StartNode(Language* L, TreeNode* Predecessor, RootNode* R)

   Constructor: Creating an empty start node
   of language L with a link to the predecessor node
   and to the root node.
  -------------------------------------------------------
*/
StartNode::StartNode(Language* L, TreeNode* Predecessor, RootNode* R)
{
        PredecessorNode = Predecessor;
        DerivationDepth = 1;
        TempIndex = 0;
        Root = R;
        RawFitness = 0;
        StandardizedFitness = 0;
        AdjustedFitness = 0;
        NormalizedFitness = 0;
        TargSamplRate = 0;
        PhenoString = NULL;
        GenoString = NULL;
        Symbol = L->StartSymbol;
        Derivs = NULL;
        NoOfDerivSymbols = 0;
        DerivationNo = 0;
        TreeLanguage = L;
}


/*-------------------------------------------------------
  TreeNode* StartNode::SubTree(unsigned int TargetTreeNumber)

  Returning a pointer to the starting TreeNode of the
  complete sub-tree referenced by TargetTreeNumber.

  0 <= TargetTreeNumber < NoOfSubTrees()

  In case the sub-tree numered TargetTreeNumber
  does not exist, this methode returns NULL.
  -------------------------------------------------------
*/
TreeNode* StartNode::SubTree(unsigned int TargetTreeNumber)
{
        int i;
        unsigned int Offset;
        TreeNode* CurrentTreeNode;

        Offset=0;
        CurrentTreeNode = this;
        TargetTreeNumber++;

        while(CurrentTreeNode->NoOfDerivSymbols>0)
        {
                if(Offset+CurrentTreeNode->DerivationDepth==TargetTreeNumber)
```

```
                        return(CurrentTreeNode);

                i = 0;

                while(i<CurrentTreeNode->NoOfDerivSymbols)
                {
                        if(Offset +
                        CurrentTreeNode->Derivs[i]->DerivationDepth>=
                        TargetTreeNumber)
                                break;

                        Offset +=
                        CurrentTreeNode->Derivs[i]->DerivationDepth;

                        i++;
                }

                if(i==CurrentTreeNode->NoOfDerivSymbols)
                        return(NULL);
                else
                        CurrentTreeNode = CurrentTreeNode->Derivs[i];
        }
        return(NULL);
}


/*-------------------------------------------------------
  TreeNode* StartNode::SubTree(SymbTabEntry* SelectedSymbol,
                               unsigned int TargetTreeNumber,
                               unsigned int MaxDerivDepth)

  Returning a pointer to the starting TreeNode of the
  complete sub-tree referenced by SelectedSymbol,
  TargetTreeNumber and MaxDerivSymbol.

  For example: SubTree(L.Symbol("f0),3,20) is returning
  the 3rd subtree starting with symbol "f0", counting
  only subtrees with maximum derivation depth 20.

  In case the aquired subtree does not exist, this
  methode is returning NULL.
  -------------------------------------------------------
*/
TreeNode* StartNode::SubTree(SymbTabEntry* SelectedSymbol,
                             unsigned int TargetTreeNumber,
                             unsigned int MaxDerivDepth)
{
        int i;
        unsigned int Offset;
        TreeNode* CurrentTreeNode;

        Offset=0;
        CurrentTreeNode = this;
        TargetTreeNumber++;

        while(CurrentTreeNode->NoOfDerivSymbols>0)
        {
                if(
                Offset + CurrentTreeNode->TempIndex==TargetTreeNumber &&
                CurrentTreeNode->Symbol==SelectedSymbol &&
                CurrentTreeNode->DerivationDepth<=MaxDerivDepth)
                        return(CurrentTreeNode);

                i = 0;
                while(i<CurrentTreeNode->NoOfDerivSymbols)
                {
                        if(
```

```
                        Offset+CurrentTreeNode->Derivs[i]->TempIndex>=
                        TargetTreeNumber)
                                break;

                        Offset +=
                        CurrentTreeNode->Derivs[i]->TempIndex;

                        i++;
                }

                if(i==CurrentTreeNode->NoOfDerivSymbols)
                        return(NULL);
                else
                        CurrentTreeNode = CurrentTreeNode->Derivs[i];
        }
        return(NULL);
}


/*-------------------------------------------------------
  virtual int StartNode::operator =(StartNode& TheOtherTree)

  New assignement operator =

  Deleting, if necessary, the complete subtree,
  and building up a copy of the right tree. This
  is done by using the methode

  void TreeNode::EnhanceWith(TreeNode*)
  -------------------------------------------------------
*/
int StartNode::operator =(StartNode& TheOtherTree)
{
        int i;

        if(NoOfDerivSymbols>0)
        {
                for(i=0;i<NoOfDerivSymbols;i++)
                        delete Derivs[i];
                delete [] Derivs ;
        }

        RawFitness = TheOtherTree.RawFitness;
        StandardizedFitness = TheOtherTree.StandardizedFitness;
        AdjustedFitness = TheOtherTree.AdjustedFitness;
        NormalizedFitness = TheOtherTree.NormalizedFitness;
        TargSamplRate = TheOtherTree.TargSamplRate;
        TreeLanguage = TheOtherTree.TreeLanguage;
        Symbol = TheOtherTree.Symbol;
        DerivationDepth = TheOtherTree.DerivationDepth;
        TempIndex = TheOtherTree.TempIndex;

        EnhanceWith(&TheOtherTree);

        if(PredecessorNode!=NULL)
                PredecessorNode->DerivationDepthUpdate();

        return(TRUE);
}


/*-------------------------------------------------------
  void StartNode::PrintParents();

  Printing out the parents of this tree and the mutation
  status.
  -------------------------------------------------------
```

```
*/
void StartNode::PrintParents()
{
        cout << endl << endl;
        if(Root==NULL)
                cout << "No parents available." << endl << endl;
        else
        {
                if(Root->Home==NULL)
                        cout << "No parents available." << endl << endl;
                else
                {
                        if(!Root->Home->ParentsAvailable || Root->Parent1==NULL)
                                cout << "No parents available." << endl << endl;
                        else
                        {
                                if(Root!=this)
                                {
                                        cout << "Individual is part of the ";
                                        cout << "following:";
                                        cout << endl << endl;
                                        Root->Print();
                                        cout << endl << endl;
                                }
                                if(Root->Parent2==NULL)
                                {
                                        cout << "This individual has ";
                                        cout << "been cloned without ";
                                        cout << "any crossover from:";
                                        cout << endl << endl;
                                        Root->Parent1->Print();
                                        cout << endl << endl;
                                }
                                else
                                {
                                        cout << "The Parents are:";
                                        cout << endl << endl;
                                        Root->Parent1->Print();
                                        cout << endl << endl;
                                        Root->Parent2->Print();
                                        cout << endl << endl;
                                }
                        }
                }

                if(Root->Mutated)
                {
                        cout << "There was a mutation.";
                        cout << endl << endl;
                }
        }
}


/*-------------------------------------------------------
  int StartNode::operator ==(StartNode& TheOtherTree)

  New boolean operator ==

  Comparing the complete tree, including all subtrees
  -------------------------------------------------------
*/
int StartNode::operator ==(StartNode& TheOtherTree)
{
        return(CompareWith(&TheOtherTree));
}
```

```
/*------------------------------------------------------
  void StartNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                        unsigned int MinDepth)

  Adding all complete subtrees (trees starting with a
  StartNode) with a depth larger or equal MinDepth
  to the ThePopulation.
  ------------------------------------------------------
*/
void StartNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                      unsigned int MinDepth)
{
        int i;

        ThePopulation->AddTreeToVirtualPop(this);

        for(i=0;i<NoOfDerivSymbols;i++)
        {
                if(Derivs[i]->DerivationDepth>=MinDepth)
                        Derivs[i]->
                        AddCompleteSubTreesTo(ThePopulation,MinDepth);
        }
}


/*------------------------------------------------------
  void StartNode::EnhanceWith(StartNode* TheOtherNode)

  Enhancing the StartNode with the derivation given
  by TheOtherNode.
  ------------------------------------------------------
*/
void StartNode::EnhanceWith(StartNode* TheOtherNode)
{

        int i;

        RawFitness = TheOtherNode->NormalizedFitness;
        StandardizedFitness = TheOtherNode->StandardizedFitness;
        AdjustedFitness = TheOtherNode->AdjustedFitness;
        NormalizedFitness = TheOtherNode->NormalizedFitness;
        TargSamplRate = TheOtherNode->TargSamplRate;
        NoOfDerivSymbols = TheOtherNode->NoOfDerivSymbols;
        DerivationNo = TheOtherNode->DerivationNo;
        Derivs = new TreeNode*[NoOfDerivSymbols];

        for(i=0;i<NoOfDerivSymbols;i++)
        {
                if(TheOtherNode->Derivs[i]->Symbol==TreeLanguage->StartSymbol)
                        Derivs[i] =
                        new StartNode(TreeLanguage, this, Root);
                else
                        Derivs[i] =
                        new TreeNode(TheOtherNode->Derivs[i]->Symbol,
                                     TreeLanguage,this,Root);

                Derivs[i]->DerivationDepth =
                TheOtherNode->Derivs[i]->DerivationDepth;

                TempIndex = 0;

                if(TheOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                        Derivs[i]->EnhanceWith(TheOtherNode->Derivs[i]);
        }
}
```

```
/*-------------------------------------------------------
  char* StartNode::GenoTypeString(int Representation)

  Returning a char-string representation of the tree.
  There are two representations available:


  Long representation (human-readable)
  (representation==LONG)

    best explained by this example:

                        <fe>
               "("   <f1>     <fe>    ")"
                   "NOT"  "(" <f0> ")"
                              "D2"

    is represented by the stream

    4<fe>"("1<f1>"NOT"3<fe>"("1<f0>"D2"")"")"


  Short representation: (machine-readable)
  (representation==SHORT)

    Same as in the ostream&<< operator.
  -------------------------------------------------------
*/
char* StartNode::GenoTypeString(int Representation)
{
        int Position;

        if(GenoString!=NULL)
                delete [] GenoString;

        GenoString = new char[GenoTypeStringLength(Representation)+1];
        Position = 0;
        BuildUpGenoTypeStringIn(GenoString, Position, Representation);
        GenoString[Position] = '\0';
        return(GenoString);
}

/*-------------------------------------------------------
  char* StartNode::PhenoTypeString()

  Returning a char-string of the tree like a
  TreeNode::Print() - output.
  -------------------------------------------------------
*/
char* StartNode::PhenoTypeString()
{
        int Position;

        if(PhenoString!=NULL)
                delete [] PhenoString;

        PhenoString = new char[PhenoTypeStringLength()+1];
        Position = 0;
        BuildUpPhenoTypeStringIn(PhenoString, Position);
        PhenoString[Position] = '\0';
        return(PhenoString);
}
```

```
/*=======================================================
  And here comes the TreeNode.....
  =======================================================
*/


/*-------------------------------------------------------
  TreeNode::TreeNode()

