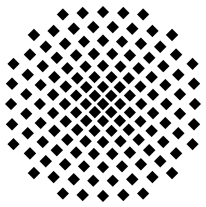
Abschlußbericht der
Projektgruppe
Evolutionäre Algorithmen
Bericht Nr. 1997/02



Universität Stuttgart

Abschlußbericht der Projektgruppe Evolutionäre Algorithmen

Matthias Großmann
Alexander Leonhardi
Thomas Schmidt

Betreuung
Prof. Dr. Volker Claus
Dipl.-Inf. Wolfgang Reissenberger
Dipl.-Math. Nicole Weicker

Abteilung Formale Konzepte
Fakultät Informatik
Universität Stuttgart

20. Februar 1997

Prof. Dr. Volker Claus
Abteilung Formale Konzepte
Institut für Informatik
Universität Stuttgart

Breitwiesenstr. 20-22
D-70565 Stuttgart

Telefon:

0711-7816-300 (Prof. Dr. V. Claus)
0711-7816-301 (Sekretariat)
0711-7816-330 (FAX)

E-Mail: claus@informatik.uni-stuttgart.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Als im Mai 1995 die Projektgruppe Genetische Algorithmen (PGA) – die erste Projektgruppe der Informatik an der Universität Stuttgart – endete, war das Ergebnis hinter den ursprünglichen Planungen zurückgeblieben. Vorgesehen war, eine Experimentierplattform für Evolutionäre Algorithmen zu entwerfen und zu implementieren. Die Implementierung des von der Projektgruppe „EAGLE“ genannten Systems scheiterte jedoch, im Endbericht [AJK+95] wurde lediglich der Entwurf sowie eine funktionale Spezifikation eines Teilsystems veröffentlicht. Aufbauend auf dem Endbericht entstand nach Abschluß der Projektgruppe eine funktionale Spezifikation des Gesamtsystems, die gegenüber dem ursprünglichen Entwurf von EAGLE bereits einige Änderungen aufwies. Die Spezifikation kann in [JW95] nachgelesen werden.

Im Oktober 1995 fand das erste Treffen der Projektgruppe Evolutionäre Algorithmen statt. Ziel unserer Projektgruppe war zunächst, ausgehend vom Endbericht der PGA und der funktionalen Spezifikation, tatsächlich auch ein lauffähiges Programm zu implementieren. Während unserer Arbeit sind aber so viele eigene Ideen eingeflossen, daß von der funktionalen Spezifikation von EAGLE nur grundlegende Ideen übernommen wurden. Im Gegensatz zur PGA, die zu Beginn aus acht, später sieben Mitgliedern bestand, umfaßte die Projektgruppe EVA zunächst nur fünf Studenten, von denen einer bereits nach wenigen Wochen absprang. Die vorgesehene Untergrenze von sechs Teilnehmern war damit zwar deutlich unterschritten, die Projektgruppe wurde aber dennoch fortgesetzt, da sowohl von den Mitgliedern als auch von unseren Betreuern bereits viel Arbeit in die Projektgruppe investiert worden war. Trotz (wegen?) der geringen Teilnehmerzahl entstand im Laufe eines Jahres der Entwurf eines Systems, das z.T. komplexer als EAGLE ist, ein Prototyp sowie ein lauffähiges Programm.

Dieser Bericht stellt zum einen die Dokumentation dieses Programms dar und beschreibt zum anderen das Vorgehen der Projektgruppe Evolutionäre Algorithmen über verschiedene Zwischenergebnisse bis zum endgültigen Produkt. Er gliedert sich im Wesentlichen in

- eine Beschreibung einer Projektgruppe allgemein sowie der Aufgabe der Projektgruppe EVA,

- unser Vorgehen und unsere Anforderunge an GENOM,
- Überblick über den Aufbau von GENOM,
- Dokumentation von Kodierung, Populationsverwaltung, LEA und des Interpreters,
- Erweiterungsmöglichkeiten und Bedienung und
- Anhänge mit Beschreibungen der Bibliotheken.

Die Mitglieder der Projektgruppe EVA waren:

- Matthias Großmann: verantwortlich für Kodierung, Einleitung, Vorgehen
- Darko Ivančan: verantwortlich für Populationsverwaltung
- Alexander Leonhardi: verantwortlich für Interpreter, Anforderungen, Bedienung, Erweiterungen, Anhang
- Thomas Schmidt: verantwortlich für Interpreter, Systembeschreibung

Unser Dank gilt Professor Claus, der diese Projektgruppe initiiert und trotz widriger Anfangsumstände ihre Fortsetzung ermöglicht hat, sowie besonders unseren Betreuern Wolfgang Reissenberger und Nicole Weicker für ihr großes Engagement.

Wir haben unser Programm GENOM (GENOM is an Environment for Optimization Methods) genannt. Es soll helfen, Probleme bei der Analyse Evolutionärer Algorithmen zu lösen: Da auf diesem Gebiet bisher nur sehr wenig Sätze bewiesen sind, ist man auf experimentelle Untersuchungen und damit auf entsprechende, möglichst flexibel einsetzbare Werkzeuge angewiesen.

Teil I

Entwicklungsprozeß

Kapitel 2

Projektgruppe Evolutionäre Algorithmen

2.1 Was ist eine Projektgruppe

Das Studium der Informatik vermittelt dem Studenten zwar einen großen Teil des nötigen Fachwissens, jedoch stellt das Berufsleben noch weitere Anforderungen an den Informatiker. Teamfähigkeit und Erfahrung spielen gerade bei der Mitarbeit an großen Software-Projekten eine wichtige Rolle. Hier verfolgt die Idee der Projektgruppe folgende Ausbildungsziele:

- Arbeiten im Team
- Analyse von Problemen, Strukturierung von Lösungen und gemeinsamer Entwurf geeigneter Systeme
- Selbstständige Erarbeitung von Lösungsvorschlägen und deren Vorstellung und Verteidigung in einer Gruppe
- Übernahme von Verantwortung für die Lösung von Teilaufgaben und die Erstellung von Modulen
- Mitwirkung an einer umfassenden Dokumentation
- Erstellen eines Software-Produktes, das ein Einzelner innerhalb des vorgegebenen Zeitraumes unmöglich bewältigen kann
- Projekt-Planung und Kosten/Nutzen-Analyse
- Einsatz von Werkzeugen
- Persönlichkeitsbildung (Übernahme von Verantwortung, Selbstvertrauen, Verläßlichkeit, Rücksichtnahme, Durchsetzungsfähigkeit usw.)

An der Projektgruppe nehmen in der Regel acht bis zwölf Studierende des Hauptstudiums teil. Sie erarbeiten im Laufe eines Jahres ein Software-Produkt, welches einem Zeitaufwand von mehreren Personenjahren entspricht. Hierbei sollen sämtliche Phasen eines Software-Lifecycles — von der Planung bis zur Wartung — durchlaufen werden, was in anderen Lehrveranstaltungen nicht üblich ist. Bei Software- und Fachpraktika wird zumeist eine gegebene, genau festgelegte Aufgabenstellung in ein Programm umgesetzt.

Eine Projektgruppe vereinigt die Lehrveranstaltungsformen „Hauptseminar“ (2 SWS), „Fachpraktikum“ (4 SWS) und „Studienarbeit“ (10 SWS) in sich. Demzufolge ist eine Projektgruppe mit 16 SWS einzustufen.

Der Ablauf einer Projektgruppe folgt meist folgendem Schema: Seminar-, Planungs-, Entwurfs-, Implementierungs-, Integrations-, Experimentier- und Schlußphase. Diese Phasen werden im folgenden genauer erläutert.

Seminarphase: Die Themenstellung wird gründlich analysiert. Dazu werden von den Mitgliedern Originalpublikationen durchgearbeitet und die Ergebnisse vorgetragen. Ergebnisse dieser Phase sind viel Wissen, je eine Vortragsausarbeitung und eine zusammenfassende Darstellung der Literaturauswertung.

Planungsphase: Die Projektgruppe analysiert den Problembereich, stellt Einsatzmöglichkeiten und Anwendungen zusammen, erarbeitet einen Anforderungskatalog und diskutiert Lösungsmöglichkeiten für diese Fragestellungen. Hierbei werden die in der Literatur bekannten Lösungsvorschläge und eigene Ideen gegeneinander abgewogen. Insbesondere wird frühzeitig diskutiert, welche Hard- und Software für die jeweiligen Lösungen erforderlich ist, welche sonstigen Kosten entstehen, wie hoch der Zeitaufwand sein wird, usw. Wichtig ist eine frühe Spezifizierung der Eigenschaften des Systems (Robustheit, Antwortverhalten, Flexibilität, Schutzmechanismen, Erweiterbarkeit, Verteiltheit, ...). Inhaltliches Ergebnis ist eine möglichst eindeutige, ausschnittsweise sogar formale Spezifikation. Für jede ins Auge gefaßte Anwendung wird darüber hinaus ein Szenario bzgl. des Einsatzes, der Nutzung, der Tests und der Wartung skizziert. Organisatorische Ergebnisse sind ein grober Zeitplan und die erste Aufteilung von Aufgabengebieten. Hier setzt auch eine Spezialisierung der Gruppenmitglieder ein.

Entwurfsphase: Voraussetzung für die Entwurfsphase ist, daß Begriffsbestimmungen, Anwendungen und Modelle weitgehend geklärt sind. Nach Festlegung des grundsätzlichen Lösungsverfahrens werden Teilprobleme und charakteristische Objekte herauskristallisiert, miteinander in Beziehung gesetzt, auf ihre Realisierbarkeit geprüft und grundlegende Datenstrukturen und Kommunikationswege festgelegt. Dabei werden die Schnittstellen der Einzelteile des Systems untereinander genau definiert. Ergebnis ist ein Plan des zu erstellenden (oder zu modifizierenden) Systems. Stehen die einzelnen Aufgaben fest, werden sie auf die Mitglieder verteilt. Die Implementierungssprache(n) sowie die erforderliche Hardware und die zu verwendenden Werkzeuge werden festgelegt. Eine Liste von Beispielen, die das System später positiv bewältigen muß, wird für die Testphase erstellt.

In der *Implementationsphase* und *Integrationsphase* wird der Programmcode erstellt, zusammengebunden (integriert) und getestet.

Die *Experimentierphase* schließt weitere Tests mit speziellen Anwendungen ein.

Zur *Schlußphase* zählt in erster Linie der Abschluß der *Dokumentation*, die ständig parallel zur Projektgruppenarbeit erstellt und auf den neuesten Stand gebracht wird.

Das Konzept der Projektgruppe wird bereits seit Jahren an anderen Universitäten wie z.B. in Oldenburg und Dortmund erprobt und durchgeführt. Dort sind Projektgruppen z.T. schon Pflichtveranstaltungen im Rahmen des Informatikstudiums.

2.2 Problemstellung

2.2.1 Optimierungsprobleme

Viele Probleme, die sich in der Wissenschaft, Technik oder Wirtschaft stellen, lassen sich als Optimierungsproblem beschreiben, als Aufgabenstellung also, bei der zu einer gegebenen Umwelt (Suchraum) ein optimaler Punkt innerhalb dieses Umwelt gesucht ist. Die Güte eines solchen Punktes in gegebenen Suchraum wird in der Regel durch eine Qualitäts- oder Kostenfunktion berechnet.

Vielen dieser Optimierungsprobleme gemeinsam ist ihre Schwierigkeit, die in der Informatik durch den Begriff der NP-Härte ausgedrückt wird. (NP-Härte heißt, daß sich die Lösung eines solchen Problems nur um einen polynomiellen Faktor von der eines beliebigen Problems unterscheidet, das von einer nichtdeterministischen Turingmaschine in polynomieller Laufzeit gelöst werden kann; vgl. [CS88]) Dies bedeutet, daß diese Probleme nicht effektiv lösbar sind. Deshalb werden bei einer solchen Problemstellung mittels Heuristiken oder Evolutionärer Algorithmen Näherungslösungen gesucht.

Beispiele von schwierigen Optimierungsproblemen

1. Minimierung von mathematischen Testfunktionen [DJ75]
2. Minimierung der Kosten in einer Fabrik, die durch Leerlauf oder Umrüstung von Maschinen entstehen (bekannt als Produktionsplanungsproblem) [BBSS88, Bru93, Joh73, WSF89, Zäp82]
3. Findung einer kürzesten Rundreise zu verschiedenen Orten (bekannt als das Traveling Salesman Problem, kurz TSP) [Beu81, Bra90, GL85, GH91, LLRKS85, WSF89]
4. Konstruktion eines möglichst leichten jedoch stabilen Tragwerkes für den Bau von leichteren Flugzeugen [Ben92, Hö79, Kir90, Kir94, Mau94, Mle92]

2.2.2 Evolutionäre Algorithmen

Die speziellen Verfahren der Evolutionären Algorithmen [AJK+95] sind das Thema dieser Projektgruppe. Unter einem Evolutionären Algorithmus wird dabei

ein zufallsgesteuertes Optimierungsverfahren verstanden, daß an Methoden der Natur angelehnt ist.

Beispiele Evolutionärer Algorithmen

- Simulated Annealing (SA) [GWH90]
- Threshold Accepting (TA) [DS90]
- Genetische Algorithmen (GA) [Hol75, Gol89, Dav91]
- Evolutionsstrategien (ES) [Rec73, Sch81]

Den Verfahren gemeinsam ist, daß sie iterativ auf einer oder mehreren möglichen Lösungen (Individuen genannt) arbeiten und von dort aus durch verschiedene Operatoren (Mutation bzw. Rekombination) bessere Lösungen zu erreichen suchen. Eine Selektion leitet einen neuen Iterationsschritt ein.

2.2.3 Aufgabenstellung der Projektgruppe

Ausgehend aus der Problembeschreibung ergibt sich die folgende Aufgabe für die Projektgruppe „Evolutionäre Algorithmen“ : Es soll, aufbauend auf die Ergebnisse der Projektgruppe „Genetische Algorithmen“ [AJJ+94, AJK+95] und des Technischen Berichts [JW95] ein System zur Bearbeitung hartnäckiger (NP-harter) Probleme mit Hilfe von Evolutionären Algorithmen wie z.B. Evolutionsstrategien und Genetischen Algorithmen erstellt werden.

Das System soll dabei folgende Konzepte enthalten:

- Unterscheidung zwischen Problem- und Kodierungsstruktur. Dadurch wird eine einheitliche Darstellung des Problems erreicht und damit von der Sichtweise der Algorithmen getrennt. Den Übergang zwischen Problem- und Kodierungsstruktur bilden die Kodierungs- bzw die Dekodierungsfunktionen.
- Verwendung verschiedener Datentypen innerhalb einer Problem- bzw. Kodierungsstruktur.
- Möglichst freie Kombinierbarkeit von Verfahren und Operatoren, um unter Rückgriff auf vorhandene Operatoren neue Verfahren ausprobieren zu können.
- Austauschbarkeit von Individuen verschiedener Kodierungen zwischen Verfahren, um hybride Verfahren möglich zu machen.
- Abgestufte Einstiegsmöglichkeiten für den Benutzer.
- Nebenläufige Algorithmen sollen implementierbar sein.

2.3 Vorgehen

Bereits zu Beginn der Projektgruppe war eine Einteilung der zwei Semester in verschiedene Phasen vorgesehen. Im Einzelnen waren dies eine Seminar-, eine Spezifikations-, eine Enwurfs- und eine Implementierungsphase. Auch wenn wir später zum Teil insbesondere zeitlich von der vorgegebenen Gliederung abgewichen sind, ist grundsätzlich die Einteilung in einzelne Phasen immer erhalten geblieben.

2.3.1 Seminarphase

Während der Seminarphase arbeiteten sich die Mitglieder der Projektgruppe in verschiedene Teilbereiche der Themengebiete genetische und evolutionäre Algorithmen und Grundlagen funktionaler Programmiersprachen[1] ein. Jedes Mitglied hielt im Rahmen dieser Phase einen Vortrag über das jeweils vertiefte Gebiet. Folgende Themen wurden behandelt:

- Algebraische Spezifikation und Typ-Polymorphismus: Dieser Vortrag sollte für die Beteiligten der Projektgruppe eine erste Einführung in die Konzepte von SML darstellen, die für die meisten völlig neu waren. Ein großer Unterschied von SML zu anderen funktionalen Sprachen (wie LISP) besteht in einer strengen Typprüfung, die zur Übersetzungszeit durchgeführt wird, jedoch polymorphe Deklarationen zuläßt. Die Grundlagen dieser Typprüfung sollen in der Ausarbeitung vorgestellt werden.
- Sammlung von Problemen und Optimierungsverfahren: In diesem Abschnitt wurde ein Überblick über typische Optimierungsprobleme gegeben. Einige bekannte Optimierungsverfahren wurden vorgestellt.
- Genetisches Programmieren: Dabei handelt es sich um eine Anwendung Genetischer Algorithmen zur Erzeugung von Programmen. Sie läßt sich mit EAGLE nicht realisieren, da die Individuen Baumstrukturen variabler Größe sind.
- Parallele Modelle Evolutionärer Algorithmen: Hier wurden einige Verfahren vorgestellt, die parallel mit mehreren Populationen arbeiten. Solche Verfahren lassen sich in EAGLE nicht verwenden, so daß hier wie schon beim letzten Vortrag der Wunsch nach einer entsprechenden Erweiterung entstand.