  Constructor: Creating an empty TreeNode
  -------------------------------------------------------
*/
TreeNode::TreeNode()
{
        Symbol = NULL;
        Derivs = NULL;
        NoOfDerivSymbols = 0;
        TreeLanguage = NULL;
        PredecessorNode = NULL;
        DerivationDepth = 0;
        TempIndex = 0;
        Root = NULL;
        DerivationNo = 0;
}


/*-------------------------------------------------------
  TreeNode::TreeNode(SymbTabEntry* TheSymbol,
                     Language* L,
                     TreeNode* Predecessor,
                     RootNode* R)

  Constructor: Creating an TreeNode of Language L
  with Symbol TheSymbol and linking it to the
  predecessor and root node.
  -------------------------------------------------------
*/
TreeNode::TreeNode(SymbTabEntry* TheSymbol,
                  Language* L,
                   TreeNode* Predecessor,
                   RootNode* R)
{
        Symbol = TheSymbol;
        Derivs = NULL;
        NoOfDerivSymbols = 0;
        TreeLanguage = L;
        PredecessorNode = Predecessor;
        DerivationDepth = 0;
        TempIndex = 0;
        Root = R;
        DerivationNo = 0;
}


/*-------------------------------------------------------
  TreeNode::~TreeNode()

  Destructor: Deleting the complete subtree.
  -------------------------------------------------------
*/
TreeNode::~TreeNode()
{
        int i;

        if(NoOfDerivSymbols>0)
        {
                for(i=0;i<NoOfDerivSymbols;i++)
```

```
                        delete Derivs[i];

                delete [] Derivs;
        }
}


/*---------------------------------------------------------
  int TreeNode::EnhanceWithString(char* TheString, int& Position)

  Enhancing this tree with the tree given in TheString from
  Position on. TheString must have the short phenotype format.
  --------------------------------------------------------
*/
int TreeNode::EnhanceWithString(char* TheString, int& Position)
{
        int i;
        int j;
        int s;

        i = 0;
        do
        {
                switch(TheString[Position])
                {
                        case '0':
                                i = 10*i + 0;
                                break;
                        case '1':
                                i = 10*i + 1;
                                break;
                        case '2':
                                i = 10*i + 2;
                                break;
                        case '3':
                                i = 10*i + 3;
                                break;
                        case '4':
                                i = 10*i + 4;
                                break;
                        case '5':
                                i = 10*i + 5;
                                break;
                        case '6':
                                i = 10*i + 6;
                                break;
                        case '7':
                                i = 10*i + 7;
                                break;
                        case '8':
                                i = 10*i + 8;
                                break;
                        case '9':
                                i = 10*i + 9;
                                break;
                        case '.':
                                break;
                        default:
                                return(FALSE);
                }
                Position++;
        } while(TheString[Position]!='.');

        if(i>Symbol->NoOfDerivs)
                return(FALSE);

        EnhanceWith(Symbol->DerivList+(i-1));
```

```
        s = TRUE;
        for(j=0;j<NoOfDerivSymbols;j++)
        {
                if(Derivs[j]->Symbol->NoOfDerivs)
                        s = Derivs[j]->EnhanceWithString(TheString, Position);
                if(!s)
                        break;
        }
        return(s);
}


/*---------------------------------------------------------
  virtual void TreeNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                             unsigned int MinDepth)

  Adding all complete subtrees (trees starting with a
  StartNode) with a depth larger or equal MinDepth
  to the ThePopulation.
  -------------------------------------------------------
*/
void TreeNode::AddCompleteSubTreesTo(Population* ThePopulation,
                                     unsigned int MinDepth)
{
        int i;

        for(i=0;i<NoOfDerivSymbols;i++)
        {
                if(Derivs[i]->DerivationDepth>=MinDepth)
                        Derivs[i]->
                        AddCompleteSubTreesTo(ThePopulation,MinDepth);
        }
}


/*---------------------------------------------------------
  void TreeNode::EnhanceWith(ProdTabEntry* Derivation)

  Enhancing the TreeNode with the given Derivation
  --------------------------------------------------------
*/
void TreeNode::EnhanceWith(ProdTabEntry* Derivation)
{
        int i;
        DerivSymbol* CurrentDerivSymbol;

        NoOfDerivSymbols = Derivation->NoOfDerivSymbols;
        DerivationNo = int(Derivation - Symbol->DerivList);
        Derivs = new TreeNode*[NoOfDerivSymbols];

        CurrentDerivSymbol = Derivation->FirstSymbol;
        for(i=0;i<NoOfDerivSymbols;i++)
        {
                if(CurrentDerivSymbol->Symbol==TreeLanguage->StartSymbol)
                        Derivs[i] =
                        new StartNode(TreeLanguage, this, Root);
                else
                        Derivs[i] =
                        new TreeNode(CurrentDerivSymbol->Symbol,
                                     TreeLanguage, this, Root);

                CurrentDerivSymbol = CurrentDerivSymbol->NextSymbol;
        }
}
```

```
/*-------------------------------------------------------
  void TreeNode::EnhanceWith(TreeNode* TheOtherNode)

  Enhancing the TreeNode with the derivation given
  by TheOtherNode.
  -------------------------------------------------------
*/
void TreeNode::EnhanceWith(TreeNode* TheOtherNode)
{
        int i;

        NoOfDerivSymbols = TheOtherNode->NoOfDerivSymbols;
        DerivationNo = TheOtherNode->DerivationNo;
        Derivs = new TreeNode*[NoOfDerivSymbols];

        for(i=0;i<NoOfDerivSymbols;i++)
        {
                if(TheOtherNode->Derivs[i]->Symbol==TreeLanguage->StartSymbol)
                        Derivs[i] =
                        new StartNode(TreeLanguage, this, Root);
                else
                        Derivs[i] =
                        new TreeNode(TheOtherNode->Derivs[i]->Symbol,
                                     TreeLanguage, this, Root);

                Derivs[i]->DerivationDepth =
                TheOtherNode->Derivs[i]->DerivationDepth;

                TempIndex = 0;

                if(TheOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                        Derivs[i]->EnhanceWith(TheOtherNode->Derivs[i]);
        }
}


/*-------------------------------------------------------
  int TreeNode::operator =(TreeNode& TheOtherTreeNode)

  New assignement operator =

  Deleting, if necessary, the complete tree,
  and building up a copy of the right tree. This
  is done by using the methode

  void TreeNode::EnhanceWith(TreeNode*)
  -------------------------------------------------------
*/
int TreeNode::operator =(TreeNode& TheOtherTreeNode)
{
        int i;

        if(NoOfDerivSymbols>0)
        {
                for(i=0;i<NoOfDerivSymbols;i++)
                        delete Derivs[i];
                delete [] Derivs;
        }

        TreeLanguage = TheOtherTreeNode.TreeLanguage;
        DerivationDepth = TheOtherTreeNode.DerivationDepth;
        TempIndex = TheOtherTreeNode.TempIndex;
        Symbol = TheOtherTreeNode.Symbol;
        EnhanceWith(&TheOtherTreeNode);

        if(PredecessorNode!=NULL)
                PredecessorNode->DerivationDepthUpdate();
```

```
        return(TRUE);
}



/*-------------------------------------------------------
  TreeNode* TreeNode::XEnhanceWith(TreeNode* TheFirstOtherNode,
                                   TreeNode* TheCrossOverPoint,
                                   TreeNode* TheSecondOtherNode)

  Enhancing the TreeNode recursively with the derivation
  given by TheFirstOtherNode, until during this copying
  TheCrossOverPoint occurs. From this point on, the
  enhancement is continued recursively
  until the end of the tree.
  -------------------------------------------------------
*/
TreeNode* TreeNode::XEnhanceWith(TreeNode* TheFirstOtherNode,
                                 TreeNode* TheCrossOverPoint,
                                 TreeNode* TheSecondOtherNode)
{
        int i;
        TreeNode* NewCrossOverPoint;

        if(TheFirstOtherNode==TheCrossOverPoint)
        {
                NewCrossOverPoint = this;
                EnhanceWith(TheSecondOtherNode);
        }

        else

        {
                NoOfDerivSymbols = TheFirstOtherNode->NoOfDerivSymbols;
                DerivationNo = TheFirstOtherNode->DerivationNo;
                Derivs = new TreeNode*[NoOfDerivSymbols];
                NewCrossOverPoint = NULL;

                for(i=0;i<NoOfDerivSymbols;i++)
                {
                        if(
                        TheFirstOtherNode->Derivs[i]->Symbol==
                        TreeLanguage->StartSymbol)
                                Derivs[i] =
                                new StartNode(TreeLanguage, this, Root);
                        else
                                Derivs[i] =
                                new TreeNode(TheFirstOtherNode->Derivs[i]->Symbol,
                                             TreeLanguage, this, Root);

                        Derivs[i]->DerivationDepth =
                        TheFirstOtherNode->Derivs[i]->DerivationDepth;

                        Derivs[i]->TempIndex = 0;

                        if(TheFirstOtherNode->Derivs[i]->NoOfDerivSymbols!=0)
                                if(NewCrossOverPoint==NULL)
                                        NewCrossOverPoint =
                                        Derivs[i]->
                                        XEnhanceWith(TheFirstOtherNode->Derivs[i],
                                        TheCrossOverPoint, TheSecondOtherNode);
                                else
                                        Derivs[i]->
                                        XEnhanceWith(TheFirstOtherNode->Derivs[i],
                                        TheCrossOverPoint,TheSecondOtherNode);
                }
```

```
        }

        return(NewCrossOverPoint);
}


/*--------------------------------------------------------
  int TreeNode::CompareWith(TreeNode* TheOtherNode)

  Comparing the TreeNode with TheOtherNode,
  including all subtrees.
  --------------------------------------------------------
*/
int TreeNode::CompareWith(TreeNode* TheOtherNode)
{
        int i;

        if(Symbol!=TheOtherNode->Symbol ||
        NoOfDerivSymbols!=TheOtherNode->NoOfDerivSymbols)
                return(FALSE);
        else
                for(i=0;i<NoOfDerivSymbols;i++)
                        if(!Derivs[i]->CompareWith(TheOtherNode->Derivs[i]))
                                return(FALSE);
        return(TRUE);
}


/*--------------------------------------------------------
  int TreeNode::RandomEnhance(unsigned int MaxDerivDepth,
                              unsigned int &NoOfNonTerminalNodes)

  Enhancing the TreeNode with a randomly selected
  derivations out of the related production table
  entry.

  Returning FALSE in case the random enhancement
  has failed because of exceeding the given maximum
  derivation depth.

  NoOfNonTerminalNodes is kind of a global counter,
  countig the non-terminal symbols so far.
  --------------------------------------------------------
*/
int TreeNode::RandomEnhance(unsigned int MaxDerivDepth,
                            unsigned int &NoOfNonTerminalNodes)
{
        int i;

        static Uniran Random(RANDTREESEED);

        if(Symbol->DerivList!=NULL)
        {
                EnhanceWith(Symbol->
                DerivList+Random.dice(Symbol->NoOfDerivs));

                DerivationDepth = (NoOfDerivSymbols>0);
                for(i=0;i<NoOfDerivSymbols;i++)
                {
                        if(Derivs[i]->Symbol->NoOfDerivs>0)
                        {
                                NoOfNonTerminalNodes++;
                                if(NoOfNonTerminalNodes>MaxDerivDepth)
                                        return(FALSE);
                                else
                                {
                                        if(!Derivs[i]->RandomEnhance(
```

```
                                        MaxDerivDepth, NoOfNonTerminalNodes))
                                                return(FALSE);

                                        DerivationDepth =
                                        DerivationDepth +
                                        Derivs[i]->DerivationDepth;
                                }
                        }
                }
        }
        return(TRUE);
}


/*--------------------------------------------------------
  void TreeNode::Print()

  Printing, if no further derivations available, the
  terminal symbol, otherwise the Print()-methods of
  the derivation-nodes are called.

  This leads to a printout of the phenotype-string.

  cout << PhenoTypeString(); would have the same effect,
  but had the disadvantage of needing more CPU-time
  and memory.
  --------------------------------------------------------
*/
void TreeNode::Print()
{
        int i;
        if(NoOfDerivSymbols==0)
                cout << Symbol->Symbol;
        else
        {
                for(i=0;i<NoOfDerivSymbols;i++)
                        Derivs[i]->Print();
        }
}


/*--------------------------------------------------------
  int TreeNode::DerivationDepthUpdate()

  Updating of the derivation depth from this point
  on up to the RootNode.
  --------------------------------------------------------
*/
void TreeNode::DerivationDepthUpdate()
{
        int i;

        DerivationDepth = (NoOfDerivSymbols>0);

        for(i=0;i<NoOfDerivSymbols;i++)
                DerivationDepth =
                DerivationDepth + Derivs[i]->DerivationDepth;

        if(PredecessorNode!=NULL)
                PredecessorNode->DerivationDepthUpdate();
}


/*--------------------------------------------------------
  unsigned int TreeNode::NoOfSubTrees(SymbTabEntry* SelectedSymbol,
                                      unsigned int MaxDerivDepth)
```

```
  Returning the number of subtrees, starting
  with the SelectedSymbol, not exceeding the
  given maximum derivation depth.

  Builds up a tepmporary index to be used
  by the methode SubTree(SelectedSymbol)
  -------------------------------------------------------
*/
unsigned int TreeNode::NoOfSubTrees(SymbTabEntry* SelectedSymbol,
                                    unsigned int MaxDerivDepth)
{
        int i;

        TempIndex = 0;

        for(i=0;i<NoOfDerivSymbols;i++)
                if(Derivs[i]->NoOfDerivSymbols>0)
                        TempIndex = TempIndex +
                        Derivs[i]->NoOfSubTrees(SelectedSymbol, MaxDerivDepth);

        if(Symbol==SelectedSymbol && DerivationDepth<=MaxDerivDepth)
                TempIndex++;

        return(TempIndex);
}


/*-------------------------------------------------------
  unsigned int TreeNode::Depth()

  Returning the depth (!= derivation depth) of this
  (sub-)tree.
  Implemented for experimental use only.
  -------------------------------------------------------
*/
unsigned int TreeNode::Depth()
{
        unsigned int CurrentDepth;
        unsigned int ThisDepth;
        int i;

        ThisDepth = 0;

        for(i=0;i<NoOfDerivSymbols;i++)
        {
                CurrentDepth = Derivs[i]->Depth();

                if(CurrentDepth>=ThisDepth)
                        ThisDepth = CurrentDepth + 1;
        }

        return(ThisDepth);
}


/*-------------------------------------------------------
  int TreeNode::GenoTypeStringLentgh(int Representation)

  Coputing the length of the genotype-string-representation
  of this tree from this node on, not counting the /0
  character. Representation may be LONG or SHORT
  -------------------------------------------------------
*/
int TreeNode::GenoTypeStringLength(int Representation)
{
        int i;
        int j;
```

```
        i = 0;
        if(Symbol!=NULL)
        {
                if(Representation==LONG)
                {
                        if(NoOfDerivSymbols>0)
                        {
                                i = 1 + int(log10(NoOfDerivSymbols));
                                i = i + 2 + strlen(Symbol->Symbol);
                        }
                        else
                                i = i + 2 + strlen(Symbol->Symbol);
                }
                else
                        if(NoOfDerivSymbols>0)
                                i = 2 + int(log10(DerivationNo+1));

                for(j=0;j<NoOfDerivSymbols;j++)
                        i+= Derivs[j]->GenoTypeStringLength(Representation);
        }

        return(i);
}


/*--------------------------------------------------------
  int TreeNode::PhenoTypeStringLength()

  Computing the length of the phenotype-string-representation
  of this tree from this node on, not counting the /0
  character
  --------------------------------------------------------
*/
int TreeNode::PhenoTypeStringLength()
{
        int i;
        int j;

        i = 0;
        if(NoOfDerivSymbols==0)
                i+= strlen(Symbol->Symbol);
        else
                for(j=0;j<NoOfDerivSymbols;j++)
                        i+= Derivs[j]->PhenoTypeStringLength();

        return(i);
}


/*--------------------------------------------------------
  void TreeNode::BuildUpGenoTypeStringIn(char* ResultString,
                                          int& Position,
                                          int Representation)

  Building up the genotype-string-representation in the
  ResultString-array. Position is the position of the
  current character
  --------------------------------------------------------
*/
void TreeNode::BuildUpGenoTypeStringIn(char* ResultString,
                                       int& Position,
                                       int Representation)
{
        int i;
        int n;
```

```
        if(Symbol!=NULL)
        {
                if(Representation==LONG)
                {
                        if(NoOfDerivSymbols>0)
                        {
                                n = int(log10(NoOfDerivSymbols));
                                for(i=n;i>=0;i--)
                                {
                                        ResultString[Position] = '0' +
                                        (
                                                int(NoOfDerivSymbols/pow(10,i)) -
                                                10*int(NoOfDerivSymbols/pow(10,(i+1)))
                                        );
                                        Position++;
                                }
                                ResultString[Position] = '<';
                                Position++;
                                ResultString[Position] = '\0';
                                strcat(ResultString,Symbol->Symbol);
                                Position+= strlen(Symbol->Symbol);
                                ResultString[Position] = '>';
                                Position++;
                        }
                        else
                        {
                                ResultString[Position] = '"';
                                Position++;
                                ResultString[Position] = '\0';
                                strcat(ResultString,Symbol->Symbol);
                                Position+= strlen(Symbol->Symbol);
                                ResultString[Position] = '"';
                                Position++;
                        }
                }
                else
                {
                        if(NoOfDerivSymbols>0)
                        {
                                n = int(log10(DerivationNo+1));
                                for(i=n;i>=0;i--)
                                {
                                        ResultString[Position] = '0' +
                                        (
                                                int((DerivationNo+1)/pow(10,i)) -
                                                10*int((DerivationNo+1)/pow(10,(i+1)))
                                        );
                                        Position++;
                                }
                                ResultString[Position] = '.';
                                Position++;
                        }
                }
                for(i=0;i<NoOfDerivSymbols;i++)
                        Derivs[i]->
                        BuildUpGenoTypeStringIn(ResultString,Position,Representation);
        }
}


/*--------------------------------------------------------
  void TreeNode::BuildUpPhenoTypeStringIn(char* ResultString, int& Position)

  Building up the phenotype-string-representation in the
  ResultString-array. Position is the position of the
  current character
  --------------------------------------------------------
```

```
*/
void TreeNode::BuildUpPhenoTypeStringIn(char* ResultString, int& Position)
{
        int i;

        if(NoOfDerivSymbols==0)
        {
                ResultString[Position] = '\0';
                strcat(ResultString,Symbol->Symbol);
                Position+= strlen(Symbol->Symbol);
        }
        else
        {
                for(i=0;i<NoOfDerivSymbols;i++)
                        Derivs[i]->BuildUpPhenoTypeStringIn(ResultString,Position);
        }
}


/*------------------------------------------------------
  ostream& operator <<(ostream& s, TreeNode& TheTreeNode)

  Friend of TreeNode.
  Returning the complete TreeNode, including the
  derivation, as ostream.

  This example explains the syntax. The Tree

                       <fe>
             "("   <f1>     <fe>    ")"
                 "NOT"  "(" <f0> ")"
                             "D2"

  is within the grammar

  S := <fe>;
  <fe> := "(" <f0> ")" |
          "(" <f1> <fe> ")" |
          "(" <f2> <fe> <fe> ")" ;
  <f0> := "D1" | "D2" ;
  <f1> := "NOT" ;
  <f2> := "OR" | "AND" ;

  represented by the stream

  2.1.1.2.

  because

  "("<f1><f2>")" is the 2nd derivation of <fe>,
  "NOT" is the 1st (and the only) derivation of <f1>,
  "("<f0>")" is the 1st derivation of <fe> in the second row,
  and "D2" ia the 2nd derivation of <f0>.
  ---------------------------------------------------
*/
ostream& operator <<(ostream& s, TreeNode& TheTreeNode)
{
        int i;

        if((&TheTreeNode)!=NULL && TheTreeNode.Symbol!=NULL)
        {
                if(TheTreeNode.NoOfDerivSymbols>0)
                        s << TheTreeNode.DerivationNo+1 << '.' ;

                for(i=0;i<TheTreeNode.NoOfDerivSymbols;i++)
                        s << *(TheTreeNode.Derivs[i]);
        }
```

```
        return(s);
}
```

A.1.4 Die Datei language.cc

```
// File: language.cc

#include "ga.h"
#define PTBUFF 4
#define MAXDIGITS 40

/*========================================================
   IMPLEMENTATION OF CLASSES

   SuperUnsignedInt (for experimental use)
   SuperFloat
   Language
   ProdTabEntry
   SymbTabEntry
  =======================================================
*/



/*=======================================================
  Class SuperUnsignedInt
  =======================================================
*/


/*-------------------------------------------------------
  SuperUnsignedInt::SuperUnsignedInt()

  Constructor of SuperUnsignedInt-number, containing zero
  -------------------------------------------------------
*/
SuperUnsignedInt::SuperUnsignedInt()
{
        Length = 1;
        Digit = new unsigned char[1];
        Digit[0] = 0;
}


/*-------------------------------------------------------
  SuperUnsignedInt::~SuperUnsignedInt()

  Destructor: Deleting all digits.
  -------------------------------------------------------
*/
SuperUnsignedInt::~SuperUnsignedInt()
{
        if(Length)
                delete [] Digit;
}


/*-------------------------------------------------------
  ostream& operator <<(ostream& s, SuperUnsignedInt TheNumber)

  Friend of SuperUnsignedInt.