Ausarbeitungen der ersten drei Vorträge wurden in [GILS96] veröffentlicht.

2.3.2 Entwurfsphase

Von EAGLE, dem System der ersten Projektgruppe, stand uns eine Spezifikation zur Verfügung. Hätten wir uns stärker an EAGLE orientiert, wäre eine kurze

[1] Zu diesem Zeitpunkt stand bereits fest, daß wir GENOM in der funktionalen Sprache SML implementieren.

Entwurfsphase ausreichend gewesen. Mit dem Konzept von EAGLE lassen sich aber einige der in den Seminarvorträgen vorgestellten Ideen und Probleme, wie Verfahren mit mehreren Populationen oder genetisches Programmieren, nicht realisieren. GENOM sollte gegenüber EAGLE mit mehreren Populationen arbeiten können und wesentlich mehr Freiheiten bei der Wahl der Datentypen für Individuen zulassen, z.B. auch Listen variabler Länge oder Bäume.

Eine wichtige Rolle spielte in EAGLE der Interpreter für die eigens entwickelte Programmiersprache LEA (Language for Evolutionary Algorithms), in der die Evolutionären Algorithmen für das System geschrieben werden sollten. Der Interpreter übernahm damit während des Experiments die Steuerung des Gesamtsystems.

Um das System trotz der vorgesehenen Erweiterungen mit nur vier Personen fertigstellen zu können, war zunächst geplant, den Interpreter nicht zu implementieren. Alle Verfahren und Operatoren hätten dann in SML geschrieben werden müssen. Das System bestand damit aus zwei zentralen Komponenten, der Populationsverwaltung, die Funktionen zum Zugriff auf Individuen und Populationen zur Verfügung stellen sollte, und der Experimentsteuerung, die anstatt des Interpreters die Funktion einer Schnittstelle zwischen dem Verfahren und dem System übernehmen sollte.

Wir haben dieses Konzept gegen Ende der Entwurfsphase im Februar weitgehend geändert, da uns die Möglichkeiten, die die Experimentsteuerung bot, als zu wenig komfortabel erschienen. Die Experimentsteuerung wurde durch einen Interpreter ersetzt, dem die Populationsverwaltung sowie die Kodierung untergeordnet sind.

2.3.3 Prototyp

Bereits in den ersten Wochen der Projektgruppe hatten wir entschieden, für die Implementierung die funktionale Programmiersprache SML zu verwenden, zum einen, weil für EAGLE eine funktionale Spezifikation vorhanden war, zum anderen erschien uns SML wegen seiner polymorphen Datentypen für unser Projekt besonders geeignet. Da bis zu diesem Zeitpunkt keiner von uns ein größeres Programm in SML geschrieben hatte, entschieden wir uns, einen Prototyp zu erstellen. So konnten wir uns uns sowohl mit der Programmiersprache vertraut machen, als auch Probleme, die sich bei der Implementierung des Endsystems stellen würden, frühzeitig erkennen.

Wir beschränkten uns beim Prototyp auf eine Umsetzung der einfacheren Konzepte von EAGLE und verzichteten auf die Implementierung der neuen Fähigkeiten von GENOM sowie den Interpreter. Der Prototyp sollte nur mit einer Population arbeiten. Die Individuen bestanden aus einer festen Anzahl von Atomen. Als Atome waren reelle und ganze Zahlen vorgesehen, als Kodierungen Identität und Binärkodierung. Der Prototyp wurde von einer Untergruppe in einer Woche spezifiziert und in drei Wochen implementiert.

Da sich direkt an die Fertigstellung des Prototyps die Arbeit am Zwischenbericht anschloß, haben wir mit dem Prototyp sehr wenig Experimente durchgeführt. Ein Threshold- und ein genetischer Algorithmus fanden das Optimum einer einfachen Fitneßfunktion (vierdimensionale Hypersphäre).

2.3.4 Spezifikations- und Entwurfsphase

Die Spezifikationsphase begann nach Abschluß des Zwischenberichts im April. Einige Teile des Systems, insbesondere der Interpreter und die Kodierung, waren in der Entwurfsphase nur sehr grob durchdacht worden, so daß sich auch angesichts der noch verbleibenden Zeit bis zum Ende der Projektgruppe die Spezifikation ziemlich hektisch gestaltete. Da einige Mitglieder der Projektgruppe bereits genauere Vorstellungen vom Interpreter hatten, wurde beschlossen, die Teile des Interpreters, die vom Restsystem unabhängig sind, bereits zu implementieren, während der Rest der Gruppe die übrigen Teile spezifizierte. Die Spezifikationsphase ging so weitgehend nahtlos in die Implementierung über, auch während der nächsten Phase mußten immer wieder Systemteile (neu) spezifiziert werden.

2.3.5 Implementierungs- und Testphase

Mit der Implementierung von GENOM begannen wir im Mai und schlossen sie Ende September weitgehend ab. Zu Beginn der Implementierungsphase waren Teile der Projektgruppe noch mit Spezifikationen befaßt, gegen Ende führten wir bereits Tests der schon fertiggestellten Teile durch. Die Arbeiten an diesem Abschlußbericht begannen ebenfalls vor Ende der Implementierungsphase. Der Interpreter wurde im Juni fertiggestellt, die übrigen zentralen Teile wie Kodierung und Populationsverwaltung im Juli. Der Rest dieser Phase entfiel auf Korrektur von Fehlern sowie Erstellung von Bibliotheken und Experimenten zu Testzwecken. Zur Durchführung systematischer Versuche fehlte nach Abschluß der Implementierungsphase die Zeit, da wir uns in erster Linie auf den Abschlußbericht konzentrierten.

Kapitel 3

Anforderungen an das System

Nach den Erfahrungen mit dem Prototyp wurden die Anforderungen festgelegt, die dem zu entwickelnden System zugrunde liegen sollen. Dabei wurde noch nicht entschieden, welche der geforderten Eigenschaften im Rahmen der Projektgruppe realisiert werden. Allerdings sollen auch die Anforderungen, die nicht direkt in das System einfließen, bei der Entwicklung mit beachtet werden und durch spätere Erweiterungen möglich gemacht werden.

3.1 Allgemeine Anforderungen

Beschrieben werden zuerst die allgemeinen Anforderungen an das System. Aus diesen werden im weiteren die Anforderungen an die einzelnen Teile abgeleitet, aus denen das System besteht.

Neben der Möglichkeit, neue Evolutionäre Algorithmen zu entwickeln, soll das System möglichst viele der existierenden Verfahren aus dem Bereich der Evolutionären Algorithmen und verwandten Optimierungsverfahren unterstützen. Im einzelnen sind die folgenden Verfahren bei der Entwicklung des Systems zu beachten:

- Hill-Climbing Verfahren,
- Threshold Algorithmus,
- Great Deluge Algorithmus,
- Simulated Annealing,
- Genetische Algorithmen,
- Evolutionsstrategien,
- Genetisches Programmieren

- und eventuell parallele Varianten dieser Verfahren.

Ein Teil des Systems stellen auch die Probleme dar, auf die die Verfahren angewendet werden. Es wird eine Unterstützung für möglichst viele der Probleme gefordert, die normalerweise mit Evolutionären Algorithmen bearbeitet werden. Dies sind vor allem

- die übliche Testfunktionen (z.B. Hypersphäre, De Jong'sche Testfunktionen etc.), mit denen die Eigenschaften der entwickelten Algorithmen überprüft werden können,
- andere einfach berechenbare Funktionen,
- Programme für das genetische Programmieren,
- Travelling-Salesman-Probleme und
- Netzstrukturen oder Matrizen für die Optimierung neuronaler Netze.

Für die Anwendung der Verfahren auf die Probleme und die Verwendung des Systems allgemein gibt es die folgenden Anforderungen:

- Die verschiedenen Möglichkeiten des Einstiegs in das System werden durch ein Schichtenmodell beschrieben. Es existieren mehrere Schichten, die jeweils auf die darunterliegenden aufbauen. Je tiefer eine der Schichten gelegen ist, desto größer ist der Umfang, in dem dort Veränderungen vorgenommen werden können. Im gleichen Maße vertiefen sich aber auch die Kenntnisse, die für das Verständnis dieser Schichten nötig sind.
- In der obersten Schicht sollen Verfahren unkompliziert auf einfachere Probleme (einfach berechenbare Funktionen) angewendet werden können, vor allem ohne tiefere Kenntnisse vom internen Aufbau unseres Programms und, wenn möglich, ohne Kenntnisse von SML. Dazu soll eine übersichtliche und einfache Möglichkeit zur Durchführung eines Experiments ohne umständliche Vorbereitungen gegeben sein, wenn dieses nur existierende Komponenten verwendet. Wünschenswert ist hierbei eine graphische Oberfläche.
- Die Parameter dieser Verfahren (z.B. die Temperatur bei SA) sind einfach und, wenn möglich, auch während des Programmlaufs zu setzen und zu verändern.
- Vom System produzierte Daten erlauben eine umfassende Analyse, die vom System unterstützt wird.

3.2 Struktur des Systems

Aus den oben genannten Anforderungen und den Erfahrungen aus der Erstellung des Prototyps wurde die, in der folgenden Abbildung gezeigte, Struktur

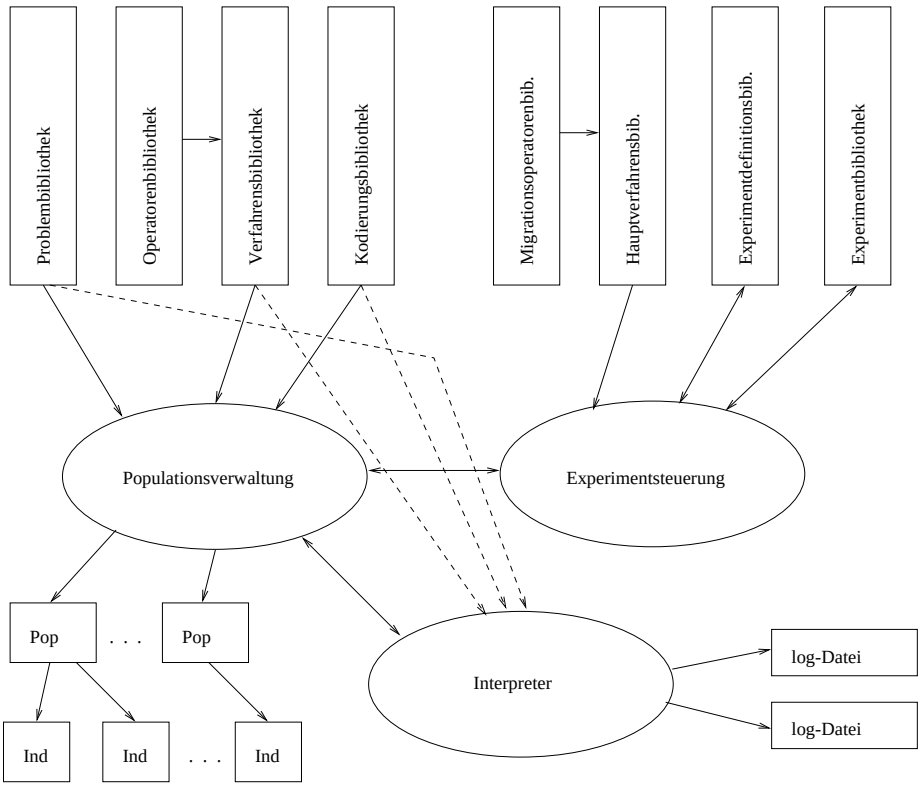


Abb. 3.1: Entwurf des Systems GENOM

des Systems entwickelt. Bei dieser Abbildung wird zwischen Systemelementen zur Kapselung von Daten und solchen zur Steuerung von Berechnungen unterschieden. Im folgenden werden die einzelnen Teile beschrieben, wie sie als Entwurf geplant wurden. Diese Struktur und die darin enthaltenen Programmteile dienten als Grundlage für die weitere Ausarbeitung der Anforderungen. An beiden wurden im weiteren Vorgehen noch wesentliche Änderungen vorgenommen.

3.2.1 Elemente zur Kapselung der Daten

3.2.1.1 Experimentdefinition

Eine Experimentdefinition beinhaltet die folgenden Informationen:

- ein Problem (Problemstruktur oder -raum, Fitneßfunktion)
- eine bestimmte Anzahl von Populationen, denen jeweils ein Verfahren und eine Kodierung zugeordnet ist
- ein Hauptverfahren (Meta-Verfahrens, das die Migration von Individuen zwischen verschiedenen Populationen steuert)

- für jede Population einen Random-Seed

3.2.1.2 Experiment

Ein Experiment beinhaltet zu einer Experimentdefinition die folgenden Daten:

- Laufinitialisierung (Belegung der Parameter, Setzen der Anfangspopulationen)
- log-Dateien für jeder Verfahren und das Hauptverfahren

3.2.1.3 Problem

Ein Problem beinhaltet

- eine Problembeschreibung (nur für den Menschen)
- eine Problemstruktur
- eine Fitneßfunktion

3.2.1.4 Kodierung

Eine Kodierung beinhaltet

- Beschreibung der Kodierung (nur für den Menschen)
- statische Informationen zur Überprüfung von der Kompatibilität bzgl. verschiedener Verfahren
- Kodierungsstruktur (Grobstruktur, Belegung der Atome mit konkreten Typen)
- Funktionen zur Kodierung und Dekodierung
- die für die konkrete Kodierungsstruktur benötigten Funktionen

3.2.1.5 Verfahren

Ein Verfahren besteht aus

- einer großen Iterationsschleife, in der verschiedene Operatoren verwendet werden
- statischen Informationen zur Überprüfung der Kompabilität bzgl. verschiedener Kodierungen
- einer Liste von Parametern mit Defaultbelegungen (bestehend aus den eigenen Parametern und denen der verwendeten Operatoren)

3.2.1.6 Operator

Ein Operator besteht aus

- einem Algorithmusteil (eventuell können hier andere Operatoren verwendet werden)
- einer Parameterliste mit Defaultbelegungen (auch der eventuellen Suboperatoren.

3.2.1.7 Hauptverfahren

Spezielles Verfahren zur Steuerung mehrerer Verfahren auf mehreren Populationen. Verwendet eventuell spezielle Migrationsoperatoren.

Ruft die verschiedenen Verfahren mit einer Generationshaltebedingung auf. Führt nach Stop aller Verfahren Migration durch.

3.2.1.8 Migrationsoperatoren

Spezielle Operatoren zum Austausch von Individuen zwischen verschiedenen Populationen. z.B. Einfügen des besten Individuums einer Population in alle anderen Populationen oder Ersetzen des schlechtesten Individuums einer Population durch das beste einer anderen.

3.2.1.9 Population

Eine Population besteht aus

- Bezeichner für eine Kodierung
- Bezeichner für ein Verfahren
- Random-Seed
- Liste von Individuen

3.2.1.10 Individuum

Ein Individuum ist eine konkrete Ausprägung des Vereinigungsdatentyps aller möglichen Individuen.

3.2.2 Elemente zur Steuerung von Berechnungen

3.2.2.1 Experimentsteuerung

Die Experimentsteuerung ist die Stelle, an der das Laufzeitverhalten festgelegt wird (Initialisierung, Aufruf von Verfahren (über Populationsverwaltung), Migration). Dazu wird entweder eine Experimentdefinition geladen oder es werden die folgenden Dinge der Populationsverwaltung übergeben:

- Bezeichner eines Problems
- Kombination Bezeichner von Kodierung und Bezeichner Verfahren und Anzahl der Populationen, die mit dieser Kombination initialisiert werden sollen

Die Experimentsteuerung besitzt Funktionen, die es ermöglichen, Parameterlisten zu lesen und zu ändern.

Ist Schnittstelle zum Benutzer (über Dateien, Kommandozeile, UI oder GUI).

3.2.2.2 Populationsverwaltung

Die Populationsverwaltung

- führt bei der Initialisierung eine statische Überprüfung der Kompabilität von Kodierungen und Verfahren durch
- veranlaßt, daß die entsprechenden lauffähigen Teile in den Interpreter geladen werden
- gibt die Parameterliste (inklusive Defaultbelegungen) der jeweiligen Verfahren an die Experimentsteuerung
- ruft für jede Population den Interpreter auf (übergibt dabei die Population dem Interpreter)
- schreibt nach jedem Iterationsschritt des Interpreters die aktuelle Population zurück
- kennt die initialisierten Populationen, ihre jeweiligen Kodierungen und Verfahren, weiß wieviele Individuen in den jeweiligen Dateien sind
- meldet an die Experimentsteuerung, wenn die jeweiligen Haltebedingungen erfüllt sind
- führt die Migration aus (eventuell macht das der Interpreter)

3.2.2.3 Interpreter

Der Interpreter führt die eigentlichen Berechnungen durch. Für jede Population wird der Interpreter mit den jeweiligen Initialisierungen gestartet. Nach einem Iterationsdurchlauf eines Verfahrens gibt der Interpreter die aktuelle Population an die Popverwaltung zurück. Während der Berechnung schreibt der Interpreter in eine ihm zugewiesene log-Datei die Daten, die in dem Verfahren als Ausgabe vorgesehen sind.

Nach jedem Iterationsdurchlauf kann der Interpreter abfragen, ob das System halten soll. Es sind Laufinitialisierungsveränderungen durch den Benutzer möglich. Danach wird die Berechnung durch den Interpreter fortgesetzt.

3.3 Anforderungen an die Programmteile

Die Anforderungen an die Programmteile, die in der oben beschriebenen Struktur des Systems enthalten sind, sollen im weiteren aus den Anforderungen für das gesamte System und den geforderten Verfahren und Problemen abgeleitet werden.

3.3.1 Problemstruktur und Fitneßfunktion

Ein Problem wird im System durch die Problemstruktur und eine Fitneßfunktion dargestellt. Um die oben genannten Probleme mit dem System bearbeiten zu können, wird für die Problemstruktur die Unterstützung folgender Elemente gefordert:

- Vektoren aus reellen Zahlen (für mathematische Funktionen und die meisten Testfunktionen), sowie Ganzzahlvektoren und Bitvektoren (für Genetische Algorithmen),
- Permutationen für das Travelling Salesman Problem,
- Vektoren mit gemischten Typen (reelle, binäre und ganze Zahlen) z.B. für die Topologieoptimierung ebener Fachwerke,
- genetischen Programmen (entweder als Baum oder in einer Klammerdarstellung),
- Matrizen, z.B. für Neuronale Netze.

Zusätzlich ist eine Unterstützung für das Anbinden von externen Problemen sinnvoll, da sich die Fitneß eines Individuums bei praktischen Problemen oft nicht einfach berechnen läßt. Für ihre Ermittlung werden in diesem Fall externe Programme mit komplexeren Berechnungsverfahren (wie z.B. einer Finite Elemente Methode) oder Simulatoren benötigt.

Für die Fitneßfunktion ergeben sich folgende Forderungen:

- Einfache Umsetzung von beliebigen mathematischen Funktionen und
- die Unterstützung einer externen Berechnung.

3.3.2 Kodierungsstruktur

Einige der Verfahren benötigen eine bestimmte Darstellung der von ihnen zu bearbeitenden Strukturen. Wenn verschiedene Verfahren auf ein Problem angewendet werden sollen, ist es sinnvoll, das Problem in einer verfahrensunabhängigen Darstellung anzugeben, die je nach Verfahren angepaßt werden kann. Bei der Kombination von Verfahren und Problemen ist es daher oft notwendig, das Problem so zu kodieren, daß es eine Struktur erhält, die von einem bestimmten Verfahren bearbeitet werden kann. Eine solche Struktur wird Kodierungsstruktur genannt. So kann z.B. ein Problem mit reellwertigen Parametern in

einen Bitvektor kodiert werden, um darauf einen Genetischen Algorithmus anzuwenden. Es soll natürlich auch möglich sein, ein Verfahren direkt auf eine Problemstruktur anzuwenden. Für die oben geforderten Verfahren muß die Kodierungsstruktur die folgenden Elemente unterstützen:

- Ganzzahlvektoren,
- Vektoren von reellen Zahlen für Great Deluge, Simulated Annealing, Evolutionsstrategien und Threshold Algorithmen,
- Bitvektoren für Genetische Algorithmen,
- Strategieparameter, wie sie von Evolutionsstrategien verwendet werden,
- Vektoren mit gemischten Typen und
- Genetische Programme entweder als Syntaxbaum oder in Klammerdarstellung.

3.3.3 Kodierung und Dekodierung

Die Kodierung bzw. Dekodierung wandelt eine Problemstruktur in eine Kodierungsstruktur um bzw. umgekehrt. Aus den aufgelisteten Forderungen an Problemstrukturen und den Verfahren ergeben sich für Kodierung und Dekodierung die folgenden Eigenschaften:

- Es soll vom System aus möglich sein, ganze und reelle Zahlen als Bitstring entweder einfach binär oder mit einer Gray-Kodierung zu kodieren.
- Permutationen sollen als Realzahlvektoren kodiert werden können.
- Außerdem müssen Elemente der Problemstruktur unkodiert in die Kodierungsstruktur übernommen werden können.

3.3.4 Operatoren

Operatoren sind die Bausteine aus denen die Verfahren bestehen. Um möglichst einfach neue Verfahren erstellen zu können, müssen diese flexibel erstellt und eingesetzt werden können. Forderungen an die Operatoren sind:

- Die Erstellung von beliebigen eigenen Operatoren, die die kodierten Individuen verändern können, soll möglich sein. Für Verfahren, die mehrere Individuen verwenden (z.B. Genetische Algorithmen und Evolutionsstrategien), muß es möglich sein, Individuen zu erstellen und zu löschen. Um auch parallele Varianten der Algorithmen zu unterstützen, ist eine Migration von Individuen zwischen verschiedenen Populationen nötig (siehe unten).
- Um einen flexiblen Einsatz der Operatoren zu ermöglichen, sollen Parameter für diese Operatoren definiert werden können, mit denen wichtige Eigenschaften der Operatoren gesteuert werden.

- Für die Kompontenten der oben genannten Verfahren (z.B. Mutation und Selektion) sollen Operatoren in einer Bibliothek vorgegeben sein, die in eigene Verfahren eingebaut werden können.
- Für auftretende Fehler soll angegeben werden können, wie diese behandelt werden sollen. Wenn keine Behandlung angegeben ist, wird vom System die Ausgabe einer sinnvollen Fehlermeldung erwartet.

3.3.5 Experimentsteuerung

Der Ablauf eines Experiments wird in einer Experimentdefinition beschrieben. Sie soll die unten angegebenen Möglichkeiten bieten:

- Die Möglichkeit, die Auswahl von Problem, Fitneßfunktion, Kodierung und Verfahren zusammenzufassen.
- Die Parameter der Operatoren sollen voreingestellte Werte haben und vor Beginn der entsprechenden Verfahren neu gesetzt werden können.
- Eventuell die Möglichkeit, Haltepunkte in Operatoren setzen und Parameter während des Ablaufs verändern zu können.
- Um Experimente auch selbständig ablaufen lassen zu können, ist eine vollständige Steuerung über Dateien sinnvoll.
- In diesem Fall müssen auch alle Initialisierungen über Dateien möglich sein.

3.3.6 Individuen und Population

Individuen sind die Ausprägung einer Kodierungsstruktur. Für die Verwendung im System werden sie zu Populationen zusammengefaßt, die an die Verfahren übergeben werden. Diese bearbeiten die Populationen, um für ein Problem ein optimales Individuum zu finden. Für die Verwaltung der Individuen ergeben sich aus den einzelnen Verfahren diese Anforderungen:

- Unterstützung von Populationen, die mehrere Individuen enthalten z.B. für Genetische Algorithmen und Evolutionsstrategien,
- Unterstützung von verschiedenen Populationen mit gleicher Kodierung z.B. für parallele Varianten dieser Verfahren.
- Ebenso die Unterstützung von mehreren Populationen mit verschiedenen Kodierungen
- und die Möglichkeit der Migration von Individuen zwischen zwei Populationen gleicher oder verschiedener Kodierung.
- Um auf ein früheres oder abgebrochenes Experiment wieder aufsetzen zu können, sollen Populationen aus Dateien initialisiert werden können.

Teil II

GENOM

Kapitel 4

Überblick über GENOM

GENOM bietet eine Plattform um Parameter-Optimierungsprobleme mit Evolutionären Verfahren zu bearbeiten, sowie die Verfahren zu untersuchen. Das System erlaubt einen einfachen Zugang zu Evolutionären Verfahren, indem es dem Benutzer Werkzeuge, Funktionen und Bibliotheken zur Verfügung stellt. Diese Bibliotheken enthalten einige der am häufigsten verwendeten Algorithmen und Probleme, die vom Benutzer einfach an die konkrete Aufgabenstellung angepaßt werden können.

4.1 Besonderheiten

Es gibt bereits eine Anzahl von Systemen, die zur Durchführung Evolutionärer Verfahren dienen. Das vorliegende System unterscheidet sich jedoch in einigen Punkten von den bisher bekannten (genauere Erläuterungen finden sich im Abschnitt Modulgruppen):

- Darstellung der Individuen
  Das System erlaubt sehr unterschiedliche Formen von Individuen. Durch die Definition von Atomen und aus diesen aufgebauten Strukturen ist eine nahezu unbegrenzte Vielfalt möglich. So kann z.B. eine Liste (Grobstruktur) mit ganzen Zahlen (Atome) gefüllt werden, aber auch ein Baum aus Zeichenketten bestehen. Individuen können aus verschiedenen Atom- und Strukturtypen bestehen, Strukturen können von variabler Länge sein.
- Unterscheidung zwischen Phäno- und Genotyp
  Der Phänotyp ist die Struktur der Individuen aus Problemsicht, der Genotyp die Struktur aus Sicht der Verfahren. Die Überführung von Phäno- in den Genotyp erfolgt durch Kodierung; der umgekehrte Weg durch Dekodierung. So kann ein Problem mit verschiedenen Verfahren bearbeitet werden.
- Kombination von Verfahren
  In einem Experiment können mehrere unterschiedliche Verfahren kombiniert werden. Individuen können von verschiedenen Verfahren bearbeitet

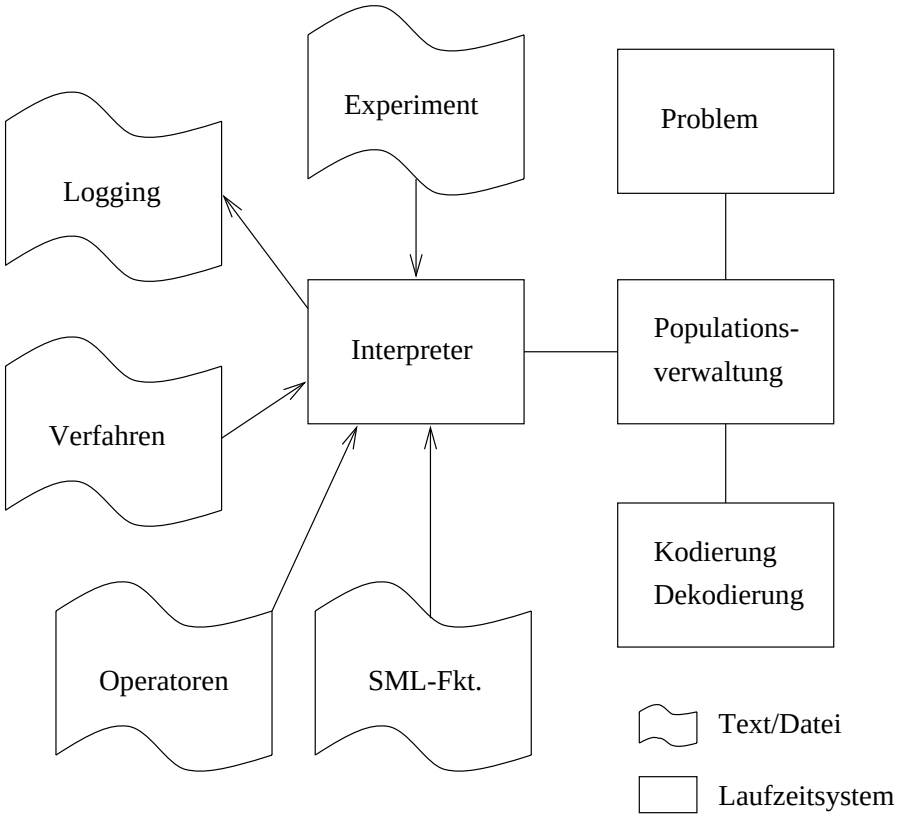


Abb. 4.1: Aufbau des Gesamtsystems

werden. So kann z.B. zuerst ein Genetischer Algorithmus ein gutes Individuum finden, das anschließend durch ein Hill-Climbing-Verfahren verbessert wird.

- Mehrere Populationen
  Ein Experiment kann mehrere Populationen bearbeiten. Es ist möglich, dasselbe oder verschiedene Verfahren auf Populationen mit unterschiedlicher Kodierung anzuwenden. Individuen können zwischen Populationen ausgetauscht werden.

- Toolbox-Charakter
  Das System ist so ausgelegt, daß Operatoren, Verfahren und Kodierungen einfach wiederverwendet werden können. Durch Bibliotheken ist es möglich, daß derselbe Operator je nach Kontext verschiedene Aufgaben erfüllt und auf verschiedenen Typen von Individuen arbeitet.

  Experimente, Verfahren und Operatoren werden in einer einfachen imperativen Sprache formuliert und können auf bereits vorbereitete Operatoren und Verfahren zurückgreifen. So ist es leicht möglich, eigene Ideen in Experimente umzusetzen.

- Zugangsmöglichkeiten auf verschiedenen Ebenen
  Das System kann durch neue Experimente, Verfahren und Operatoren in der imperativen Sprache erweitert werden. Es ist auch möglich, neue Probleme und Kodierungen als SML-Programme zu erstellen. Außerdem kann das Laufzeitsystem um neue Atomtypen und Grobstrukturen erweitert werden.

4.2 Modulgruppen

Individuen Ein Individuum besteht aus einer Liste fester Länge von Zellen. Jede Zelle enthält eine Grobstruktur, die mit Atomen gefüllt ist. Grobstrukturen sind z.B. ein einzelnes Atom, ein Paar von Atomen, aber auch eine Liste variabler Länge oder ein Baum usw. Die Atome enthalten die Daten, also die Information, die optimiert werden soll. Atome sind z.B. ganze Zahlen, reelle Zahlen oder Bits (Abb. 4.2).

Simple (real)	Pair (real, int)	List (real)	Permutation (14)	Simple (real)	Simple (real)

Abb. 4.2: Beispiel-Individuum aus sechs Zellen

Jedes Individuum repräsentiert einen Punkt im Lösungsraum der Problemfunktion. Die Werte, die der Problemfunktion übergeben werden, heißen „Problemparameter“; die von der Problemfunktion gelieferte Zahl „Fitneßwert“.

Das System unterscheidet zwischen der Darstellung, die die Problemfunktion erwartet, und der in evolutionären Verfahren verwendeten: Die Problemparameter eines Individuums stehen i.a. nicht direkt in ihm. Sie sind vielmehr kodiert, d.h.

ihre Darstellung wird auf das Verfahren angepaßt. So kann z.B. ein Integerwert in eine Folge von Bits kodiert werden. Das kodierte Individuum wird in Anlehnung an die Biologie „Genotyp“ genannt, das unkodierte „Phänotyp“. Durch dieses Konzept ist eine Migration zwischen Populationen verschiedener Kodierungen möglich (s. Populationen).

Probleme Mit Evolutionären Verfahren sollen für Parameter-Optimierungsprobleme möglichst gute Lösungen gefunden werden. Das Problem wird durch eine Problemfunktion dargestellt, die für ein unkodiertes Individuum (einen Phänotyp) eine Zahl (den Fitneßwert) liefert. Die Art des Problems oder die Art der Berechnung des Fitneßwerts ist nicht eingeschränkt, jedoch wird vereinfachend eine *Minimierungsaufgabe* angenommen[1]. Ferner wird eine Initialisierungsfunktion gefordert, die Phänotypen liefert. Der Bildbereich dieser Funktion definiert die zulässigen Individuen.

Kodierungen Die Problemfunktion erwartet Parameter in einer zu ihr passenden Form, dem Phänotyp. Diese Form kann aber für ein Verfahren ungeeignet sein, da die darin verwendeten Operatoren bestimmte Anforderungen an das Aussehen eines Individuums stellen. So könnte das Problem beispielsweise reelle Zahlen erwarten, das Verfahren jedoch nur auf Bit-Ketten arbeiten.

Um nun nicht eine neue Problemfunktion schreiben zu müssen, d.h. den Phänotyp an das Verfahren anzupassen, kann der Phänotyp *kodiert* werden: Er wird in einen Genotyp transformiert, dessen Darstellung an das Verfahren angepaßt ist (Abb. 4.3).