  Returning the SuperUnsignedInt-number TheNumber as
  ostream. Preprocessor-variable MAXDIGITS gives
  the maximum number of output digits.
  -------------------------------------------------------
*/
ostream& operator <<(ostream& s, SuperUnsignedInt& TheNumber)
```

```
{
        int i;
        int j;

        if(TheNumber.Length<=MAXDIGITS)
        {
                for(i=TheNumber.Length-1;i>=0;i--)
                {
                       j = TheNumber.Digit[i];
                        s << j ;
                }
        }
        else
        {
                for(i=TheNumber.Length-1;i>=TheNumber.Length-MAXDIGITS;i--)
                {
                        if(i==TheNumber.Length-2)
                                s << ".";

                        j = TheNumber.Digit[i];
                        s << j;
                }
                s << "e+" << TheNumber.Length-1;
        }

        return(s);
}


/*----------------------------------------------------------
  int SuperUnsignedInt::operator !=(int TheOtherNumber)

  Overloading the != operator
  ----------------------------------------------------------
*/
int SuperUnsignedInt::operator !=(int TheOtherNumber)
{
        SuperUnsignedInt Temp;
        int i;

        Temp = TheOtherNumber;

        if(Length!=Temp.Length)
                return(TRUE);

        for(i=0;i<Length;i++)
                if(Digit[i]!=Temp.Digit[i])
                        return(TRUE);

        return(FALSE);
}


/*----------------------------------------------------------
  int SuperUnsignedInt::operator =(SuperUnsignedInt)

  Assignement from SuperUnsignedInt to SuperUnsignedInt
  ----------------------------------------------------------
*/
int SuperUnsignedInt::operator =(SuperUnsignedInt TheOtherNumber)
{
        int i;
        delete [] Digit;

        Digit = new unsigned char[TheOtherNumber.Length];
        Length = TheOtherNumber.Length;
        for(i=0;i<Length;i++)
```

```
                Digit[i] = TheOtherNumber.Digit[i];

        TheOtherNumber.Length = 0;   // helps the Digit-Array surviving
                                     // the SuperUnsignedInt-Destructor.

        return(TRUE);
}


/*-----------------------------------------------------------
  int SuperUnsignedInt::operator =(int);

  Assignement from int to SuperUnsignedInt
  -----------------------------------------------------------
*/
int SuperUnsignedInt::operator =(int TheOtherNumber)
{
        int i;
        int k;

        if(TheOtherNumber<0)
                TheOtherNumber = -TheOtherNumber;

        delete [] Digit;

        if(TheOtherNumber==0)
        {
                Length = 1;
                Digit = new unsigned char[1];
                Digit[0] = 0;
        }
        else
        {
                Length = int(log10(TheOtherNumber))+1;
                Digit = new unsigned char[Length];

                for(i=0;i<Length;i++)
                        Digit[i] = int(TheOtherNumber/pow(10,i))-
                                   10*int(TheOtherNumber/pow(10,i+1));

        }
}


/*-----------------------------------------------------------
  int SuperUnsignedInt::operator =(unsigned int);

  Assignement from unsigned int to SuperUnsignedInt
  -----------------------------------------------------------
*/
int SuperUnsignedInt::operator =(unsigned int TheOtherNumber)
{
        int i;
        int k;

        delete [] Digit;

        if(TheOtherNumber==0)
        {
                Length = 1;
                Digit = new unsigned char[1];
                Digit[0] = 0;
        }
        else
        {
                Length = int(log10(TheOtherNumber))+1;
                Digit = new unsigned char[Length];
```

```
                for(i=0;i<Length;i++)
                        Digit[i] = int(TheOtherNumber/pow(10,i))-
                                   10*int(TheOtherNumber/pow(10,i+1));
        }
}


/*---------------------------------------------------------
  int SuperUnsignedInt::operator =(long int);

  Assignement from long int to SuperUnsignedInt
  ---------------------------------------------------------
*/
int SuperUnsignedInt::operator =(long int TheOtherNumber)
{
        int i;
        int k;

        delete [] Digit;

        if(TheOtherNumber==0)
        {
                Length = 1;
                Digit = new unsigned char[1];
                Digit[0] = 0;
        }
        else
        {
                Length = int(log10(TheOtherNumber))+1;
                Digit = new unsigned char[Length];

                for(i=0;i<Length;i++)
                        Digit[i] = int(TheOtherNumber/pow(10,i))-
                                   10*int(TheOtherNumber/pow(10,i+1));
        }
}


/*---------------------------------------------------------
  int SuperUnsignedInt::operator +=(SuperUnsignedInt)

  Overloading the + operator
  ---------------------------------------------------------
*/
int SuperUnsignedInt::operator +=(SuperUnsignedInt& TheOtherNumber)
{
        int i;
        unsigned char Overflow;
        unsigned char* ResultNumber;
        SuperUnsignedInt* LongerNumber;
        SuperUnsignedInt* ShorterNumber;

        if(Length>TheOtherNumber.Length)
        {
                LongerNumber = this;
                ShorterNumber = &TheOtherNumber;
        }
        else
        {
                LongerNumber = &TheOtherNumber;
                ShorterNumber = this;
        }

        ResultNumber = new unsigned char[LongerNumber->Length+1];
        Overflow = 0;
```

```
        for(i=0;i<ShorterNumber->Length;i++)
        {
                ResultNumber[i] =
                ShorterNumber->Digit[i] + LongerNumber->Digit[i] + Overflow;

                Overflow = 0;
                if(ResultNumber[i]>9)
                {
                        ResultNumber[i] = ResultNumber[i] - 10;
                        Overflow = 1;
                }
        }

        for(;i<LongerNumber->Length;i++)
        {
                ResultNumber[i] = LongerNumber->Digit[i] + Overflow;
                Overflow = 0;

                if(ResultNumber[i]>9)
                {
                        ResultNumber[i] = ResultNumber[i] - 10;
                        Overflow = 1;
                }
        }

        if(Overflow==1)
        {
                ResultNumber[i] = Overflow;
                Length = LongerNumber->Length + 1;
        }
        else
                Length = LongerNumber->Length;

        delete [] Digit;
        Digit = ResultNumber;

        return(TRUE);
}


/*--------------------------------------------------------
  int SuperUnsignedInt::operator *=(SuperUnsignedInt)

  Overloading the * operator
  --------------------------------------------------------
*/
int SuperUnsignedInt::operator *=(SuperUnsignedInt& TheOtherNumber)
{
        int i;
        int j;

        unsigned char* ResultNumber;
        int ResultLength;
        unsigned char Overflow;

        if(Length==1 && Digit[0]==0)
                return(TRUE);

        if(TheOtherNumber.Length==1 && TheOtherNumber.Digit[0]==0)
        {
                delete [] Digit;
                Digit = new unsigned char[1];
                Digit[0] = 0;
                Length = 1;
                return(TRUE);
        }
```

```
        ResultLength = Length+TheOtherNumber.Length-1;
        ResultNumber = new unsigned char[ResultLength+1];
        for(i=0;i<=ResultLength;i++)
                ResultNumber[i] = 0;

        for(j=0;j<TheOtherNumber.Length;j++)
        {
                Overflow = 0;
                for(i=0;i<Length;i++)
                {
                        ResultNumber[i+j] =
                        ResultNumber[i+j] + Digit[i]*TheOtherNumber.Digit[j] + Overflow;
                        Overflow = 0;

                        if(ResultNumber[i+j]>9)
                        {
                                Overflow = ResultNumber[i+j]/10;

                                ResultNumber[i+j] =
                                ResultNumber[i+j] - 10*Overflow;
                        }
                }
                ResultNumber[i+j] = Overflow;
        }

        delete [] Digit;
        Digit = ResultNumber;
        if(Digit[ResultLength]==0)
                Length = ResultLength;
        else
                Length = ResultLength + 1;

        return(TRUE);
}



/*=========================================================
  Class SuperFloat
  =========================================================
*/


/*---------------------------------------------------------
  SuperFloat::SuperFloat()

  Constructor: Creating a SuperFloat-number, containing zero
  ---------------------------------------------------------
*/
SuperFloat::SuperFloat()
{
        Mant = 0;
        Exp = 1;
}


/*---------------------------------------------------------
  ostream& operator <<(ostream& s, SuperFloat& TheNumber)

  Friend of SuperFloat.

  Returning the SuperFloat-number TheNumber as ostream
  ---------------------------------------------------------
*/
ostream& operator <<(ostream& s, SuperFloat& TheNumber)
{
```

```
        s << TheNumber.Mant << "e" ;

        if(TheNumber.Exp>=0)
                s << "+";

        s << TheNumber.Exp;
        return(s);
}


/*-------------------------------------------------------
  int SuperFloat::operator =(int&);

  Assignement from int to SuperFloat
  -------------------------------------------------------
*/
int SuperFloat::operator =(int TheOtherNumber)
{
        if(TheOtherNumber==0)
        {
                Mant = 0;
                Exp = 1;
        }
        else
        {
                if(TheOtherNumber<0)
                        Exp = int(log10(-TheOtherNumber));
                else
                        Exp = int(log10(TheOtherNumber));

                Mant = (TheOtherNumber+0.0)/pow(10,Exp);
        }

        return(TRUE);
}


/*-------------------------------------------------------
  int SuperFloat::operator !=(int TheOtherNumber)

  Overloading the != operator
  -------------------------------------------------------
*/
int SuperFloat::operator !=(int TheOtherNumber)
{
        SuperFloat Temp;

        Temp = TheOtherNumber;

        if(Exp!=Temp.Exp)
                return(TRUE);

        if(Mant!=Temp.Mant)
                return(TRUE);

        return(FALSE);
}


/*-------------------------------------------------------
  SuperFloat SuperFloat::operator--()

  Overloading the --() operator
  -------------------------------------------------------
*/
SuperFloat SuperFloat::operator--()
{
```

```
        SuperFloat Temp;
        SuperFloat MinusOne;

        Temp = *this;

        MinusOne = -1;
        *this = *this + MinusOne;

        return(Temp);
}


/*-------------------------------------------------------
  int SuperFloat::operator >(SuperFloat TheOtherNumber)

  Overloading the > operator
  -------------------------------------------------------
*/
int SuperFloat::operator >(SuperFloat TheOtherNumber)
{
        if(Mant>=0 && TheOtherNumber.Mant<0)
                return(TRUE);

        if(Mant<0 && TheOtherNumber.Mant>=0)
                return(FALSE);

        if(Mant==0)
        {
                if(TheOtherNumber.Mant<0)
                        return(TRUE);
                else
                        return(FALSE);
        }

        if(TheOtherNumber.Mant==0)
        {
                if(Mant>0)
                        return(TRUE);
                else
                        return(FALSE);
        }

        if(Mant>0)
        {
                if(Exp>TheOtherNumber.Exp)
                        return(TRUE);

                if(Exp<TheOtherNumber.Exp)
                        return(FALSE);

                if(Mant>TheOtherNumber.Mant)
                        return(TRUE);

                return(FALSE);
        }
        else
        {
                if(Exp<TheOtherNumber.Exp)
                        return(TRUE);

                if(Exp>TheOtherNumber.Exp)
                        return(FALSE);

                if(Mant>TheOtherNumber.Mant)
                        return(TRUE);

                return(FALSE);
```

```
        }
}


/*--------------------------------------------------------
  int SuperFloat::operator >(int TheOtherNumber)