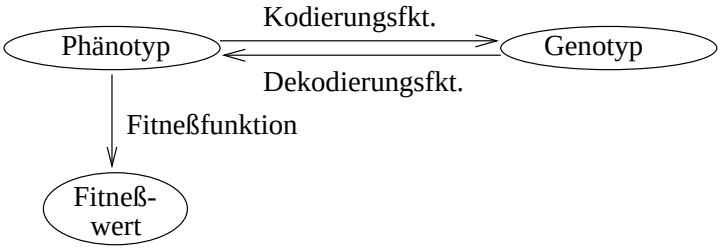


Abb. 4.3: Erzeugung eines Phänotyps, Kodierung, Dekodierung, Berechnung der Fitneß

Eine Kodierung legt fest, wie die Zellen des Phänotyps in Zellen des Genotyps überführt werden, und umgekehrt.

Populationen Eine Population ist eine Menge gleich kodierter Individuen, die als zusammengehörig betrachtet werden. Verfahren arbeiten auf einer Population[2], wobei jeweils Operatoren wie Selektion diese Population betrachten.

Das System kann für dasselbe Problem mehrere Populationen verwalten, deren Individuen verschieden kodiert sein können. Es ist möglich, ein Individuum von

[1] Jedes Maximierungsproblem kann durch Negation in ein Minimierungsproblem transformiert werden.

[2] Die Population kann auch aus einem Individuum bestehen

einer Population in eine andere zu überführen, zu „migrieren“. Dabei wird es automatisch an die Kodierung der Zielpopulation angepaßt, d.h. ein Individuum wird zuerst dekodiert und dann wieder kodiert.

LEA Im System ist die imperative Programmiersprache `LEA` implementiert. Experimentoperatoren, Verfahren und Operatoren werden in `LEA` geschrieben. `LEA` ist dazu gedacht, in einer einfachen Sprache Verfahren zu definieren. Damit ist die Einschränkung verbunden, daß mit `LEA` auf Individuen nur als Ganzes zugegriffen werden kann. Will ein Verfahren Zellen eines Individuums verändern, so muß es SML-Funktionen verwenden. Das System stellt Bibliotheken mit solchen Funktionen zur Verfügung, die zu komplexen Operatoren zusammengesetzt werden können.

Experimentoperator Ein Experiment wird durch Angabe eines Experimentoperators (ein `LEA`-Programm, das oft auch als Experimentdefinition bezeichnet wird) gestartet. Im Experimentoperator werden das Problem festgelegt, die Populationen samt Kodierungen angegeben und die Verfahren deklariert. Bsp.:

```
EXPERIMENT Threshold_TSP;
PROBLEM = "TSP";
POPULATIONS
  Pop CODED "Identity" LOG "pop1" = RANDOMPOP(1);
OPERATORS
  Optimate = threshold(T : -4.0);
```

Im Anweisungsteil des Experimentoperators wird die Reihenfolge der Verfahrensaufrufe bestimmt; hier können auch Migrationen von Individuen zwischen Populationen vorgenommen werden.

Verfahren Verfahren sind `LEA`-Operatoren, die in der Regel ganze Populationen erhalten und liefern. Sie implementieren eine *Strategie*, nach der das Optimum gesucht wird. Dies ist i.allg. ein iterativer Algorithmus, der verschiedene Operatoren und Funktionen aufruft. Bekannte Verfahren sind Genetische Algorithmen, Evolutionsstrategien, Threshold Algorithmen und Simulated Annealing.

Operatoren In Evolutionären Verfahren werden Individuen bzw. Populationen bearbeitet. Dies wird durch in `LEA` geschriebene Operatoren implementiert, die z.B. einzelne Individuen mutieren oder zwei Individuen kreuzen. Operatoren können beliebige Mengen von Individuen ineinander überführen.

Operatoren können nicht direkt auf einzelne Zellen eines Individuums zugreifen. Sie erhalten nur ganze Individuen, können diese aber an beliebige Funktionen übergeben und von ihnen erhalten. Das System stellt Bibliotheken mit SML-Funktionen bereit, die aus Operatoren aufgerufen werden und die auf Teile von Individuen zugreifen und diese ändern können. Die SML-Funktion `"mutate_real_normal"` der Bibliothek `SimpleCellLib` kann z.B. einen bestimmten Zellentyp mutieren.

Es ist möglich, Parameter zu deklarieren, die zwischen Aufrufen weiter existieren (Beispiel: `PARAMETER REAL T = (-2.0, -4.0, 4.0, "Temperatur")`). Parameter steuern das Verhalten eines Operators, sie stellen z.B. Schrittweiten oder Grenzen dar. Diese Parameter verfügen über einen Standardwert, einen Wertebereich und eine textuelle Beschreibung ihrer Funktion.

Es ist möglich, innerhalb von Experimenten Verfahren und Operatoren aufzurufen. Verfahren und Operatoren können Operatoren aufrufen. Somit entsteht ein „Operatorbaum“, an dessen Wurzel der Experimentoperator steht (Abb. 4.4).

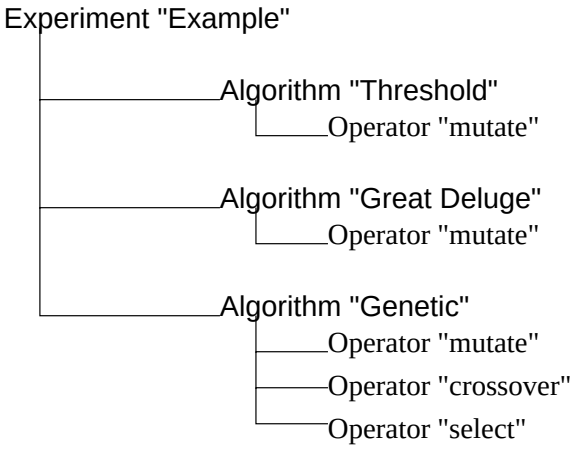


Abb. 4.4: Operatorbaum

Ein Operator kann außerdem Bibliotheken angeben, aus denen Funktionen entnommen werden sollen. Beim Aufruf einer Funktion wird in diesen Bibliotheken nach der Funktion gesucht und sie wird ausgeführt, wenn sie gefunden wurde.

Bibliotheken Funktionen werden in Bibliotheken zusammengefaßt. Diese Funktionen können auf Zellen von Individuen zugreifen und werden in SML geschrieben. So gibt es Funktionen, die Zellen mutieren oder zwei Individuen kreuzen.

Durch neue Bibliotheken kann der Anwendungsbereich des Systems erweitert werden, zum einen um neue Funktionen auf vorhandene Genotypen, als auch um Funktionen auf neuen Genotypstrukturen.

Interpreter Der Interpreter ist das Rückgrat des Systems, das die einzelnen Teile wie Populationsverwaltung, Problem und Kodierungen verbindet. Der Ablauf eines Experiments wird durch den Experimentoperator festgelegt, der vom Interpreter ausgeführt wird. In ihm werden nicht nur Problem und Populationen festgelegt, auch die Abfolge von Verfahren und Operationen auf Individuen wird bestimmt.

Durch die Verwendung des Interpreters kann die Experimententwicklung in verschiedene Stufen aufgeteilt werden: Es ist möglich, Vorhandenes zu verwenden, einfache Algorithmen in der Interpretersprache `LEA` zu schreiben, oder das

System selbst zu erweitern. Letzteres kann durch eine Erweiterung der SML-Bibliotheken erfolgen oder durch Änderungen am Grundsystem.

Auswertungswerkzeuge Während der Ausführung eines Experiments kann Information über Populationen und Individuen in eine Log-Datei geschrieben werden. Diese Log-Datei kann an weitere Programme übergeben werden, die die Informationen aufbereiten (Abb. 4.5); z.B. sie in eine für Menschen leicht verständliche Form bringen.

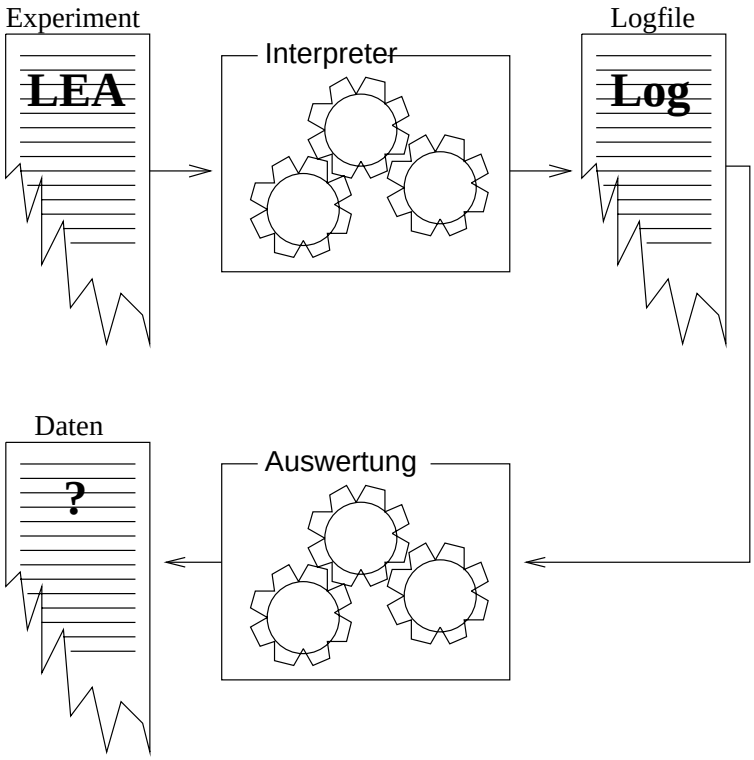


Abb. 4.5: Experiment erzeugt Logfile, welches ausgewertet wird

Es stehen Werkzeuge zur Verfügung, die die Entwicklung der Fitneß einer Population grafisch darstellen. So kann verglichen werden, wie sich die mittlere Fitneß im Vergleich zur besten oder schlechtesten entwickelt, außerdem kann man Individuen selbst anzeigen lassen.

Kapitel 5

Umsetzung der Konzepte

In Kapitel 3 sind die Ergebnisse der Spezifikationsphase beschrieben. In diesem Abschnitt wird die Umsetzung dieser Anforderungen in einen Entwurf für GENOM dokumentiert. Die Beschreibung ist, entsprechend der in den Anforderungen festgelegten Aufteilung des Systems in die drei Teile Kodierung, Populationsverwaltung und Interpreter gegliedert. Die Sprache LEA sowie der Aufbau von Experimenten, Verfahren und Operatoren werden jeweils in einem eigenen Kapitel beschreiben.

5.1 Individuen und Kodierung

Die Eigenschaften von Individuen, die mit evolutionären Algorithmen bearbeitet werden, können durch einen festen Satz von Parametern beschrieben werden. Gleichartige Individuen, die zusammen eine Population bilden, haben folglich mindestens eine Gemeinsamkeit: sie werden mit gleich vielen Parametern beschrieben. Aus diesem Grund sind Individuen Listen fester Länge von Parametern, die im weiteren **Zellen** genannt werden.

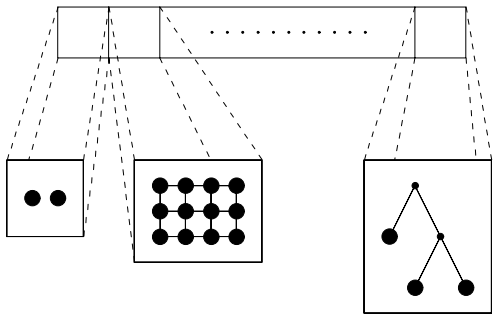


Abb. 5.1: Beispiel eines Individuums

Jede Zelle besteht aus einer **Grobstruktur**, die mit **Atomen** gefüllt sind. Atome sind die elementaren Bestandteile der Individuen und können unter anderen

reelle Zahlen, bool'sche Werte oder Permutationen sein. Diese Atome können mit Hilfe von Grobstrukturen angeordnet, beispielsweise in einer Liste, in einem Baum, einer Matrix oder in einer anderen beliebigen Struktur, deren einzige Einschränkung ist, daß in ihr der Begriff einer Position wohldefiniert ist. In Abbildung 5.1 ist ein Beispiel für ein Individuum, dessen Zellen verschiedene Grobstrukturen enthalten: in der ersten Zelle ein Paar von Atomen, in der zweiten eine Matrix und in der letzten ein binärer Baum.

Eine Reihe von Evolutionären Verfahren – die Genetischen Algorithmen – arbeiten nach einem Prinzip, das an die Vererbung bei Lebewesen angelehnt ist. Zur Berechnung der Qualität einer Lösung wird wie bei anderen Verfahren der Punkt im Lösungsraum verwendet, alle Veränderungen werden aber an kodierten Lösungen vorgenommen. Der Punkt im Lösungsraum entspricht einem Lebewesen, die kodierte Lösung seiner DNS. Die unkodierten Lösungen werden daher auch als Phänotypen, ihre kodierten Darstellungen als Genotypen bezeichnet.

Auf Individuen bestehen zwei Sichtweisen: der **Phänotyp** und der **Genotyp**; der Phänotyp ist die Sicht auf das Individuum von der Problemseite, der Genotyp von der Verfahrensseite. Beide Sichtweisen können mit der Kodierungs- (Phänotyp nach Genotyp) bzw. der Dekodierungsfunktion (Genotyp nach Phänotyp) ineinander überführt werden.

5.1.1 Atome

Das System stellt eine Reihe von Atomtypen zur Verfügung; diese Typen können verschiedenartige Parameter besitzen, wie beispielsweise das zulässige Intervall. Dementsprechend werden die Atome in Klassen aufgeteilt. Realisiert wurden bisher Atome ohne zusätzliche Parameter (reelle und ganze Zahlen, bool'sche Werte) sowie Atome mit zugehörigem Intervall (reelle und ganze Zahlen). In den Grobstrukturen dürfen beliebige Atome vorkommen. Daher muß ein Vereinigungstyp über alle vorhandenen Atomtypen gebildet werden:

```
structure AtomTypes =
    struct
        datatype atom_type =
                 real_atom of real
               | int_atom of int
               | bool_atom of bool;
               | bound_real_atom of (real * (real * real))
               | bound_int_atom of (int * (int * int));
    end
```

Zu allen Atomtypen sind Funktionen definiert, deren Art jeweils von der Klasse, in die der Typ eingeordnet ist. Beispielsweise stellt jeder Atomtyp der Klasse ohne zusätzliche Parameter (reelle und ganze Zahlen, bool'sche Werte) folgende Funktionen zur Verfügung (nähere Informationen in A.6:

```
signature ATOM =
    sig
        type base; (* Typ des Atoms *)
```

```
        val name: string; (* Bezeichner fuer diesen Atomtyp *)
        val base2atom: base -> AtomTypes.atom_type;
                       (* nach Vereinigungstyp *)
        val atom2base: AtomTypes.atom_type -> base;
                       (* von Vereinigungstyp *)
        val base2string: base -> string; (* nach string *)
        val string2base: string -> base; (* von string *)
        val init_random: real -> base; (* Zufallswert *)
    end;
```

Auf manche Funktionen der Atome kann einheitlich zugegriffen werden. Dies ermöglicht die Struktur **UnionOfAtoms** mit folgender Signatur:

```
signature UNIONOFATOMS =
    sig
        type atom_type;
        val atom2string: atom_type -> string;
        val string2atom: string -> atom_type;
    end;
```

Aufbauend auf den Vereinigungstyp über den Atomen können nun Zellen definiert werden, die Atome enthalten. Alle Zellen müssen folgende Signatur erfüllen:

```
signature CELL =
  sig
    type 'a rawstructure;
    type index;
    val name: string;
    val init: ((index -> AtomTypes.atom_type) * index list)
                      -> AtomTypes.atom_type rawstructure;
    val get_element: (AtomTypes.atom_type rawstructure * index)
                           -> AtomTypes.atom_type;
    val set_element: (AtomTypes.atom_type rawstructure *
                      AtomTypes.atom_type * index)
                           -> AtomTypes.atom_type rawstructure;
    val cell2rawstructure: CellTypes.cell_type
                           -> AtomTypes.atom_type rawstructure;
    val rawstructure2cell: AtomTypes.atom_type rawstructure
                           -> CellTypes.cell_type;
    val cell2string: AtomTypes.atom_type rawstructure
                                                  -> string;
    val string2cell: string
                           -> AtomTypes.atom_type rawstructure;
  end;
```

Erläuterung:

- 'a rawstructure ist die Grobstruktur, in der die Atome angeordnet sind.