  Overloading the > operator
  --------------------------------------------------------
*/
int SuperFloat::operator >(int TheOtherNumber)
{
        SuperFloat Temp;
        int Result;

        Temp = TheOtherNumber;
        Result = (*this>Temp);
        return(Result);
}


/*--------------------------------------------------------
  int SuperFloat::operator =(float TheOtherNumber);

  Assignement from float to SuperFloat
  --------------------------------------------------------
*/
int SuperFloat::operator =(float TheOtherNumber)
{
        if(TheOtherNumber==0)
        {
                Mant = 0;
                Exp = 1;
        }
        else
        {
                if(TheOtherNumber<0)
                        Exp = int(log10(-TheOtherNumber));
                else
                        Exp = int(log10(TheOtherNumber));

                Mant = (TheOtherNumber+0.0)/pow(10,Exp);
                if((Mant>0&&Mant<1)||(Mant<0&&Mant>(-1)))
                {
                        Mant = Mant*10;
                        Exp--;
                }
        }
        return(TRUE);
}


/*--------------------------------------------------------
  int SuperFloat::operator =(double TheOtherNumber);

  Assignement from double to SuperFloat
  --------------------------------------------------------
*/
int SuperFloat::operator =(double TheOtherNumber)
{
        if(TheOtherNumber==0)
        {
                Mant = 0;
                Exp = 1;
        }
        else
        {
```

```
                if(TheOtherNumber<0)
                        Exp = int(log10(-TheOtherNumber));
                else
                        Exp = int(log10(TheOtherNumber));

                Mant = (TheOtherNumber+0.0)/pow(10,Exp);
                if((Mant>0&&Mant<1)||(Mant<0&&Mant>(-1)))
                {
                        Mant = Mant*10;
                        Exp--;
                }
        }

        return(TRUE);
}


/*-------------------------------------------------------
  SuperFloat SuperFloat::operator +(SuperFloat TheOtherNumber);

  Overloading the + operator
  -------------------------------------------------------
*/
SuperFloat SuperFloat::operator +(SuperFloat TheOtherNumber)
{
        SuperFloat ResultNumber;
        SuperFloat* TheLargerOne;
        SuperFloat* TheSmallerOne;

        if(Exp-TheOtherNumber.Exp > ACC)
                return(*this);

        if(Exp-TheOtherNumber.Exp < -ACC)
                return(TheOtherNumber);

        if(Exp>TheOtherNumber.Exp)
        {
                TheLargerOne = this;
                TheSmallerOne = &TheOtherNumber;
        }
        else
        {
                TheLargerOne = &TheOtherNumber;
                TheSmallerOne = this;
        }

        ResultNumber.Mant =
        TheLargerOne->Mant +
        TheSmallerOne->Mant/pow(10,TheLargerOne->Exp-TheSmallerOne->Exp);

        ResultNumber.Exp = TheLargerOne->Exp;

        if(ResultNumber.Mant>=10)
        {
                ResultNumber.Mant = ResultNumber.Mant / 10;
                ResultNumber.Exp++;
        }

        if(ResultNumber.Mant==0)
                ResultNumber.Exp = 1;

        return(ResultNumber);
}


/*-------------------------------------------------------
  SuperFloat SuperFloat::operator *(SuperFloat TheOtherNumber);
```

```
  Overloading the * operator
  --------------------------------------------------------
*/
SuperFloat SuperFloat::operator *(SuperFloat TheOtherNumber)
{
        SuperFloat ResultNumber;

        ResultNumber.Mant = Mant * TheOtherNumber.Mant;

        if(ResultNumber.Mant==0)
                ResultNumber.Exp = 1;
        else
                ResultNumber.Exp = Exp + TheOtherNumber.Exp;

        if(ResultNumber.Mant>=10||ResultNumber.Mant<=-10)
        {
                ResultNumber.Mant = ResultNumber.Mant / 10;
                ResultNumber.Exp++;
        }
        else
        {
                if((ResultNumber.Mant>0&&ResultNumber.Mant<1)||
                   (ResultNumber.Mant<0&&ResultNumber.Mant>-1))

                {
                        ResultNumber.Mant = ResultNumber.Mant*10;
                        ResultNumber.Exp--;
                }
        }

        return(ResultNumber);
}



/*=========================================================
  Class Language
  =========================================================
*/

/*--------------------------------------------------------
   Language::Language()

   Constructor: Creating an empty grammar with no symbols.
  --------------------------------------------------------
*/
Language::Language()
{
        FirstSymbol = NULL;
        LastSymbol = NULL;
        StartSymbol = NULL;
        ArbCard = NULL;
        ExactCard = NULL;
        ArbCardSize = -1;
        ExactCardSize = -1;
}


/*--------------------------------------------------------
  Language::~Language()

  Destructor: Deleting the whole grammar by using Clear()
  --------------------------------------------------------
*/
Language::~Language()
```

```
{
        Clear();
}


/*-------------------------------------------------------
  void Language::Clear()

  Deleting the whole Grammar
  -------------------------------------------------------
*/
void Language::Clear()
{
        if(FirstSymbol!=NULL)
        {
                SymbTabEntry* CurrentSymbol = FirstSymbol;
                SymbTabEntry* NextSymbol;

                while(CurrentSymbol!=LastSymbol)
                {
                        NextSymbol = CurrentSymbol->NextEntry;
                        delete CurrentSymbol;
                        CurrentSymbol = NextSymbol;
                }
                delete CurrentSymbol;
        }
}



/*-------------------------------------------------------
  ostream& operator << (ostream& s, Language& TheGrammar)

  Friend of Language

  Returning the Language in its BNF-representation
  as ostream.
  -------------------------------------------------------
*/
ostream& operator <<(ostream& s, Language& TheGrammar)
{
        int i;
        int j;
        int k;
        SymbTabEntry* CurrentSymbol;
        DerivSymbol* CurrentDerivSymbol;

        if(TheGrammar.StartSymbol!=NULL)
        {
                s << 'S' << ' ' << ':' << '=' << ' ' << '<';
                for(i=0;i<strlen(TheGrammar.StartSymbol->Symbol);i++)
                        s << TheGrammar.StartSymbol->Symbol[i];
                s << '>' << ' ' << ';' << endl;

                CurrentSymbol = TheGrammar.FirstSymbol;
                while(TRUE)
                {
                        if(CurrentSymbol->NoOfDerivs)
                        {
                                s << '<';
                                for(i=0;i<strlen(CurrentSymbol->Symbol);i++)
                                        s << CurrentSymbol->Symbol[i];
                                s << '>' << ' ' << ':' << '=' << ' ';

                                for(j=0;j<CurrentSymbol->NoOfDerivs;j++)
                                {
                                        CurrentDerivSymbol =
```

```
                                        CurrentSymbol->DerivList[j].FirstSymbol;

                                        for(k=0;
                                        k<CurrentSymbol->DerivList[j].NoOfDerivSymbols;
                                        k++)
                                        {
                                                if(CurrentDerivSymbol->Symbol->NoOfDerivs==0)
                                                        s << '"';
                                                else
                                                        s << '<';

                                                for(i=0;
                                                i<strlen(CurrentDerivSymbol->Symbol->Symbol);
                                                i++)
                                                        s << CurrentDerivSymbol->
                                                        Symbol->Symbol[i];

                                                if(CurrentDerivSymbol->Symbol->NoOfDerivs==0)
                                                        s << '"';
                                                else
                                                        s << '>';

                                                CurrentDerivSymbol =
                                                CurrentDerivSymbol->NextSymbol;
                                        }

                                        if(j+1==CurrentSymbol->NoOfDerivs)
                                                s << ' ' << ';' << endl;
                                        else
                                                s << ' ' << '|' << ' ';
                                }
                        }

                        if(CurrentSymbol==TheGrammar.LastSymbol)
                                break;

                        CurrentSymbol = CurrentSymbol->NextEntry;
                }
        }

        return(s);
}


/*--------------------------------------------------------
  SymbTabEntry* Language::Symbol(char* NewSymbol)

  Adding the NewSymbol, given in its text-representation,
  to the symbol-table and returning the token of symbol
  (i.e.: Pointer to Symbol-Table-Entry)

  If the NewSymbol already exists, and therefore is
  no new symbol at all, nothing is added, just the
  token (Pointer to Symbol-Table-Entry) is returned.
  -------------------------------------------------------
*/
SymbTabEntry* Language::Symbol(char* NewSymbol)
{
        int SymbolAlreadyExists = NO;
        SymbTabEntry* CurrentSymbol = NULL;

        if(FirstSymbol!=NULL)
        {
                CurrentSymbol = FirstSymbol;

                while(strcmp(CurrentSymbol->Symbol,NewSymbol)!=EQUALSTRINGS &&
                CurrentSymbol!=LastSymbol)
```

```
			CurrentSymbol = CurrentSymbol->NextEntry;

		SymbolAlreadyExists=
		(strcmp(CurrentSymbol->Symbol,NewSymbol)==EQUALSTRINGS);
	}

	if(!SymbolAlreadyExists)
	{
		SymbTabEntry* BeforeLast = LastSymbol;
		LastSymbol = new SymbTabEntry(NewSymbol, this);

		if(FirstSymbol==NULL)
			FirstSymbol = LastSymbol;
		else
			BeforeLast->NextEntry = LastSymbol;

		CurrentSymbol = LastSymbol;
	}

	return(CurrentSymbol);
}


/*----------------------------------------------------------
  void Language::Print()

  Printing out the language definition -
  first the non-terminal symbols and its derivations,
  followed by the terminal symbols.
  ---------------------------------------------------------
*/
void Language::Print()
{
	int i;
	int j;
	DerivSymbol* CurrentDerivSymbol;

	cout << endl << "LANGUAGE DEFINITION:" << endl << endl;
	if(FirstSymbol==NULL)
		cout << "Error: empty Language!" << endl;
	else
	{
		for(j=0;j<=1;j++)
		{
			SymbTabEntry* CurrentSymbol = FirstSymbol;
			while(TRUE)
			{
				if(
				(j==0&&CurrentSymbol->NoOfDerivs>0) ||
				(j==1&&CurrentSymbol->NoOfDerivs==0))
				{
					cout << CurrentSymbol->Symbol << endl;
					for(i=0;i<CurrentSymbol->NoOfDerivs;i++)
					{
						cout << "  Derivation " << i+1 << ": ";

						if(
						(CurrentSymbol->DerivList+i)->FirstSymbol==NULL)
							cout << "empty derivation." << endl;
						else
						{
							CurrentDerivSymbol =
							(CurrentSymbol->DerivList+i)->FirstSymbol;

							while(TRUE)
							{
								cout << CurrentDerivSymbol->
```

```
                                                                Symbol->Symbol << " ";