- index ist der Typ, mit dem die Position innerhalb der Grobstruktur festgelegt wird.
- name ist der Bezeichner für die Struktur.
- get_element(r, i) liefert das Atom an der i-ten Stelle in r.
- set_element(r, a, i) setzt das i-te Atom in r auf a.
- cell2rawstructure und rawstructure2cell sind Konvertierungsfunktionen zwischen dem Zelltyp und dem Vereinigungstyp über alle Zellen.
- cell2string und string2cell sind Konvertierungsfunktionen zwischen Zeichenketten und Zellen.

Ebenso wie für Atome existiert auch für Zellen ein Vereinigungstyp. In diesem Beispiel werden Zellen mit einem einzelnen Atom (simple_cell), Zellen mit einem Paar von Atomen (pair_cell) und Zellen, die aus einer Liste von Atomen bestehen (list_cell), zu einem Vereinigungstyp zusammengefaßt:

```
structure CellTypes =
  struct
    datatype cell_type =
      simple_cell of AtomTypes.atom_type
    | pair_cell of (AtomTypes.atom_type * AtomTypes.atom_type)
    | list_cell of (AtomTypes.atom_type list);
  end
```

Auch für diesen Vereinigungstyp gibt es eine Struktur, um auf Teile der Funktionen der Zellen einheitlich zugreifen zu können. Dies wird mit der Struktur **UnionOfCells** erreicht:

```
signature UNIONOFCELLS =
    sig
        type cell_type;
        val cell2string: cell_type -> string;
        val string2cell: string -> cell_type;
    end;
```

Individuen bestehen aus einer Liste von Zellen und dem Fitneßwert. Dieser ist nur bekannt, wenn seit seiner letzten Berechnung die Zellen nicht verändert wurden. Mit Hilfe der Struktur **Individuum** werden Funktionen bereitgestellt, um einzelne Zellen zu lesen, zu ändern, sowie das Individuum in eine Zeichenkette zu konvertieren. Diese Struktur erfüllt folgende Signatur:

```
signature INDIVIDUUM =
    sig
        type individuum_type;
        val new_individual: individuum_type;
        val get_fitness: individuum_type -> real option;
        val set_fitness: individuum_type * real
```

```
                                        -> individuum_type;
    val init: ((int -> UnionOfCells.cell_type) * int)
                                        -> individuum_type;
    val get_cell: (individuum_type * int)
                                    -> UnionOfCells.cell_type;
    val set_cell: (individuum_type * UnionOfCells.cell_type
                  * int)                -> individuum_type;
    val indiv2string: individuum_type -> string;
    val string2indiv: string -> individuum_type;
end;
```

Erläuterung:

- individuum_type ist der Datentyp des Individuums.
- new_individual ist ein leeres Individuum.
- get_fitness ermittelt die letzte bekannte Fitneß oder stellt fest, daß keine gültige Fitneß vorliegt.
- set_fitness setzt die gespeicherte Fitness auf einen neuen Wert. Bis Zellen verändert werden, kann diese Fitness mit get_fitness ausgelesen werden.
- init(f, n) liefert ein Individuum mit n Zellen, wobei die Zellen $1, \ldots, n$ mit $f(1), \ldots, f(n)$ belegt sind.
- get_cell(i, n) liefert die n-te Zelle aus dem Individuum i.
- set_cell(i, c, n) setzt die n-te Zelle in i auf cdn Wert c.
- indiv2string und string2indiv sind Konvertierungsfunktionen zwischen Zeichenketten und Individuen.

5.1.2 Kodierungs- und Dekodierungsfunktion

Eine Reihe von Evolutionären Verfahren — die Genetischen Algorithmen — arbeiten nach einem Prinzip, das an die Vererbung bei Lebewesen angelehnt ist. Zur Berechnung der Qualität einer Lösung wird wie bei anderen Verfahren der Punkt im Lösungsraum verwendet, alle Veränderungen werden aber an kodierten Lösungen vorgenommen. Der Punkt im Lösungsraum entspricht einem Lebewesen, die kodierte Lösung seiner DNS. Die unkodierten Lösungen werden daher auch als Phänotypen, ihre kodierten Darstellungen als Genotypen bezeichnet. Auch bei anderen Verfahren kann eine Kodierung sinnvoll eingesetzt werden, z.B. lassen sich durch die Kodierung die von Evolutionsstrategieen benötigten Strategieparameter vor der Berechnung der Qualität der Lösung ausblenden. Die Qualitätsfunktion (Fitneßfunktion) — die diese Parameter nicht berücksichtigen soll — kann so unabhängig vom Verfahren sein.

Die Kodierung in GENOM nimmt diese Umwandlung von Phäno- in Genotypen und umgekehrt vor. Phäno- und Genotypen sind Tupel, deren Elemente aus einem oder mehreren Atomen bestehen, die in verschieden Strukturen angeordnet

sein können. Bei diese Strukturen können selbst einfache Tupel sein, aber auch Graphen oder Matrizen variabler Größe, so daß sich auch Anwendungen wie Genetisches Programmieren realisieren lassen. Wie die Tupel kodiert werden, wird durch ein Kodierungsschema beschrieben. Bestandteile dieses Schemas sind die elementaren Kodierungsschemata, die Strukturen über Atomen in andere Strukturen überführen. Da wir dem Benutzer auch die Möglichkeit bieten wollen, weitere Strukturen ins System einzufügen, existiert auch die Möglichkeit, das System nachträglich um weitere elementare Kodierungsschemata zu erweitern.

Diese Beschreibung gliedert sich in mehrere Abschnitte: nach einer Einführung in den Aufbau der Kodierung werden zunächst Atom und Zellen sowie die darauf definierten Funktionen beschrieben. Anschließend werden die Kodierungs- und Dekodierungsfunktion sowie Kodierungsschemata und elementare Kodierungsschemata erläutert.

Der fixe Aufbau von Phäno- und Genotyp ermöglicht es, die Kodierungs- und Dekodierungsfunktion nach einem festen Schema zu konstruieren. Grundlage bilden die **elementaren Kodierungsschemata**, die Listen von Grobstrukturen aus Atomen in andere Listen von Grobstrukturen umwandeln.

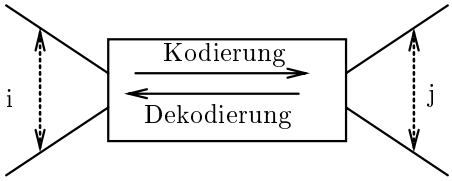


Abb. 5.2: elementares Kodierungsschema

Dieses Schema besteht im Wesentlichen aus einer Kodierungs- und einer Dekodierungsfunktion; zusätzlich enthält das Schema je eine Liste von Namen von Grobstrukturen. Dabei muß durch die Konstruktion der beiden Funktionen gewährleistet sein, daß die Dekodierungsfunktion die Umkehrfunktion der Kodierungsfunktion ist.

```
type elementary_coding_scheme
      = {in_cell_names: string list,
         out_cell_names: string list,
         coding: CellTypes.cell_type list
                  -> CellTypes.cell_type list,
         decoding: CellTypes.cell_type list
                    -> CellTypes.cell_type list}
```

Die Werte und Funktionen, mit denen dieses Schema gefüllt wird, sind in Strukturen mit folgender Signatur vereinbart:

```
signature ELEMENTARY_CODING_SCHEME =
    sig
        val in_cell_names: string list
        val out_cell_names: string list
        val coding: CellTypes.cell_type list
```

```
                    -> CellTypes.cell_type list
        val decoding: CellTypes.cell_type list
                      -> CellTypes.cell_type list
    end;
```

Für einige Anwendungen, z.B. Kodierung reeller Zahlen aus einem vorgegebenen Intervall, ist dieses feste Schema wenig geeignet: Für jedes Intervall muß ein neues Schema erstellt werden. Deshalb gibt es die Möglichkeit, parametrisierte elementare Kodierungsschemata zu schreiben.

```
signature PARAM_ELEMENTARY_CODING_SCHEME =
    sig
        type parameter
        val param_in_cell_names: parameter -> string list
        val param_out_cell_names: parameter -> string list
        val param_coding: parameter
                          -> (CellTypes.cell_type list
                              -> CellTypes.cell_type list)
        val param_decoding: parameter
                            -> (CellTypes.cell_type list
                                -> CellTypes.cell_type list)
    end;
```

Durch Einsezten des Parameters lassen sich dann Werte und Funktionen für ein elementares Kodierungsschema erzeugen.

Um diese Signaturen in Records des Typs `elementary_coding_scheme` zu überführen, existieren die zwei Funktoren `GetElementaryCodingScheme` und `GetParamElementaryCodingScheme`, die Strukturen der Signaturen `ELEMENTARY_CODING_SCHEME` bzw. `PARAM_ELEMENTARY_CODING_SCHEME` in Strukturen mit den Signaturen

```
signature GET_ELEMENTARY_CODING_SCHEME =
    sig
        structure Coding: CODING
        val elemcodscheme: Coding.elementary_coding_scheme
    end;
```

für elementare und

```
signature GET_PARAM_ELEMENTARY_CODING_SCHEME =
    sig
        structure Coding: CODING
        type parameter
        val paramelemcodscheme:
            parameter -> Coding.elementary_coding_scheme
    end;
```

für parametrisierte elementare Kodierungsschemata umwandeln. Eine SML-Datei, die ein elementares Kodierungsschema definiert, besteht damit aus

- einer Struktur zur Signatur `ELEMENTARY_CODING_SCHEME` bzw. `PARAM_ELEMENTARY_CODING_SCHEME`
- Einer Zeile, die den Funktor `GetElementaryCodingScheme` bzw. `GetParamElementaryCodingScheme` auf die vorher definierte Struktur anwendet.

Das elementare Kodierungsschema kann aus diesen Strukturen direkt übernommen werden.

Aus diesen elementaren Kodierungsschemata wird das **Kodierungsschema** zusammengesetzt.

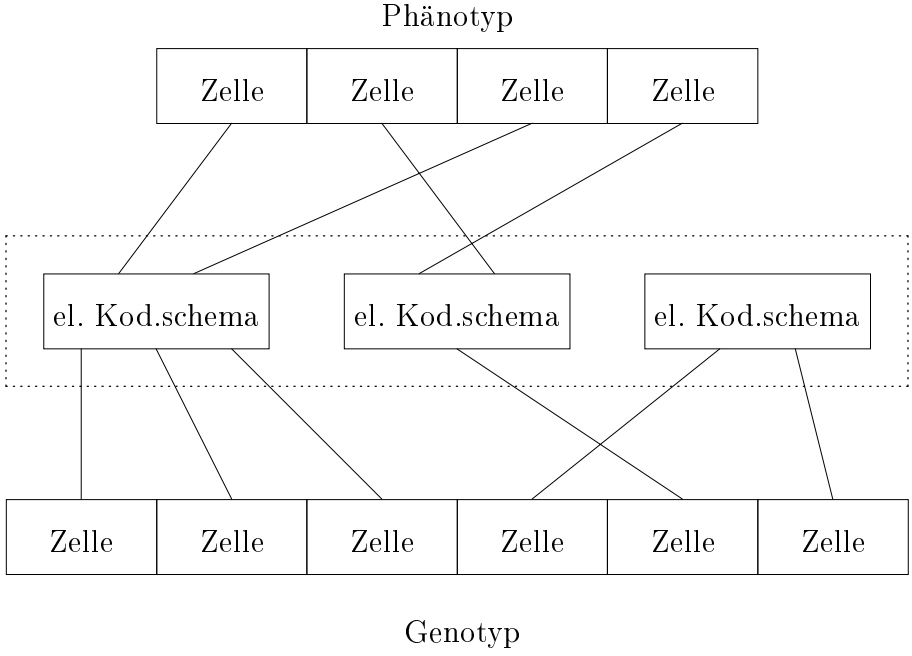


Abb. 5.3: Kodierungsschema

In diesem Schema wird jeder Zelle des Phänotyps eindeutig ein elementares Kodierungsschema und dort wiederum eindeutig eine Position in der Eingangsliste sowie jeder Zelle des Genotyps eindeutig ein elementares Kodierungsschema und dort wiederum eindeutig eine Position in der Ausgangsliste zugeordnet. Das Kodierungsschema besteht aus einem Namen und einer Liste von Tripeln (K, l_i, l_o), in dem K ein elementares Kodierungsschema und l_i, l_o Listen von Indizes in der Liste des Phäno- bzw. Genotyps sind. Die Eindeutigkeit wird durch die Konsistenzbedingung gewährleistet, daß jeder Index des Phänotyps genau einmal in einem der l_i und jeder Index des Genotyps genau einmal in einem der l_o vorkommt. Zudem muß sichergestellt sein, daß die elementaren Kodierungsfunktionen die übergebenen Grobstrukturen verabeiten können. Dazu können die Namen aus in_cell_names bzw. out_cell_names mit dem Namen der entsprechenden Zelle aus Cell.name verglichen werden. Der Benutzer muß sicherstellen,

daß auch die Atome die von den elementaren Kodierungsfunktionen geforderten Inhalte haben.

```
type coding_scheme = (string * (elementary_coding_scheme
                                * int list * int list) list)
```

Die Kodierungs- und Dekodierungsfunktionen sind in der Struktur Coding zusammengefaßt:

```
signature CODING =
    sig
        type elementary_coding_scheme
        type coding_scheme

        val codeind: PhenoType.individuum_type * coding_scheme
                     -> GenoType.individuum_type
        val decodeind: GenoType.individuum_type * coding_scheme
                       -> PhenoType.individuum_type
        val samecod: coding_scheme * coding_scheme -> bool
        val valcod: coding_scheme * coding_scheme -> bool
    end;
```

Erläuterung:

- codeind() kodiert einen Phänotyp mit dem angegebenen Kodierungsschema in einen Genotyp.
- decodeind() dekodiert umgekehrt einen Genotyp.
- samekod() prüft durch Vergleich der Namen, ob die beiden Kodierungsschemata identisch sind.
- valcod() führt Teile der o.g. Konsistenzprüfungen durch. Geprüft wird, ob das Kodierungsschema jeder Zelle zwischen Zelle 1 und der Zelle mit der höchsten Nummer auf Phäno- bzw. Genotypseite genau ein elementares Kodierungsschema zuordnet.

5.1.3 Kodierungsfunktionen

Die meisten der in Anhang A.4 beschriebenen elementaren Kodierungsschemata verwenden gebräuchliche Kodierungen wie Standardbinär- oder Graykodierungen natürlicher Zahlen. Zwei spezielle Funktionen zur Kodierung von Permutationen und reeller Zahlen werden hier definiert. Beschreibungen der Funktionen finden sich auch in [Cla96] bzw. [JW95, Seite 46] und [AJK+95, Seite 113].

5.1.3.1 Kodierung von Permutationen

Die Bijektion f zur Kodierung von Permutationen überführt eine Permutation der Länge n (dargestellt durch ein Tupel natürlicher Zahlen) in ein Tupel natürlicher Zahlen:

$f : A \to B$

mit

$$\begin{aligned} A &= \{(x_1,\ldots,x_n) \in \mathbb{N}^n \mid \forall i,j : 1 \leq x_i \leq n, i \neq j \Rightarrow x_i \neq x_j\} \\ B &= \{(x_1,\ldots,x_n) \in \mathbb{N}^n \mid \forall i : 1 \leq x_i \leq n-i+1\} \end{aligned}$$

Der Algorithmus zur Kodierung wird hier in Pseudocode beschrieben. Die Permutation ist in der Variable p abgelegt, die kodierte Permutation wird in k gespeichert.

$z := (1,\ldots,n)$
$k := ()$
`loop`
$(p_1,\ldots,p_n) := p$
$(z_1,\ldots,z_n) := z$
$(k_1,\ldots,k_m) := k$
bestimme i mit $z_i = p_1$
$p := (p_2,\ldots,p_n)$

$$z := \begin{cases} (z_2,\ldots,z_n), \text{ falls } i = 1 \\ (z_1,\ldots,z_{n-1}, \text{ falls } i = n \\ (z_1,\ldots,z_{i-1},z_{i+1},\ldots,z_n) \text{ sonst} \end{cases}$$

$k := (k_1,\ldots,k_m,z_i)$
`until` $p = ()$

Die einzelnen Komponenten von k können dann z.B. noch standardbinär kodiert werden. Standardmutations- und -rekombinationsoperatoren können dann aber bei Anwendung auf Elemente aus B Tupel erzeugen, die selbst nicht in B liegen und daher mit f^{-1} nicht dekodiert werden können. Mit der Funktion

$$\begin{aligned} g : \mathbb{Z}^n &\rightarrow B \\ (x_1,\ldots,x_n) &\mapsto ((x_1 \bmod n)+1,\ldots,(x_n \bmod 1)+1) \end{aligned}$$

können solche Tupel auf Elemente der Menge B abgebildet werden. Es gilt: $g(x) = x$, falls $x \in B$. Dekodiert wird dann mit

$$f^{-1} \cdot g.$$

5.1.3.2 Kodierung reeller Zahlen

Reelle Zahlen werden durch eine Funktion $f_{a,b,l}$ kodiert, die eine reelle Zahl zwischen a und b auf ein Bittupel der Länge l abbildet:

$$\begin{aligned} f_{a,b,l} : \{x \in \mathbb{R} \mid a \leq x \leq b\} &\rightarrow \{0,1\}^l \\ x &\mapsto (b_1,\ldots,b_l) \end{aligned}$$

mit

$$x = a + \frac{b-a}{2^l - 1} \cdot \sum_{i=1}^{l} b_i \cdot 2^{i-1}.$$

5.2 Populationsverwaltung

Das System bietet die Möglichkeit, Verfahren zu definieren, die auf mehreren Populationen arbeiten. Der Zugriff auf die Populationen wird durch die Populationsverwaltung bereitgestellt. Neben den einzelnen Populationen enthält die Populationsverwaltung auch die Informationen über das betrachtete Problem (Problemraum, Fitneß).