                                                                if(
                                                                CurrentDerivSymbol->NextSymbol==NULL)
                                                                        break;
                                                                else
                                                                        CurrentDerivSymbol =
                                                                        CurrentDerivSymbol->
                                                                        NextSymbol;
                                                        }
                                                        cout << endl;
                                                }
                                        }
                                }

                                if(CurrentSymbol->NextEntry==NULL)
                                        break;
                                else
                                        CurrentSymbol = CurrentSymbol->NextEntry;
                        }
                }

                if(StartSymbol==NULL)
                        cout << endl << "No start-symbol defined!" << endl;
                else
                {
                        cout << endl << "Starting symbol is <";
                        cout << StartSymbol->Symbol << ">." << endl << endl;
                }
        }
}


/*-------------------------------------------------------
  int Language::Load(char* filename)

  Loading the language definition, as defined in the
  language definition file in the BNF-representation.
  Returning FALSE in case of an error.
  -------------------------------------------------------
*/
int Language::Load(char* filename)
{
        ProdTabEntry* Derivation = NULL;
        SymbTabEntry* CurrentSymbol = NULL;
        ifstream LangDefFile;
        char CurrentChar;
        char* Buffer;
        int i;
        int warning;
        int line;
        int start;

        Clear();
        Buffer = new char[CHRBUFSIZE];
        line = 1;
        start = NO;

        LangDefFile.open(filename);
        if(!LangDefFile)
                return(FALSE);

        while(!LangDefFile.eof())
        {
                CurrentChar = LangDefFile.get();
                switch(CurrentChar)
                {
```

```
case '\n':
        line++;
        break;

case 'S':
        start = YES;
        break;

case ';':
        start = NO;
        Derivation = NULL;
        break;

case'"':
        CurrentChar = LangDefFile.get();
        i = 0;
        warning = NO;

        while(
        CurrentChar!='"' &&
        CurrentChar!=';' &&
        CurrentChar!='>' &&
        !LangDefFile.eof())
        {
                if(i<CHRBUFSIZE-1)
                {
                        Buffer[i] = CurrentChar;
                        i++;
                }
                else
                        warning = YES;

                CurrentChar = LangDefFile.get();

                if(CurrentChar=='\n')
                        line++;
        }

        if(warning)
        {
                cerr << "Line " << line << ": ";
                cerr << "Warning! Symbol length ";
                cerr << "exceeds " << CHRBUFSIZE-1;
                cerr << " characters!" << endl;
        }

        Buffer[i] = '\0';

        if(Derivation==NULL)
        {
                CurrentSymbol = Symbol(Buffer);
                cerr << "Line " << line << ": ";
                cerr << "Warning! Non-terminal symbol " ;
                cerr << Buffer << " occurs as terminal ";
                cerr << "symbol!" << endl;
                cerr << "        <" << Buffer << "> ";
                cerr << "expected instead of ";
                cerr << '"' << Buffer << '"' << "." << endl;
        }
        else
                Derivation->AddDerivSymbol(Symbol(Buffer));

        if(CurrentChar!='"')
        {
                start = NO;
                Derivation = NULL;
```

```
                        cerr << "Line " << line << ": ";
                        cerr << "Warning! " << '"' ;
                        cerr << " expected!" << endl;
                }
                break;

        case '<':
                CurrentChar = LangDefFile.get();
                i = 0;
                warning = NO;

                while(
                CurrentChar!= '>' &&
                CurrentChar!= ';' &&
                CurrentChar!= '"' &&
                !LangDefFile.eof())
                {
                        if(i<CHRBUFSIZE-1)
                        {
                                Buffer[i] = CurrentChar;
                                i++;
                        }
                        else
                                warning=YES;

                        CurrentChar = LangDefFile.get();

                        if(CurrentChar=='\n')
                                line++;
                }

                if(warning)
                {
                        cerr << "Line " << line << ": ";
                        cerr << "Warning! Symbol length ";
                        cerr << "exceeds " << CHRBUFSIZE-1;
                        cerr << " characters!" << endl;
                }

                Buffer[i] = '\0';

                if(Derivation==NULL)
                {
                        if(start)
                        {
                                StartSymbol = Symbol(Buffer);
                                start=NO;
                        }
                        else
                                CurrentSymbol = Symbol(Buffer);
                }
                else
                        Derivation->AddDerivSymbol(Symbol(Buffer));

                if(CurrentChar!='>')
                {
                        if(CurrentChar!='"')
                        {
                                start = NO;
                                Derivation = NULL;
                        }
                        cerr << "Line " << line << " ";
                        cerr << "Warning! '>' expected!";
                        cerr << endl;
                }
                break;
```

```
                case ':':
                        CurrentChar = LangDefFile.get();
                        if(CurrentChar=='\n')
                                line++;
                        if(CurrentChar==';')
                        {
                                Derivation = NULL;
                                start = NO;
                        }
                        if( CurrentChar!='=' || (CurrentChar=='=' && start) )
                                break;
                case '|':
                        if(CurrentSymbol==NULL)
                        {
                                cerr << "Line " << line << ": ";
                                cerr << "Error! Derivation can not ";
                                cerr << "be related with any symbol!";
                                cerr << endl;
                        }
                        else
                                Derivation =
                                CurrentSymbol->AddProdTabEntry();

                        break;
        }
    }

    if(StartSymbol==NULL && FirstSymbol!=NULL)
    {
        StartSymbol = FirstSymbol;
        while(TRUE)
        {
                if(StartSymbol->NextEntry==NULL||
                   StartSymbol->NoOfDerivs!=0)
                        break;
                else
                        StartSymbol = StartSymbol->NextEntry;
        }

        cerr << "Warning! No start-symbol defined." << endl;
        cerr << "I declare the first non-terminal symbol <";
        cerr << StartSymbol->Symbol;
        cerr << "> as starting symbol." << endl;
    }

    delete [] Buffer;

    return(FirstSymbol!=NULL);
}


/*-------------------------------------------------------
  void Language::ComputeArbSearchSpace(int Depth);

  Computing the search space Size up to Depth
  derivations and storing the result in ArbCard.

  First deleting cached old values in the symbol-
  and production table.
  -------------------------------------------------------
*/
int Language::ComputeArbSearchSpace(int Depth)
{
    int i;
    SymbTabEntry* CurrentSymbol;

    if(StartSymbol==NULL || Depth<0)
```

```
                return(FALSE);

        CurrentSymbol = FirstSymbol;
        while(TRUE)
        {
                if(CurrentSymbol->ArbCardSize>=0)
                {
                        CurrentSymbol->ArbCardSize = -1;
                        delete [] CurrentSymbol->ArbCard;
                        CurrentSymbol->ArbCard = NULL;
                }

                for(i=0;i<CurrentSymbol->NoOfDerivs;i++)
                {
                        if(CurrentSymbol->DerivList[i].ArbCardSize>=0)
                        {
                                CurrentSymbol->DerivList[i].ArbCardSize = -1;
                                delete [] CurrentSymbol->DerivList[i].ArbCard;
                                CurrentSymbol->DerivList[i].ArbCard = NULL;
                        }
                }

                if(CurrentSymbol==LastSymbol)
                        break;

                CurrentSymbol = CurrentSymbol->NextEntry;
        }

        ArbCard = StartSymbol->ComputeArbSearchSpace(Depth);
        ArbCardSize = Depth;
        return(TRUE);
}


/*-------------------------------------------------------
  void Language::ComputeExactSearchSpace(int Depth);

  Computing the exact Search Space Size up to Depth
  Derivations and storing the result in ExactCard.

  First deleting cached old values in the symbol-
  and production table.
  -------------------------------------------------------
*/
int Language::ComputeExactSearchSpace(int Depth)
{
        int i;
        SymbTabEntry* CurrentSymbol;

        if(StartSymbol==NULL || Depth<0)
                return(FALSE);

        CurrentSymbol = FirstSymbol;
        while(TRUE)
        {
                if(CurrentSymbol->ExactCardSize>=0)
                {
                        CurrentSymbol->ExactCardSize = -1;
                        delete [] CurrentSymbol->ExactCard;
                        CurrentSymbol->ExactCard = NULL;
                }

                for(i=0;i<CurrentSymbol->NoOfDerivs;i++)
                {
                        if(CurrentSymbol->DerivList[i].ExactCardSize>=0)
                        {
                                CurrentSymbol->DerivList[i].ExactCardSize = -1;
```

```
                                delete [] CurrentSymbol->DerivList[i].ExactCard;
                                CurrentSymbol->DerivList[i].ExactCard = NULL;
                        }
                }

                if(CurrentSymbol==LastSymbol)
                        break;

                CurrentSymbol = CurrentSymbol->NextEntry;
        }

        ExactCard = StartSymbol->ComputeExactSearchSpace(Depth);
        ExactCardSize = Depth;
        return(TRUE);
}


/*------------------------------------------------------
  void Language::PrintExactSearchSpace()

  Printing out the card-table of the exact search space
  depths calculateted by

  Language::ComputeExactSearchSpace(int)
  ------------------------------------------------------
*/
void Language::PrintExactSearchSpace()
{
        int i;

        cout << "Table of exact search space sizes in n derivations:" << endl << endl;

        for(i=0;i<=ExactCardSize;i++)
        {
                if(ExactCard[i]!=0)
                        cout << i << ": " << ExactCard[i] << endl;
        }
}


/*------------------------------------------------------
  void Language::PrintArbSearchSpace()

  Printing out the card-table of the search space
  depths calculateted by

  Language::ComputeExactSearchSpace(int)
  ------------------------------------------------------
*/
void Language::PrintArbSearchSpace()
{
        int i;

        cout << "Table of search space sizes in n derivations," << endl;
        cout << " calculated with long double - accuracy." << endl << endl;

        for(i=0;i<=ArbCardSize;i++)
        {
                if(ArbCard[i]!=0)
                        cout << i << ": " << ArbCard[i] << endl;
        }
}



/*======================================================
```

```
  Now the symbol-table entry
  =======================================================
*/


/*-------------------------------------------------------
  SymbTabEntry::SymbTabEntry()

  Constructor: Creating an empty entry of the Symbol-Table
  -------------------------------------------------------
*/

SymbTabEntry::SymbTabEntry()
{
        Symbol="empty symbol";
        DerivList=NULL;
        L = NULL;
        NoOfDerivs=0;
        NextEntry=NULL;
        ExactCard = NULL;
        ExactCardSize = -1;
        ArbCard = NULL;
        ArbCardSize = -1;
}


/*-------------------------------------------------------
  SymbTabEntry::SymbTabEntry(char* NewSymbol)

  Constructor: Creating an entry of the Symbol-Table
  with the written representation of the Symbol,
  achieved in NewSymbol. No derivations of this symbol
  are created by this constructor, and the NextEntry-
  Pointer is set to NULL.
  -------------------------------------------------------
*/

SymbTabEntry::SymbTabEntry(char* NewSymbol, Language* SymbolLanguage)
{
        Symbol=new char[strlen(NewSymbol)+1];
        L = SymbolLanguage;
        strcpy(Symbol,NewSymbol);
        DerivList=NULL;
        NoOfDerivs=0;
        NextEntry=NULL;
        ExactCard = NULL;
        ExactCardSize = -1;
        ArbCard = NULL;
        ArbCardSize = -1;
}


/*-------------------------------------------------------
  SymbTabEntry::~SymbTabEntry()

  Destructor: Deleting the symbol-string, the
  ProductionTable-entries, and, before that,
  the derivations related with the ProductionTable-
  Entries.
  -------------------------------------------------------
*/

SymbTabEntry::~SymbTabEntry()
{
        int i;
        delete [] Symbol;
```

```
        for(i=0;i<NoOfDerivs;i++)
                (DerivList+i)->Release();

        delete [] DerivList;

        if(ExactCardSize>-1)
                delete [] ExactCard;

        if(ArbCardSize>-1)
                delete [] ArbCard;
}


/*---------------------------------------------------------
  ProdTabEntry* SymbTabEntry::AddProdTabEntry()

  Adding an entry to the production-table. All entries
  are held in a line in an array, rather than a chained list.