5.2.1 Populationen

Populationen bestehen aus einer Menge von Individuen (als Genotyp), der verwendeten Kodierung, sowie dem Generationszähler. Die Individuenmenge wird als Liste verwaltet, die Kodierung als das zugehörige Schema, und der Generationszähler als ganze Zahl (`integer`).

Jeder Population ist ein Index zugeordnet. Zusammen mit diesen Indizes werden die Populationen in einer Liste verwaltet. Die genaue Definition dieser Datentypen ist:

```
type LogFile = string
type GenCounter_type = int
type KSrec_type = Coding.coding_scheme
type Ind_type = Individuum.individuum_type
type Pop_type = (LogFile * GenCounter_type * KSrec_type *
                 Ind_type list)
type PopId_type = int
```

LogFile ist der Typ des Protokolldateinamens, GenCounter_type der Typ des Generationszählers, KSrec_type der Typ des Kodierungsschemas, Ind_type der Typ eines Individuums, Pop_type der Typ einer Population und PopId_type der Typ des Populationsindex.

5.2.2 Problemstruktur

Ein Problem besteht aus einer Menge möglicher Phänotypen und einer Fitnessfunktion Evaluate, die die Qualität der Phänotypen bewertet. Die Menge der Phänotypen wird durch den Bildbereich der Funktion Init_Random definiert, welche zufällig erzeugte Phänotypen liefert:

```
signature PROBLEM =
  sig
    val Evaluate: (PhenoType.individuum_type -> real) ref
    val Init_Random: (unit -> PhenoType.individuum_type) ref
  end
```

Sowohl die Funktion Init_Random als auch die Fitnessfunktion werden als Referenztyp (ref) verwaltet. Um diese Referenzen zu setzen, existiert die Funktion SetProblem:

```
val Set_Problem: (unit -> PhenoType.individuum_type) *
                 (PhenoType.individuum_type -> real) -> bool
```

Set_Problem(init_indiv, fitness) belegt in der Struktur Problem die Funktionen Evaluate und Init_Random mit Referenzen auf die Funktionen init_indiv und fitness.

5.2.3 externe Individuen

Wird ein Individuum an den Interpreter übergeben, so erhält dieser ein „externes Individuum". Dieses ist ein Paar, (i, Ind), und führt neben dem Individuum in kodierter Form Ind noch den Index der Population i mit, aus welcher es stammt. Beim Zurückschreiben eines externen Individuums in eine Population wird überprüft, ob der Index der Population mit dem beim Individuum gespeicherten Index übereinstimmt; falls sie nicht übereinstimmen, wird das Individuum umkodiert.

Der lesende und schreibende Zugriff auf einzelne Individuen erfolgt über ihre Position in der Individuenliste und den Index ihrer Population mit den Funktionen GetextInd, SetextInd, AddextInd und InsextInd. Ferner gibt es noch die Möglichkeit, Listen von Individuen zu erhalten bzw. zu übergeben (GetextPop, SetextPop), sowie die Finteß eines Individuums zu berechnen:

```
type extInd_type = (PopId_type * Ind_type ref)
type extIndL_type  = extInd_type list
val AddextInd: PopId_type * extInd_type -> PopId_type
val SetextInd: PopId_type * int * extInd_type -> PopId_type
val InsextInd: PopId_type * int * extInd_type -> PopId_type
val GetextInd: PopId_type * int -> extInd_type
val GetextPop: PopId_type -> extIndL_type
val SetextPop: PopId_type * extIndL_type -> PopId_type
val Fitness: extInd_type -> real
```

Erläuterungen

- extInd_type ist der Typ eines externen Individuums. Es enthält neben der Nummer seiner Population eine Referenz auf das zugehörige Individuum.
- extIndL_type ist eine Liste externer Individuen.
- AddextInd (PopId, extInd) fügt das (externe) Individuum extInd an die Population mit der Nummer PopId an.
- SetextInd (PopId, i, extInd) ersetzt das ite Individuum in der Population Nummer PopId durch extInd.
- InsextInd(PopId, i, extInd) fügt das Individuum extInd in die Population Nummer PopId an i-ter Stelle ein.
- GetextInd (PopId, i) Gibt aus Population PopId das i-te Individuum als externes Individuum zurück.

- GetextPop (PopId) liefert die Individuen der Population Nummer PopId als Liste externer Individuen.
- SetextPop (PopId, extIndL) erstellt die Population Nummer PopId aus der Liste externer Individuuen extIndL.
- Fitness (extInd) bewertet das externe Individuum extInd, indem es dieses dekodiert, und an die Funktion Problem.Evaluate übergibt.

5.2.4 Protokolldateien

Die Protokolldateien wurden entwickelt, um Populationen zu einem gewünschten Zeitpunkt protokollieren zu können, um:

1. Einen späteren Lauf darauf aufsetzen zu können.
2. Den Fitnessverlauf betrachten zu können.

Protokolliert wird der Generationszähler der Population, sowie alle Individuen als Phänotyp. Die Datei erhält die Endung .log zum eigentlichen Dateinamen und wird im Verzeichnis log abgelegt. Jede Population hat dabei eine eigene Protokolldatei.

```
LogPop: PopId_type -> bool
```

Die Protokolldatei wird bei jedem Aufruf von LogPop um einen Eintrag erweitert. Dieser beginnt mit „{“ und endet mit „}“. Der erste Eintrag darin ist der Generationszähler, der durch „(“ und „)“ geklammert ist. Dannach folgt die Individuenzeile, die aus dem Fitnesswert, dem Trennungszeichen zum Individuum „–“ und dem Phänotyp des Individuums als Zeichenkette kodiert besteht. Diese Zeile wird für alle Individuen der Population erzeugt und durch einen Zeilenumbruch beendet.

5.2.5 Arbeiten mit der Populationsverwaltung

5.2.5.1 Initialisierung der Populationsverwaltung

Eine Initialisierung der Populationsverwaltung erfolgt durch folgende Schritte:

```
PopHandler.Init () -> bool
```

Diese Funktion löscht alle verfügbaren Populationen und gibt bei erfolgreicher Initialiserung ein true zurück.

Als nächster Schritt muß das Problem gesetzt werden. Dieses besteht aus einer Funktion RandomInd, die ein zufällig belegtes Individuum im Phänotyp zurückgibt, und der Fitnessfunktion FitnessFkt. Beide Funktionen werden als Referenzen übergeben und dann innerhalb der Problemstruktur gespeichert. Liefert diese Funktion ein true zurück, so ist das Problem gesetzt.

```
Problem.Init (!RandomInd, !FitnessFkt) -> bool
```

5.2.5.2 Erzeugen von Populationen

Das Erzeugen einer Population erfolgt durch eine der unten beschriebenen Funktionen, die alle, neben anderen Parametern, das Kodierungsschema und den Namen der Protokolldatei als Eingabe erwarten und den Index der erzeugten Population zurückgeben.

```
val CreatePopEmpty: KSrec_type * LogFile -> PopId_type
val CreatePopLoad: KSrec_type * LogFile * string -> PopId_type
val CreatePopRandom: KSrec_type * LogFile * int -> PopId_type
```

- CreatePopEmpty (KS, LogFile) Erzeugt eine leere Population mit Kodierungsstruktur KS und Logdatei LogFile und liefert die Nummer der Population zurück.
- CreatePopLoad(KS, LogFile, LoadFile) lädt die Population aus der Datei LoadFile und initialisiert eine Population mit Kodierungsstruktur KS und Logdatei LogFile. Als Ergebnis wird die Nummer der Population zurückgegeben.
- CreatePopRandom(KS, LogFile, n) erzeugt eine Population mit n zufällig erzeugten Individuen. Die Population hat die Kodierungsstruktur KS und die Logdatei LogFile. Als Ergebnis wird die Nummer der Population zurückgegeben.

5.2.5.3 Funktionen auf Populationen

Folgende Funktionen stehen zum Arbeiten auf einer Population zur Verfügung:

```
val DelPop: PopId_type -> bool
val ErasePop: PopId_type -> PopId_type
val SizeOfPop: PopId_type -> int
val AddInd: PopId_type * Ind_type -> PopId_type
val SetInd: PopId_type * int * Ind_type -> int
val InsInd: PopId_type * int * Ind_type -> PopId_type
val DelInd: PopId_type * int -> PopId_type
val GetInd: PopId_type * int -> Ind_type
```

- DelPop (PopId) entfernt die Population PopId. Wenn dies erfolgreich durchgeführt wurde, wird true zurückgeliefert, ansonsten false.
- ErasePop (PopId) Löscht alle Individuen Population PopId.
- SizeOfPop (PopId) Gibt die Anzahl der Individuen in der Population PopId zurck.
- IncGenCounter (PopId) Erhöht den Generationszähler der Population PopId um eins.
- LogPop (PopId) Schreibt die Population raus in die Logdatei.

- AddInd (PopId, Ind) Fügt das Individuum Ind in die Population PopId ein.
- SetInd (PopId, i, Ind) ersetzt das i-te Individuum der Population PopId durch Ind.
- InsInd (PopId, i, Ind) fügt das Individuum Ind an der i-ten Stelle in die Population PopId ein.
- DelInd (PopId, i) entfernt das i-te Individuum aus der Population PopId.
- GetInd (PopId, i) Gibt das i-te Individuum der Population PopId zurück.
- Evaluate (Ind) liefert die Fitneß des Individuums Ind.

5.2.6 Berechnung der Fitneß eines externen Individuums.

Das System betrachtet Individuen auf zwei Arten: Zum einen bearbeiten Hilfsfunktionen des Laufzeitsystems das Individuum wie es in der Struktur `Individuum` festgelegt wird. Andererseits wird im Interpreter ein Individuum verwandt, daß von der Populationsverwaltung zusätzlich einen Verweis auf die Population erhält, zu der es gehört. Außerdem wird hier nur eine Referenz auf das eigentliche Individuum übergeben.

Somit kann man zwischen „externen“ und „normalen“ Individuen unterscheiden. Soll nun die Fitness eines externen Individuums ermittelt werden (s. Abb. 5.4), so muß ein Funktionsaufruf an die Populationsverwaltung erfolgen (1). Die Populationsverwaltung löst das normale Individuum aus dem externen heraus und ermittelt seine Fitneß durch die Funktion `get_fitness` der Struktur `Individuum` (2). Diese Funktion liefert zwei Arten von Resultaten (3): Entweder wurde für das Individuum bereits der Fitneßwert berechnet und das Individuum seitdem nicht mehr verändert; dann wird dieser Wert geliefert (10). Es kann aber auch sein, daß die Fitneß des Individuums bisher nicht berechnet oder das Individuum seit der letzten Berechnung verändert wurde.

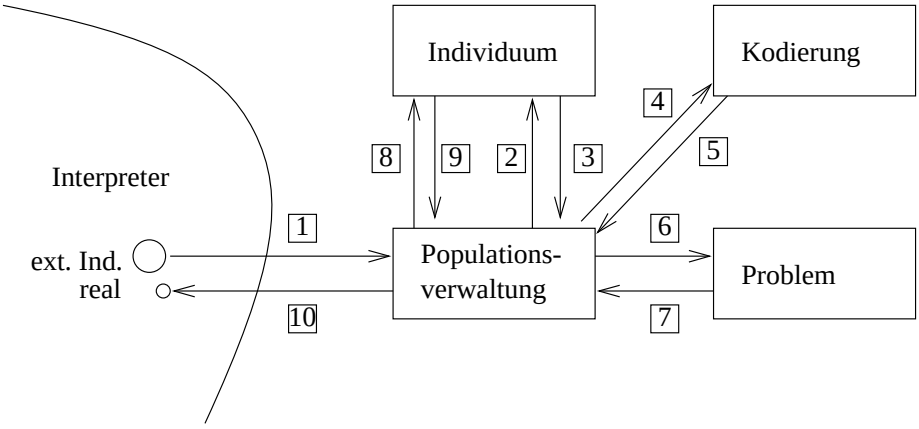


Abb. 5.4: Berechnung der Fitneß eines externen Individuums

Im letzten Fall muß nun die Populationsverwaltung das Individuum dekodieren (4, 5) und die Fitneß von Problem berechnen lassen (6, 7). Um nun den Fitneßwert im Individuum auf den neuen Stand zu bringen, muß durch einen Aufruf der Funktion `set_fitness` der Struktur `Individuum` der neue Wert gesetzt werden (8, 9). Da im Interpreterteil des Systems nur die Referenz auf das normale Individuum bekannt ist, kann in der Populationsverwaltung dieses verändert (eigentlich: ersetzt) werden. Schließlich erhält der Aufrufer den Fitneßwert zurück (10).

Dieses Verfahren ist recht umständlich, was sich aus seiner Entstehungsgeschichte erklärt: Die Trennung in normales und externes Individuum wurde eingeführt, um die Zugehörigkeit zu eines Population speichern zu können. Es wurde dann festgestellt, daß ohnehin Individuen umgewandelt werden mußten (durch die Funktionen `norm2ext` und `ext2norm` der Populationsverwaltung), so daß die Probleme beim Kopieren von Zeigern nicht entstehen. Daher war es möglich, die Speicherung der letzten Fitness beim Individuum einzuführen, durch die Verfahren mitunter bedeutend beschleunigt werden, da die Anzahl der mitunter aufwendigen Berechnungen der Fitneß verkleinert wird.

5.3 Operatoren, Parameter und Bibliotheken

5.3.1 Operatorkonzept

Um ein Experiment mit GENOM ablaufen zu lassen, existiert ein Experimentoperator, der die Verfahren aufruft. Die Verfahren verwenden Operatoren, die wiederum Funktionen aufrufen, die in Bibliotheken zusammengefaßt sind. Hierdurch wird eine hohe Wiederverwendbarkeit und eine schnelle Änderbarkeit erreicht. Man kann sich dies als Operatorbaum (siehe Abbildung 4.4) anschaulich vorstellen.

Experimentoperator Der Experimentoperator legt fest, welches Problem optimiert werden soll, indem eine SML-Datei bezeichnet wird. In dieser wird der Phänotyp und auch die Fitneßfunktion festgelegt.

Außerdem deklariert der Experimentoperator die Populationen, auf denen die einzelnen Verfahren arbeiten sollen. Dies geschieht im Experimentoperator, also „auf oberster Ebene“, um die Migration von Individuen zwischen den Populationen zu ermöglichen. Daraus folgt, daß zudem *hier* für jede Population eine Kodierung festgelegt werden muß. Dann ist eine Migration durch Dekodieren eines Individuums und anschließende Neukodierung möglich.

Der Experimentoperator hat aber noch eine weitere Aufgabe: Er definiert den Startzustand des gesamten Experiments, indem man für die Populationen Dateien anzugibt, aus denen sie geladen werden sollen. Andererseits ist es natürlich möglich, zufällige Vorbelegungen für neue Abläufe einzustellen.

Schließlich deklariert der Experimentoperator noch die Verfahren, die auf die einzelnen Populationen angewendet werden sollen. Dabei ist natürlich darauf zu achten, daß die verwendete Kodierung zum Verfahren kompatibel ist.

Verfahren Verfahren sind spezielle Operatoren, deren Aufgabe die Umsetzung eines Algorithmus ist. Hier soll nur der Ablauf gesteuert werden. Die eigentliche Arbeit soll von einzelnen Operatoren verrichtet werden.

Typische Verfahren sind z.B. das „Simulated Annealing“, der „Threshold Algorithmus“, die „Genetischen Algorithmen“ und die „Evolutionsstrategien“, deren Ablauf in der Literatur ausführlich beschrieben und bewertet wird ([GWH90, BS93, Due93]).

Operatoren Operatoren werden im System wie einzelne Werkzeuge behandelt, die sehr spezielle, mitunter komplexe Aufgaben wahrnehmen. Durch die Verwendung von Parametern (s.u.) lassen sich Operatoren in ihrer Funktionalität beeinflussen.