  If the allocated space runs out, a lager array is
  allocated, the old entries are copied to the new
  space, and the old one is deleted.

  This enables a very fast acces to all derivations
  during run-time.
  --------------------------------------------------------
*/

ProdTabEntry* SymbTabEntry::AddProdTabEntry()
{
        ProdTabEntry* NewProdTab;
        int i;

        if(NoOfDerivs%PTBUFF == 0)
        {
                NewProdTab = new ProdTabEntry[NoOfDerivs+PTBUFF]();

                for(i=0;i<NoOfDerivs;i++)
                        NewProdTab[i] = DerivList[i];

                if(NoOfDerivs!=0)
                        delete [] DerivList;

                DerivList = NewProdTab;
        }

        NoOfDerivs++;
        return(DerivList+NoOfDerivs-1);
}


/*---------------------------------------------------------
  SuperFloat* SymbTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)

  Computing the search space for MaxDerivDepth derivations
  with limited (long double float) accuracy.
  --------------------------------------------------------
*/
SuperFloat* SymbTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)
{
        SuperFloat AchtBit;
        AchtBit = 256;

        int i;
        int j;
        SuperFloat* NewCard;
        SuperFloat* TempCard;
```

```
        if(MaxDerivDepth<=ArbCardSize)
                return(ArbCard);

        NewCard = new SuperFloat[MaxDerivDepth+1];
        for(i=0;i<=MaxDerivDepth;i++)
                NewCard[i] = 0;

        if(!NoOfDerivs)
        {

// The following lines may be included if necessary or for
// experimental use to calculate search space sizes including
// eight byte floating point numbers.
//
// In the BNF-file the eight byte floating point number
// must be represented as terminal symbol
// "FLOATING_POINT_NUMBER".

/*
                if(strcmp(Symbol,"FLOATING_POINT_NUMBER")==EQUALSTRINGS)
                {
                        NewCard[0] = AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                        NewCard[0] = NewCard[0]*AchtBit;
                }
                else
*/

// End optional lines of code.

                        NewCard[0] = 1;
        }
        else
        {
                if(MaxDerivDepth)
                {
                        for(i=0;i<NoOfDerivs;i++)
                        {
                                TempCard =
                                DerivList[i].ComputeArbSearchSpace(MaxDerivDepth-1);

                                for(j=1;j<=MaxDerivDepth;j++)
                                        NewCard[j] =
                                        NewCard[j] + TempCard[j-1];
                        }
                }
        }

        delete [] ArbCard;
        ArbCard = NewCard;
        ArbCardSize = MaxDerivDepth;

        return(ArbCard);
}


/*----------------------------------------------------------
  SuperUnsignedInt* SymbTabEntry::ComputeExactSearchSpace(int MaxDerivDepth)

  Computing the search space size for MaxDerivDepth
  derivations
```

```
  ------------------------------------------------------
*/
SuperUnsignedInt* SymbTabEntry::ComputeExactSearchSpace(int MaxDerivDepth)
{
        int i;
        int j;
        SuperUnsignedInt* NewCard;
        SuperUnsignedInt* TempCard;

        SuperUnsignedInt AchtBit;
        AchtBit = 256;

        if(MaxDerivDepth<=ExactCardSize)
                return(ExactCard);

        NewCard = new SuperUnsignedInt[MaxDerivDepth+1];

        if(!NoOfDerivs)
        {

// The following lines may be included if necessary or for
// experimental use to calculate search space sizes including
// eight byte floating point numbers.
//
// In the BNF-file the eight byte floating point number
// must be represented as terminal symbol
// "FLOATING_POINT_NUMBER".

/*
                if(strcmp(Symbol,"FLOATING_POINT_NUMBER")==EQUALSTRINGS)
                {
                        NewCard[0] = AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                        NewCard[0] *= AchtBit;
                }
                else
*/

// End optional lines of code.

                        NewCard[0] = 1;
        }
        else
        {
                if(MaxDerivDepth)
                {
                        for(i=0;i<NoOfDerivs;i++)
                        {
                                TempCard =
                                DerivList[i].ComputeExactSearchSpace(MaxDerivDepth-1);

                                for(j=1;j<=MaxDerivDepth;j++)
                                        NewCard[j] += TempCard[j-1];
                        }
                }
        }

        delete [] ExactCard;
        ExactCard = NewCard;
        ExactCardSize = MaxDerivDepth;

        return(ExactCard);
```

```
}



/*=======================================================
  Now the production-table entry
  =======================================================
*/


/*-------------------------------------------------------
  ProdTabEntry::ProdTabEntry()

  Constructor: Creating an empty production-table
  entry
  -------------------------------------------------------
*/

ProdTabEntry::ProdTabEntry()
{
        FirstSymbol = NULL;
        LastSymbol = NULL;
        NoOfDerivSymbols = 0;
        ExactCard = NULL;
        ExactCardSize = -1;
        ArbCard = NULL;
        ArbCardSize = -1;
}


/*-------------------------------------------------------
  ProdTabEntry::~ProdTabEntry()

  Destructor: Deleting cached search-space-size vectors.
  -------------------------------------------------------
*/
ProdTabEntry::~ProdTabEntry()
{
        if(ExactCardSize>-1)
                delete [] ExactCard;

        if(ArbCardSize>-1)
                delete [] ArbCard;
}


/*-------------------------------------------------------
  void ProdTabEntry::Release()

  Kind of destructor: Deletes the whole derivation.
  To be called before destruction, if wished.
  Not impemented as a regular destructor, because
  of the SymbTabEntry::AddProdTabEntry()-construction.

  In this methode, an array of ProdTabEntries is copied
  to another array, before erasing the old one. The
  related derivations, however, shall not be killed,
  and have to be killed manually by using this methode,
  if wished.
  -------------------------------------------------------
*/
void ProdTabEntry::Release()
{
        DerivSymbol* CurrentSymbol;
        DerivSymbol* NextSymbol;
```

```
        if(FirstSymbol!=NULL)
        {
                CurrentSymbol = FirstSymbol;

                while(CurrentSymbol!=LastSymbol)
                {
                        NextSymbol = CurrentSymbol->NextSymbol;
                        delete CurrentSymbol;
                        CurrentSymbol = NextSymbol;
                }
                delete CurrentSymbol;
        }
}


/*--------------------------------------------------------
  DerivSymbol* ProdTabEntry::AddDerivSymbol(SymbTabEntry* NewSymbol)

  Adding a symbol given in its token-representation
  (i.e.: pointer to the symbol-table) to this derivation
  --------------------------------------------------------
*/
DerivSymbol* ProdTabEntry::AddDerivSymbol(SymbTabEntry* NewSymbol)
{
        DerivSymbol* BeforeLast;

        NoOfDerivSymbols++;
        BeforeLast = LastSymbol;
        LastSymbol = new DerivSymbol;
        LastSymbol->Symbol = NewSymbol;
        LastSymbol->NextSymbol = NULL;

        if(FirstSymbol==NULL)
                FirstSymbol = LastSymbol;
        else
                BeforeLast->NextSymbol = LastSymbol;

        if(NewSymbol->L!=NULL)
        {
                NewSymbol->L->ArbCard = NULL;
                NewSymbol->L->ArbCardSize = -1;
                NewSymbol->L->ExactCard = NULL;
                NewSymbol->L->ExactCardSize = -1;
        }

        return(LastSymbol);
}


/*--------------------------------------------------------
  SuperFloat* ProdTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)

  Computing the search space for MaxDerivDepth derivations
  with limitied (long double float) accuracy.
  --------------------------------------------------------
*/
SuperFloat* ProdTabEntry::ComputeArbSearchSpace(int MaxDerivDepth)
{
        int i;
        int j;
        int k;
        SuperFloat* NewCard;
        SuperFloat* TempCard1;
        SuperFloat* TempCard2;
        SuperFloat* XCard;
        DerivSymbol* CurrentDerivSymbol;
```

```
        if(MaxDerivDepth<=ArbCardSize)
                return(ArbCard);

        NewCard = new SuperFloat[MaxDerivDepth+1];
        TempCard2 = new SuperFloat[MaxDerivDepth+1];

        NewCard[0] = 1;

        for(i=1;i<=MaxDerivDepth;i++)
                NewCard[i] = 0;

        CurrentDerivSymbol = FirstSymbol;
        for(i=0;i<NoOfDerivSymbols;i++)
        {
                TempCard1 =
                CurrentDerivSymbol->Symbol->
                ComputeArbSearchSpace(MaxDerivDepth);

                for(j=0;j<=MaxDerivDepth;j++)
                        TempCard2[j] = 0;

                for(j=0;j<=MaxDerivDepth;j++)
                {
                        if(NewCard[j]!=0)
                        {
                                for(k=0;k<=(MaxDerivDepth-j);k++)
                                {
                                        TempCard2[j+k] =
                                        TempCard2[j+k] +
                                        NewCard[j] * TempCard1[k];
                                }
                        }
                }

                XCard = NewCard;
                NewCard = TempCard2;
                TempCard2 = XCard;

                CurrentDerivSymbol =
                CurrentDerivSymbol->NextSymbol;
        }

        delete [] TempCard2;
        delete [] ArbCard;
        ArbCard = NewCard;
        ArbCardSize = MaxDerivDepth;

        return(ArbCard);
}


/*--------------------------------------------------------
  SuperUnsignedInt* ProdTabEntry::ComputeExactSearchSpace(int MaxDerivDepth)

  Computing the search space size up to MaxDerivDepth
  derivations with unlimited precision

  (implemented for experimental use)
  --------------------------------------------------------
*/
SuperUnsignedInt* ProdTabEntry::ComputeExactSearchSpace(int MaxDerivDepth)
{
        int i;
        int j;
        int k;
        SuperUnsignedInt Temp;
        SuperUnsignedInt* NewCard;
```

```
        SuperUnsignedInt* TempCard1;
        SuperUnsignedInt* TempCard2;
        SuperUnsignedInt* XCard;
        DerivSymbol* CurrentDerivSymbol;

        if(MaxDerivDepth<=ExactCardSize)
                return(ExactCard);

        NewCard = new SuperUnsignedInt[MaxDerivDepth+1];
        TempCard2 = new SuperUnsignedInt[MaxDerivDepth+1];

        NewCard[0] = 1;

        CurrentDerivSymbol = FirstSymbol;
        for(i=0;i<NoOfDerivSymbols;i++)
        {
                TempCard1 =
                CurrentDerivSymbol->Symbol->
                ComputeExactSearchSpace(MaxDerivDepth);

                for(j=0;j<=MaxDerivDepth;j++)
                        TempCard2[j] = 0;

                for(j=0;j<=MaxDerivDepth;j++)
                {
                        if(NewCard[j]!=0)
                        {
                                for(k=0;k<=(MaxDerivDepth-j);k++)
                                {
                                        Temp = NewCard[j];
                                        Temp *= TempCard1[k];
                                        TempCard2[j+k] += Temp;
                                }
                        }
                }

                XCard = NewCard;
                NewCard = TempCard2;
                TempCard2 = XCard;

                CurrentDerivSymbol =
                CurrentDerivSymbol->NextSymbol;
        }

        for(i=0;i<=MaxDerivDepth;i++)
                TempCard2[i].Length = 0;

        delete [] TempCard2;
        delete [] ExactCard;
        ExactCard = NewCard;
        ExactCardSize = MaxDerivDepth;

        return(ExactCard);
}
```

A.1.5 Die Datei uniran.h

```
#ifndef __RAND_H__
#define __RAND_H__

/*===============================================================================
Zufallszahlengenerator (prime modulus multiplicative linear congruential generator)
Z[I]=(630360016*Z[I-1]   (MOD 2147483647)
nach dem PASCAL-Programm aus Law, A.M. and Kelton, D.W. (1991)
Simulation Modeling and Analysis. New York, McGraw-Hill,
basierend auf Roberts' FORTRAN-Zufallszahlengenerator UNIRAN.
Adaptiert auf C++ von H. Hoerner, 1995, 1996

Zur Unterstuetzung mehrerer Zufalls-Streams koennen durch Aufruf des Konstruktors
Uniran(int x) mit einem Wert zwischen (-100 < x < -1) hundert verschiedene vorbereitete
Zufalls-seeds aktiviert werden.

Aufruf mit 0 oder Aufruf des Leerkonstruktors Uniran()  bewirkt Initialisierung mit der
Systemzeit time(NULL) aus <time.h>.

Aufruf des Konstruktors Uniran(int) mit einer positiven Zahl zwischen 1 und 2147483646
inclusive (= 2 hoch 15) bewirkt Initialisierung mit eben diesem Wert.

Berechnung der naechsten Zufallszahl U(0,1) mit der Funktion float rand()

Die zuletzt berechnete, zugrundeliegende Pseudozufallszahl erhaelt man mit der Funktion
int randst()

Eine ganzzahlig Zufallszahl im Bereich (0:i-1) erhaelt man durch Aufruf
der Funktion int dice(int i)
===============================================================================
*/

extern "C"
{
#include<time.h>
}

#define B2E15 32768
#define B2E16 65536
#define MODLUS 2147483647
#define MULT1 24112
#define MULT2 26143

class Uniran
{
        int zrng;
public:
        Uniran();
        Uniran(int);
        int randst();
        float rand();
        int dice(int);
};

#endif
```

A.1.6 Die Datei uniran.cc

```
#ifndef __UNIRAN_INC_
#define __UNIRAN_INC_
#include "uniran.h"

Uniran::Uniran()
//
// Constructor: Initializing seed zrng with system-time
//
{
        zrng=time(NULL);
}

Uniran::Uniran(int i)
//         CONSTRUCTOR:
//         i>0: seed zrng is set to i
// -100<=i<=-1: seed zrng is set to one of 100 predefined values
//        else: seed zrng is set to system-time
//
{
        switch (i)
        {
        case -1:
                zrng=1973272912;
                break;
        case -2:
                zrng= 281629770;
                break;
        case -3:
                zrng=  20006270;
                break;
        case -4:
                zrng=1280689831;
                break;
        case -5:
                zrng=2096730329;
                break;
        case -6:
                zrng=1933576050;
                break;
        case -7:
                zrng= 913566091;
                break;
        case -8:
                zrng= 246780520;
                break;
        case -9:
                zrng=1363774876;
                break;
        case -10:
                zrng= 604901985;
                break;
        case -11:
                zrng=1511192140;
                break;
        case -12:
                zrng=1259851944;
                break;
        case -13:
                zrng= 824064364;
                break;
        case -14:
                zrng= 150493284;
                break;
```

```
    case -15:
            zrng= 242708531;
            break;
    case -16:
            zrng=  75253171;
            break;
    case -17:
            zrng=1964472944;
            break;
    case -18:
            zrng=1202299975;
            break;
    case -19:
            zrng= 233217322;
            break;
    case -20:
            zrng=1911216000;
            break;
    case -21:
            zrng= 726370533;
            break;
    case -22:
            zrng= 403498145;
            break;
    case -23:
            zrng= 993232223;
            break;
    case -24:
            zrng=1103205531;
            break;
    case -25:
            zrng= 762430696;
            break;
    case -26:
            zrng=1922803170;
            break;
    case -27:
            zrng=1385516923;
            break;
    case -28:
            zrng=  76271663;
            break;
    case -29:
            zrng= 413682397;
            break;
    case -30:
            zrng= 726466604;
            break;
    case -31:
            zrng= 336157058;
            break;
    case -32:
            zrng=1432650381;
            break;
    case -33:
            zrng=1120463904;
            break;
    case -34:
            zrng= 595778810;
            break;
    case -35:
            zrng= 877722890;
            break;
    case -36:
            zrng=1046574445;
            break;
    case -37:
```

```
            zrng=  68911991;
            break;
    case -38:
            zrng=2088367019;
            break;
    case -39:
            zrng= 748545416;
            break;
    case -40:
            zrng= 622401386;
            break;
    case -41:
            zrng=2122378830;
            break;
    case -42:
            zrng= 640690903;
            break;
    case -43:
              zrng=1774806513;
              break;
    case -44:
            zrng=2132545692;
            break;
    case -45:
            zrng=2079249579;
            break;
    case -46:
            zrng=  78130110;
            break;
    case -47:
            zrng= 852776735;
            break;
    case -48:
            zrng=1187867272;
            break;
    case -49:
            zrng=1351423507;
            break;
    case -50:
            zrng=1645973084;
            break;
    case -51:
            zrng=1997049139;
            break;
    case -52:
            zrng= 922510944;
            break;
    case -53:
            zrng=2045512870;
            break;
    case -54:
            zrng= 898585771;
            break;
    case -55:
            zrng= 243649545;
            break;
    case -56:
            zrng=1004818771;
            break;
    case -57:
            zrng= 773686062;
            break;
    case -58:
            zrng= 403188473;
            break;
    case -59:
            zrng= 372279877;
```

```
            break;
    case -60:
            zrng=1901633463;
            break;
    case -61:
              zrng= 498067494;
              break;
    case -62:
            zrng=2087759558;
            break;
    case -63:
            zrng= 493157915;
            break;
    case -64:
            zrng= 597104727;
            break;
    case -65:
            zrng=1530940798;
            break;
    case -66:
            zrng=1814496276;
            break;
    case -67:
            zrng= 536444882;
            break;
    case -68:
            zrng=1663153658;
            break;
    case -69:
            zrng= 855503735;
            break;
    case -70:
            zrng=  67784357;
            break;
    case -71:
            zrng=1432404475;
            break;
    case -72:
            zrng= 619691088;
            break;
    case -73:
            zrng= 119025595;
            break;
    case -74:
            zrng= 880802310;
            break;
    case -75:
            zrng= 176192644;
            break;
    case -76:
            zrng=1116780070;
            break;
    case -77:
            zrng= 277854671;
            break;
    case -78:
            zrng=1366580350;
            break;
    case -79:
            zrng=1142483975;
            break;
    case -80:
            zrng=2026948561;
            break;
    case -81:
            zrng=1053920743;
            break;
```

```
        case -82:
                zrng= 786262391;
                break;
        case -83:
                zrng=1792203830;
                break;
        case -84:
                zrng=1494667770;
                break;
        case -85:
                zrng=1923011392;
                break;
        case -86:
                zrng=1433700034;
                break;
        case -87:
                zrng=1244184613;
                break;
        case -88:
                  zrng=1147297105;
                  break;
        case -89:
                zrng= 539712780;
                break;
        case -90:
                zrng=1545929719;
                break;
        case -91:
                  zrng= 190641742;
                  break;
        case -92:
                zrng=1645390429;
                break;
        case -93:
                zrng= 264907697;
                break;
        case -94:
                zrng= 620389253;
                break;
        case -95:
                zrng=1502074852;
                break;
        case -96:
                zrng= 927711160;
                break;
        case -97:
                zrng= 364849192;
                break;
        case -98:
                zrng=2049576050;
                break;
        case -99:
                zrng= 638580085;
                break;
        case -100:
                zrng= 547070247;
                break;
        default:
                zrng=(i>0 ? i : time(NULL));
                break;
        }
}

Uniran::randst()
//
// provides last produced internal random number (between 1 and 2147483646)
//
```

```
{
        return(zrng);
}

float Uniran::rand()
//
// provides next floating-point random-number
//
{
        int hi15, hi31, low15, lowprd, ovflow, zi, rand;

        zi=zrng;
        hi15=zi/B2E16;
        lowprd=(zi - hi15 * B2E16) * MULT1;
        low15=lowprd/B2E16;
        hi31=hi15 * MULT1 + low15;
        ovflow=hi31/  B2E15;
        zi=(((lowprd - low15 * B2E16) - MODLUS) + (hi31 - ovflow * B2E15) * B2E16) + ovflow;
        if(zi < 0)
                zi=zi + MODLUS;
        hi15=zi/B2E16;
        lowprd=(zi - hi15 * B2E16) * MULT2;
        low15=lowprd/B2E16;
        hi31=hi15 * MULT2 + low15;
        ovflow=hi31/B2E15;
        zi=(((lowprd - low15 * B2E16) - MODLUS) + (hi31 - ovflow * B2E15) * B2E16) + ovflow;
    if(zi < 0)
                zi=zi + MODLUS;
        zrng= zi;
        rand=2*(zi/256)+1;
        return(rand / 16777216.0);
}

int Uniran::dice(int i)
//
// provides next int random-number 0 <= random-number < i
//
{
        int r;
        r = int( (i+0.0)*rand() );
        return( (r==i ? i - 1 : r ) );
}
#endif
```