In Operatoren wird „die eigentliche Arbeit verrichtet“: Individuen verändern und auswählen, Funktionen aufrufen, Berechnungen durchführen. Bei geschickter Aufteilung der Aufgaben kann eine hohe Wiederverwendbarkeit und Flexibilität erreicht werden, so daß z.B. die Selektion des einen Verfahrens in einem anderen Verfahren verwendbar ist.

5.3.2 Parameterkonzept

Verfahren und Operatoren können Parameter deklarieren. Diese Parameter sind Variablen besonderer Art, da sie

- vor dem eigentlichen Aufruf des Operators gesetzt werden können,
- zwischen Operatoraufrufen ihren Wert behalten,
- über einen Wertebereich und
- einen Standard- (Default-) Wert verfügen und
- eine textuelle Beschreibung besitzen.

Wird ein Operator in einem anderen Operator (auch im Experimentoperator) deklariert, so können dabei den Parametern Werte zugewiesen werden, die im Wertebereich des Parameters liegen müssen. Wird ein Parameter nicht auf diese Weise initialisiert, so wird ihm vor dem ersten Aufruf der Standardwert zugewiesen.

Die Deklaration eines Unteroperators erfolgt im Kopfteil des aufrufenden Operators, wobei eine Operatorvariable definiert wird. Der Name dieser Operatorvariable wird im Programmrumpf beim Aufruf des Operators verwendet. Bei diesem Aufruf müssen die aktuellen Parameter entsprechend den formalen Parametern im Operator übergeben werden. Es ist also möglich, denselben Operator mehrfach mit verschiedenen Parameterbelegungen im selben Experiment, ja sogar im selben Operator zu verwenden.

5.3.3 Bibliothekenkonzept (LEA-Sicht)

In jedem Operator ist es möglich, durch das `USES`-Konstrukt Bibliotheken anzugeben, aus denen Funktionen aufgerufen werden sollen. Diese Funktionen sind in SML verfaßt und haben somit den vollen Zugriff auf alle Konstrukte und Eigenschaften, die SML bietet. Insbesondere sind dies der Zugriff auf Zellen und Atome der Individuen, der aus LEA heraus nicht möglich ist.[1]

Da alle verwendeten Operatoren deklariert werden müssen, handelt es sich bei allen anderen Aufrufen um Aufrufe einer Funktion. Um diese zu finden, wird in den angegebenen Bibliotheken nach dem Funktionsnamen gesucht, wobei in der Reihenfolge vorgegangen wird, in der die Bibliotheken in der `USES`-Anweisung angegeben sind. Befindet sich die Funktion in keiner der angegebenen Bibliotheken, so wird in der Bibliothekenliste des aufrufenden Operators gefahndet usw.

Somit ist es sogar möglich, Operatoren zu schreiben, die je nach Ort ihres Aufrufs verschiedene Kodierungen des Phänotyps akzeptieren: Eine Funktion wird nicht im Operator selbst durch Angabe einer Bibliothek bestimmt, sondern im aufrufenden Operator bzw. im Verfahren.

Daneben bieten die Bibliotheken auch Standard-Funktionen, die der Benutzer vom Laufzeitsystem erwartet, wie z.B. mathematische Funktionen und Ausgabefunktionen. Durch den oben beschriebenen Mechanismus ist es ausreichend, diese Standardbibliotheken einmal im Experimentoperator zu deklarieren.

5.4 Beschreibung der Sprache LEA

Um eine einfachere Eingabe von Experimentdefinitionen, Verfahren und Operatoren zu ermöglichen, wurde die Sprache LEA (Language for Evolutionary Algorithms) entwickelt. Mit ihr können Verfahren und Operatoren in der weiter verbreiteten prozeduralen Form programmiert werden. Zusätzlich ist in einheitlicher Weise auch das Initialisieren und die Steuerung des Ablaufs eines Experiments möglich. Um den Aufwand für die Entwicklung der Sprache nicht zu groß werden zu lassen, wurde auf eine Möglichkeit verzichtet, Phäno- und Genostrukturen in LEA direkt zu verändern. Es kann daher nur bis zur Ebene der Individuen in LEA gearbeitet werden. Für die Elemente, auf denen diese aufbauen, müssen Funktionen in SML geschrieben werden, die über eine spezielle Schnittstelle in LEA verwendet werden können. Andere Konzepte, wie die Möglichkeit Parameter für Verfahren und Operatoren anzugeben, sind direkt in LEA umgesetzt. Dieser Abschnitt soll einen Überblick über die Sprache LEA geben.

[1] Die Implementation dieser Funktionalität hätte den Umfang dieser Projektgruppe gesprengt, jedoch ist dies für eine Erweiterung geplant.

5.4.1 Grundlagen

5.4.1.1 Schlüsselwörter

Die Schlüsselwörter sind neben einfachen und zusammengesetzten Zeichen wie `+`, `:=` etc. die Elemente, aus denen die Sprache LEA besteht. Schlüsselwörter sind Zeichenketten, die mit einem Buchstaben beginnen und aus einer beliebigen Folge aus Buchstaben und Ziffern bestehen. Alle in LEA verwendeten Schlüsselwörter werden im Anhang D aufgelistet.

5.4.1.2 Bezeichner

Auf die in LEA deklarierten Objekte (wie Operatoren, Parameter, lokale Variablen etc.) wird über Bezeichner zugegriffen. Wie die Schlüsselwörter beginnen sie mit einem Buchstaben und bestehen aus einer beliebigen Folge von Buchstaben und Ziffern. Die Buchstaben können dabei groß oder klein geschrieben werden. Bezeichner gelten auch dann als verschieden, wenn sie sich nur in der Groß- und Kleinschreibung unterscheiden. Die Schlüsselwörter dürfen nicht als Bezeichner verwendet werden.

5.4.1.3 Konstanten

In LEA-Programmen können die folgenden Arten von Konstanten verwendet werden:

- Ganzzahlkonstanten sind Folgen von Ziffern, die als ganze Zahl interpretiert werden. Diese Zahl darf allerdings nicht größer werden als die von SML unterstützten 31bit Ganzzahlen. Ein Beispiel für eine Ganzahlkonstante ist `42`.
- Gleitkommakonstanten unterscheiden sich von den Ganzzahlen durch einen enthaltenen Dezimalpunkt. Sie werden als reelle Zahlen interpretiert. Eine LEA-Programm kann z.B. die Gleitkommakonstante `3.1416` enthalten.
- Stringkonstanten sind in doppelte Anführungszeichen eingeschlossene Zeichenketten (z.B. `"Hello World"`). Sie dürfen nicht über das Zeilenende hinaus gehen.

5.4.1.4 Typen

LEA verwendet für die meisten Elemente eine statische Typprüfung. Für alle deklarierten Variablen, Parameter, Operatoren etc. müssen die Typen angegeben werden. Diese Typen werden überprüft, wenn diese Objekte verwendet werden. Konvertierungen müssen explizit vorgenommen werden.

In der ersten Version von LEA werden folgende Typen unterstützt:

- `BOOL` bezeichnet einen bool'schen Typ. Variablen diesen Typs können die Werte wahr und falsch annehmen und werden bei Bedingungen verwendet.
- `INT` bezeichnet einen ganzzahligen Typ. Der Wertebereich dieser Variablen entspricht dem 31bit Ganzzahltyp von SML.
- `REAL` bezeichnet einen reellwertigen Typ.
- `IND`: Variablen diesen Typs können ein Individuum aufnehmen.
- `INDLIST`: Mit diesem Typ können Variablen deklariert werden, die Listen von Individuen aufnehmen.
- `POP`: Variablen von diesem Typ bezeichnen Populationen, die von der Populationsverwaltung des Systems verwaltet werden.

5.4.1.5 Kommentare

LEA unterstützt Kommentare, die mit einem `#` beginnen und sich bis zum Ende der Zeile erstrecken.

```
T := T * faktor;      # erniedrige Temperatur
```

5.4.2 Sprachelemente

5.4.2.1 Experiment

Das oberste Sprachkonstrukt von LEA ist das Experiment. Ein bestimmtes Experiment wird beim Start aufgerufen. Es initialisiert die Populationen und wendet darauf die Verfahren an. Daneben können auch Operatoren für weitere Berechnungen verwendet werden. Ein Experiment besteht aus folgenden Abschnitten:

- `EXPERIMENT <Experimentname>`: Eine Kopfzeile, die den Namen des Experiments angibt.
- Einem Deklarationsteil, der angibt, was im folgenden Anweisungsteil verwendet werden kann. Die Reihenfolge der Deklarationen ist dabei vorgegeben, die optionalen Teile können allerdings weggelassen werden.
  - `USES <Lib1>, <Lib2>, ..., <Libn>` (optional): Eine Liste der verwendeten Funktionsbibliotheken.
  - `PROBLEM <Problemname>`: Die Angabe des Problems, das im Experiment verwendet wird.
  - `POPULATIONS <Populationen>` (optional): Eine Deklaration der im Experiment verwendeten Populationen.
  - `OPERATORS <Liste der Operatoren>` (optional): Die von dem Experiment aufgerufenen Verfahren und Operatoren.
  - `VAR <Variablen>` (optional): Die lokalen Variablen.
- Einem Anweisungsteil, der in `BEGIN` und `END` eingeschlossen ist und aus einer Liste von Anweisungen besteht (siehe unten).

5.4.2.2 Operatoren

Operatoren unterscheiden sich von der Experimentdefinition durch die Möglichkeit, beim Aufruf Parameter zu empfangen und einen Wert zurückzuliefern. Zusätzlich kann jeder Operator eine Reihe von Parametern besitzen, die sein Verhalten steuern und die bei der Verwendung in einem Verfahren (oder Experiment) oder von außen gesetzt werden können. Im Deklarationsteil von Operatoren ist es nicht möglich, ein Problem oder eine Populationen anzugeben. Ein Operator besteht aus den folgenden Teilen:

- `OPERATOR <Opname> (<Typ1> <Par1>, ..., <Typn> <Parn>): <Typ>`: Eine Kopfzeile, die den Namen des Operators, die Parameter, ihre Typen und den Typ des Rückgabewerts angibt.
- Einem Deklarationsteil, der angibt, was im folgenden Anweisungsteil verwendet werden kann. Die Reihenfolge der Deklarationen ist dabei vorgegeben, die optionalen Teile können allerdings weggelassen werden.
  - `USES <Lib1>, <Lib2>, ..., <Libn>` (optional): Eine Liste der verwendeten Funktionsbibliotheken.
  - `PARAMETER <Liste der Parameter>` (optional): Die Parameter, mit denen das Verhalten des Operators gesteuert werden kann.
  - `OPERATORS <Liste der Operatoren>` (optional): Die von diesem Operator aufgerufenen Verfahren und Operatoren.
  - `VAR <Variablen>` (optional): Die lokalen Variablen.
- Einem Anweisungsteil, der in `BEGIN` und `END` eingeschlossen ist und aus einer Liste von Anweisungen besteht (siehe unten).

5.4.2.3 Verfahren

Verfahren sind genauso aufgebaut wie die Operatoren. Die Kopfzeile beginnt mit `ALGORITHM`. Verfahren sollten, anders als Operatoren, die nur in speziellen Fällen auf Populationen arbeiten (z.B. Migrationsoperatoren), eine Population als Parameter übergeben bekommen.

5.4.2.4 Deklarationen

Parameter Parameter für die Verfahren und Operatoren können vom Typ Int, Real und Bool sein. Bei der Deklaration der Parameter wird ein Standardwert angegeben, der verwendet wird, wenn kein Wert angegeben ist. Weiter wird ein Maximal- und ein Minimalwert für den Parameter definiert. Die Werte, die der Parameter annehmen soll, müssen dazwischen liegen. Schließlich folgt noch eine Zeichenkette, die den Parameter beschreibt.

```
<Typ> <Name> = (<Standardwert>,
                <Minimalwert>,
                <Maximalwert>,
                "<Beschreibung>")
```

Operatoren Bei der Deklaration eines Operators wird der Verweis festgelegt, unter dem ein vorhandener Operator im Anweisungsteil angesprochen wird. Dabei werden für die Parameter eines Operators oder Verfahrens konkreten Werte bestimmt. Wenn ein Parameter nicht angegeben ist, so erhält er seinen Standardwert. Die Parameter, für die ein Wert angegeben ist, werden beim Start des Operators auf diesen Wert gesetzt. Auf diese Weise kann der gleiche Operator durch eine Verwendung verschiedener Verweise mit verschiedenen Parameterbelegungen verwendet werden.

```
<Verweis> = <Operatorname>(<Par1>: <Wert1>,
                           <Par2>: <Wert2>,
                           ...,
                           <Parn>: <Wertn>)
```

5.4.2.5 Anweisungen

Anweisungslisten, wie sie als Hauptteil der Operatoren verwendet werden, sind Listen von Anweisungen, die durch Strichpunkte getrennt werden. Die Semantik dieser Anweisungen entspricht im wesentlichen der von PASCAL und MODULA2, an die auch die Syntax angelehnt ist. In LEA werden folgende Anweisungen unterstützt:

- Zuweisung: Weist einer lokalen Variablen oder einem Operatorparameter den Wert zu, der durch den angegebenen Ausdruck bestimmt wird.

  ```
  <Variable> := <Ausdruck>
  ```

- Funktionsaufruf: Auf diese Weise können SML-Funktionen, Operatoren und Verfahren aufgerufen werden. Rekursive Aufrufe sind in LEA allerdings nicht zugelassen. Bei SML-Funktionen kann zur eindeutigen Beschreibung zusätzlich noch der Name der Bibliothek angegeben werden. Wenn Parameter an die Funktion übergeben werden, werden diese auf die übliche Weise nach dem Funktionsnamen als Liste in Klammern angehängt.

  ```
  [<Bibliotheksname>:]<Funktionsname>[(<Parameterliste>)]
  ```

- Verzweigung: Die Verzweigung führt abhängig von einer Bedingung eine von zwei Anweisungslisten aus. Der Ausdruck für die Bedingung muß dabei einen bool'schen Wert ergeben. Wenn dieser wahr ist, wird die erste Anweisungsliste ausgeführt, sonst die zweite.

  ```
  IF <Ausdruck> THEN <Anweisungen>
                ELSE <Anweisungen> FI
  ```

- While-Schleife: Bei der While-Schleife wird eine Anweisungsliste solange ausgeführt, bis die Bedingung falsch ergibt.

  ```
  WHILE <Ausdruck> DO <Anweisungen> OD
  ```

- Repeat-Schleife: Der Anweisungsteil der Repeat-Schleife wird, im gegensatz zur While- Schleife, immer mindestens einmal ausgeführt. Die Schleife wird dann solange wiederholt, bis die Bedingung wahr wird.

  ```
  REPEAT <Anweisungen> UNTIL <Ausdruck>
  ```

- For-To-Schleife: Bei der For-To-Schleife wird vor Beginn der Ausführung die Ganzzahlvariable auf den angegebenen Anfangswert gesetzt. Die Schleife wird dann durchlaufen, wenn die Variable den Endwert noch nicht überschritten hat. Am Ende der Schleife wird die Variable um eins erhöht und wieder zum Anfang der Schleife gesprungen.

  ```
  FOR <Variable> := <Ausdruck> TO <Ausdruck> DO
    <Anweisungen> OD
  ```

- Operator beenden: Die Return-Anweisung beendet einen Operator, ein Verfahren oder das Experiment. Wenn der Operator (oder das Verfahren) einen Rückgabewert zurückliefert, wird dieser hier durch den Ausdruck, der auf das Schlüsselwort folgt, angegeben.

  ```
  RETURN <Ausdruck>
  ```

5.4.2.6 Ausdrücke

Ausdrücke in LEA sind ebenfalls denen in anderen prozeduralen Sprachen nachempfunden. Sie können aus Konstanten, Variablen, Klammerungen, Funktionsaufrufen (wie bei den Anweisungen beschrieben) und den unten aufgelisteten Operatoren aufgebaut werden. Ein Ausdruck wird zu einem Wert von einem der in LEA zulässigen Typen ausgewertet. Ein Beispiel für einen Ausdruck ist:

```
((XR*XR + XI*XI) < 4.0)
```

5.5 Interpreter

Der Interpreter für die Sprache LEA ist der umfangreichste Teil von GENOM. Er nimmt eine zentrale Position ein, da er die Verbindung zwischen Experimenten, Verfahren und Operatoren auf der einen und der Populationsverwaltung auf der anderen Seite darstellt. Der Interpreter zerfällt in mehrere Module, die im folgenden jeweils in einem eigenen Kapitel beschrieben werden. Ein Überblick über die Module findet sich in Kapitel 5.5.1, ein Glossar der verwendeten Fachbegriffe im Kapitel 5.5.8.

5.5.1 Interpreter-Programmteile

Der Interpreter besteht aus folgenden Teilen:

- Preter: Führt eine Anweisungsfolge unter einer Variablenumgebung aus, liefert eine Variablenumgebung zurück.

- Linker: Sorgt für Initialisierung der Umgebungen und startet das Experiment. Führt Aktionen wie Operator- und Funktionsaufrufe durch.
- Parser: Liest eine Datei ein und liefert einen geparsten Operator zurück, der u.a. eine Anweisungsfolge aus Elementar-Befehlen und eine initiale Variablenumgebung enthält.
- Library: Sammlung/Organisation der einzelnen Operatoren. Sorgt für die Ermittlung der benötigten Dateien.
- Frame: Sorgt für das Parsen der abgespeicherten Operatoren und das Einbinden der Bibliotheken durch SML. Startet das Experiment über den Linker.

5.5.2 Preter

Das Modell der Maschine, auf der die Programme des Interpreters ablaufen, ist eine Stackmaschine. Die Berechnung von Ausdrücken wird durch Ablegen der Operatoren auf den Stack und Ausführen einer Operation, die ihre Operanden auf dem Stack erwartet, realisiert.

Elementare Anweisungen Das Maschinenmodell, auf das der Preter aufbaut, verwendet einen lokalen Stapel, der zur Berechnung komplexerer Ausdrücke dient. Somit brauchen Anweisungen oft nur ein Objekt zu kennen, da andere sich auf dem Stack befinden.

Um einen Stack aller möglichen Typen zu bilden, wurden diese zum Datentyp `Varvalue` vereint. In einem Programm können Werte verwendet werden, die entweder konstant sind, eine Variable bezeichnen oder einen Parameter (mit seinem relativen Namen) darstellen. Dies wurde im Datentyp `Value` umgesetzt.

Eine Variable wird durch ihren Namen angesprochen, der als String implementiert ist. Sie kann alle konstanten Werte annehmen.

Der Anweisungsteil eines Programms besteht aus einer Folge von Kommandos, die aus dem Befehlsvorrat entnommen werden. Der Befehlsvorrat wird durch den Datentyp `StackCmd` repräsentiert.

Ein Befehl der Stackmaschine wird durch die Funktion `exec` realisiert. Diese Funktion erhält das auszuführende Kommando, die Nummer des folgenden Befehls, die Variablenumgebung und den aktuellen Stack. Nach Ausführung der Anweisung werden der neue Programmzähler, die neue Variablenumgebung und der neue Stack zurück gegeben.

Soll z.B. eine Konstante auf den Stack gelegt werden (`sc_push`), so wird sie einfach vorne an den Stack angehängt. Soll dagegen das oberste Element des Stacks in eine Variable geschrieben werden (`sc_pop`), so wird die Konstante von Stack entfernt und die Variablenumgebung geändert; d.h. die betreffende Variable auf den gewünschten Wert gesetzt.

Sprünge werden realisiert, indem einfach die berechnete Nummer der nächsten Anweisung zurückgeliefert wird (`sc_jmp`). Eine arithmetische Operation (`sc_op`

```
datatype Varvalue =
           notdeklared
          |notdefinedint
          |notdefinedreal
          |notdefinedbit
          |notdefinedstring
          |notdefinedindi
          |notdefinedindilist
          |notdefinedpop
          |intval of int
          |realval of real
          |bitval of bool
          |stringval of string
          |indival of PopHandler.extInd_type
          |indilistval of PopHandler.extIndL_type
          |popval of PopHandler.PopId_type
          |opval of string * (string * Varvalue) list
          |parval of Varvalue * Varvalue * Varvalue * string
```

Abb. 5.5: Konstante Werte

```
datatype Value =
variabel of string | konst of Varvalue | parref of string list
```

Abb. 5.6: Variable Werte

```
type Variable = string * Varvalue
```

Abb. 5.7: Typ der Variablen

`sc_push(value)`	Legt eine Konstante oder Variable, die mit dem Ausdruck `value` beschrieben wird, auf den Stack.
`sc_pop(bez)`	Speichert das oberste Element auf dem Stack in der Variable mit dem angegebenen Bezeichner (`bez`) in der aktuellen Umgebung und nimmt den Wert vom Stack.
`sc_popparam(parbez)`	Setzt den Wert des Parameters mit der Bezeichnung `parbez` in der aktuellen Umgebung auf den Wert des obersten Stackelements und nimmt diesen vom Stack.
`sc_drop`	Nimmt das oberste Element vom Stack.
`sc_dup`	Legt das oberste Stackelement noch einmal oben auf den Stack.

Tabelle 5.1: Stackverwaltung

`sc_jmp(pos)`	Unbedingter Sprung an die Position `pos` im Stackprogramm.
`sc_jmpT(pos)`	Sprung an die Position `pos` im Stackprogramm, wenn das oberste Element auf dem Stack ein bool'scher Wert und wahr ist.
`sc_jmpF(pos)`	Sprung an `pos`, wenn das oberste Element den bool'schen Wert falsch hat.

Tabelle 5.2: Sprungbefehle

`sc_call_fct(lib, bez, z)`	Ruft die SML-Funktion mit dem Namen `bez` aus der Bibliothek `lib` auf, wobei `z` Elemente als Parameter auf dem Stack liegen. Die Reihenfolge entspricht der in der Prozedurdeklaration, wobei der zuerst deklarierte Parameter auch zuerst auf den Stack gelegt wird. Der letzte Parameter liegt also oben auf dem Stack. SML-Funktionen legen immer einen Wert als Rückgabe auf den Stack, Operatoren nur, wenn ein Rückgabewert in der Deklaration angegeben ist.
`sc_call_op(bez, z)`	Ruft den Operator oder das Verfahren mit dem Namen `bez` auf, wobei `z` Elemente als Parameter auf dem Stack liegen (wie bei `sc_call_fct`).

Tabelle 5.3: Aufrufe von Funktionen

`sc_op(so_add)`	Addiert die beiden obersten Stackelemente und ersetzt sie durch das Ergebnis der Addition.
`sc_op(so_sub)`	Subtrahiert des zweite vom ersten Stackelement und legt das Ergebnis auf den Stack.
`sc_op(so_neg)`	Negiert das oberste Stackelement.
`sc_op(so_mul)`	Multipliziert die beiden obersten Stackelemente und legt das Ergebnis auf den Stack.
`sc_op(so_div)`	Dividiert das zweite vom ersten Stackelement und legt das Ergebnis auf den Stack.
`sc_op(so_eq)`	Vergleicht die beiden obersten Stackelemente auf Gleichheit und legt das Ergebnis des Tests auf den Stack.
`sc_op(so_neq)`	Dasselbe für Ungleichheit.
`sc_op(so_lt)`	Überprüft ob das erste Stackelement kleiner als das zweite ist und legt das bool'sche Ergebnis des Tests auf den Stack.
`sc_op(so_gt)`	Dasselbe mit größer.
`sc_op(so_lte)`	Dasselbe mit kleiner oder gleich.
`sc_op(so_gte)`	Dasselbe mit größer oder gleich.
`sc_op(so_and)`	Berechnet ein logisches Und für die beiden obersten Stackelemente (es müssen bool'sche Werte sein) und legt das Ergebnis auf den Stack.
`sc_op(so_or)`	Dasselbe mit einem logischen Oder.
`sc_op(so_not)`	Invertiert das oberste Stackelement (es muß sich um einen bool'schen Wert handeln).

Tabelle 5.4: Arithmetische Befehle

```
fun exec (sc_push(konst(v)), pc, vars, stack) =
       (pc, vars, v::stack)
| exec (sc_pop(s), pc, vars, v::stack) =
       (pc, setvar(s,vars,v), stack)
| exec (sc_jmp(new_pc), _, vars, stack) =
       (new_pc, vars, stack)
| exec (sc_op(operation), pc, vars, stack) =
       (pc, vars, stack_op(operation, stack))
| ...
```

Abb. 5.8: Ausführung von Stackbefehlen

```
fun stack_op (so_add, realval(a)::realval(b)::tl) =
       realval(b+a)::tl
| stack_op (so_neg, intval(a)::tl) = intval(~a)::tl
| stack_op (so_eq, realval(a)::realval(b)::tl) =
       bitval(b=a)::tl
| ...
```

Abb. 5.9: Ausführung arithmetischer Befehle

(`opbez`)) wird durch eine eigene Funktion (`stack_op`) umgesetzt, die neben dem Operationsbezeichner (`opbez`) nur den Stack erhält und den neuen Stack liefert.

Eine arithmetische Operation nimmt ihre(n) Operanden vom Stack, führt die geforderte Aktion aus und legt das Ergebnis in der geforderten Form wieder auf den Stack (`stack_op`).

Die Übergabe von aktuellen Parametern erfolgt durch das Ablegen der Werte auf dem Stack, wobei die Anzahl der Parameter im „Call"-Befehl gespeichert ist. Der Rückgabewert eines Operators ist zu oberst auf dem Stack zu finden.

Die Elementar-Anweisungen werden zu den Operatoren in einem Array gespeichert, wobei bei der Ausführung über einen Index auf den aktuellen Befehl zugegriffen wird. Sprünge sind somit möglich, außerdem kann so leicht die Position einer Unterbrechung zurückgegeben werden — wodurch Wiederaufsetzen ermöglicht wird.

Die Preter-Funktion `run` nimmt vom Linker die aktuelle Variablenumgebung, den Anweisungsteil des auszuführenden Operators, den aktuellen Stack und die Position der Anweisung, mit der die Ausführung beginnen soll.

```
val run:
          (Types.Variable list *          (* Environment at Start *)
          Commands.StackProg *            (* Statements *)
          Stack *                         (* Stack at Start *)
          Commands.InstrPos               (* Start here *)
          ) -> Exit_state
```

Abb. 5.10: Funktion Preter.run

Die Ausführung eines Operators kann zu verschiedenen Endzuständen führen (s. `Exit_state`):

- `done`: Der Operator wurde durch ein `return` normal beendet.
- `call_op`: Es soll ein anderer Operator ausgeführt werden. Deshalb werden zusätzliche Informationen zurückgeliefert, die das Fortsetzen der Ausführung nach Beendigung des nachgeladenen Operators erlauben.
- `call_fct`: Es soll eine Benutzer-Funktion aufgerufen werden. Der Ablauf ist mit einem Operator-Aufruf vergleichbar.
- `break`: Es wurde eine Unterbrechungsanweisung gefunden! Die Kontrolle soll nun an die Systemumgebung übergeben werden, die ggf. eine Veränderung des Experiments durch den Benutzer ermöglicht.
- `trace`: Eine Kontroll-Anweisung wurde aktiv, weil eine Bedingung eingetreten ist. Das Verhalten ist das selbe wie bei `break`.

5.5.3 Linker

Der Linker bereitet den Start eines Experiments vor: In die initiale Variablenumgebung wird für die Parameter der Operatoren der Default-Wert eingetragen, dies wird rekursiv für alle Variablenumgebungen von der Funktion `init_openv` durchgeführt. Sie ermittelt jeweils die Unter-Operatoren eines Operators durch eine Anfrage an die Library und ruft sich für diese noch einmal auf. Die theoretische Möglichkeit einer Endlosschleife (d.h. ein Operator steht mehrmals in einem Pfad des Operatorbaums) wird durch Konvention ausgeschlossen.

Somit entsteht für das Experiment eine Variablenumgebung, die für jedes Verfahren eine eigene Umgebung enthält; in diesen für jeden Operator eine weitere usw. In den initialen Variablenumgebungen werden aber nur die Daten für Unteroperatoren und Parameter abgelegt, da diese auch außerhalb der Ausführungzeit eines Operators oder Verfahrens existieren müssen. Es wäre somit möglich, z.B. im Experimentoperator durch Setzen eines Parameters eines „tief unten“ im Operatorbaum liegenden Operators, diesen in seiner Funktion zu beeinflussen oder auch nach einer Programmunterbrechung diese Parameter zu verändern. (Dies wurde im vorliegenden System nicht implementiert.)

Der Linker erzeugt außerdem die im Experimentoperator festgelegten Populationen, wobei die angegebenen Kodierungen verwandt werden. Dann wird der Anweisungsteil des Experimentoperators gestartet, dessen Beendigung das Ende der evolutionären Berechnung darstellt.

Steht bei der Ausführung eines Operators ein „Call“-Befehl zur Ausführung an, so wird die Ausführung der Anweisungen im Preter unterbrochen und die augenblickliche Umgebung und die Position der Anweisung zurückgegeben.

Soll ein Operator ausgeführt werden, so sucht ihn der Linker in der Library und erzeugt seine Variablenumgebung, die aus vier Teilen besteht:

- Parameter: Diese Variablen können in hierarchisch höher liegenden Operatoren verändert werden, daher müssen sie in der Variablenumgebung des Aufrufers existieren, was beim Start eines Experiments durchgeführt wurde.

- lokale Variablen: Auf diese Variablen kann nur vom Operator selbst zugegriffen werden. Sie werden beim Parser mit einer Vorbelegung versehen, die auch „nicht-definiert“ sein kann.

- aktuelle Parameter: Diese Parameter werden beim Aufruf des Operators übergeben und liegen dann als belegte lokale Variablen vor. Die Übergabe erfolgt über den Stack, der Linker nimmt die Parameter ab und weist sie den formalen Parametern zu.

- Bezeichner der auf diesem Pfad im Operatorbaum benutzten Bibliotheken. (Somit kann derselbe Funktionsaufruf im selben Operator verschiedene Funktionen anstoßen. Es wird die zuletzt deklarierte Bibliothek verwandt!)

Wird dagegen eine Funktion aufgerufen, so werden die zu übergebenden Parameter vom Stapel genommen und die Bibliothek bestimmt, deren Dispatcher-Funktion den eigentlichen Funktionsaufruf und die Umwandlung der Parameter durchführt. Das Ergebnis des Aufrufs wird auf den Stapel gelegt.

Die Bestimmung der Bibliothek, deren Funktion ausgeführt wird, geschieht über den Bibliotheken-Pfad. Der Bibliotheken-Pfad wird durch den Operatorbaum definiert, d.h. die Struktur der Operatoraufrufe eines Experiments. In der Wurzel steht der Experimentoperator, direkt darunter die Verfahren, unter diesen die Operatoren. In jedem Operator bzw. Verfahren kann man durch das `USES`-Konstrukt von `LEA` Bibliotheken angeben. Diese Namen werden beim Abstieg in den Baum vorne an den Bibliotheken-Pfad angehängt, beim Funktions-Aufruf wird in den angegebenen Bibliotheken nach dem Funktionsnamen gesucht. Da diese Suche vorne beginnt, kann ein Operator festlegen, aus welcher Bibliothek eine Funktion entnommen werden soll, er kann aber auch keine Angabe machen. Dann wird die Funktion in den Bibliotheken gesucht, die oberhalb des Operators benannt wurden. Somit kann derselbe Funktionsaufruf im selben Operator je nach Operatorbaum verschiedene Bedeutungen haben.

Beispiel Im Operatorbaum steht an oberster Stelle das Experiment `Example`. Es ruft die Verfahren `Threshold, Other_Threshold` und `Genetic` auf. Man beachte, daß sowohl in `Threshold` als auch in `Genetic` der Operator `mutate` Verwendung findet. Jedoch unterscheiden sich die Bibliotheken-Pfade für diesen Operator, je nachdem wo er sich im Baum befindet. Somit kann ein Funktionsaufruf in `mutate` einmal in der Bibliothek `RealLib`, ein anderes Mal in `BinaryLib` ausgeführt werden.

Als Gegenbeispiel sei der Operator `other_mutate` angegeben, der die Bibliothek `MutateLib` angibt. Somit werden Aufrufe wohl von dieser ausgeführt, egal wo der Operator im Baum steht.

Nach einem Aufruf wird die Bearbeitung des Operators an der dem Aufruf folgenden Anweisung fortgesetzt.

<table>
<tr><td>Experiment "Example"</td><td colspan="2">Libraries: "General", "PopHandler"
Populations: "Binaries", "Reals"
Lib.Path: "General", "PopHandler"</td></tr>
<tr><td>Algorithm "Threshold"

Libraries: "Reallib"
Lib.Path: "Reallib", "General", "PopHandler"</td><td>Algorithm "Other_Threshold"

Libraries: "Reallib"
Lib.Path: "Reallib", "General", "PopHandler"</td><td>Algorithm "Genetic"

Libraries: "BinaryLib"
Lib.Path: "BinaryLib", "General", "PopHandler"</td></tr>
<tr><td>Operator "mutate"

Libraries:
Lib.Path: "Reallib", "General", "PopHandler"</td><td>Operator "other_mutate"

Libraries: "MutateLib"
Lib.Path: "MutateLib", "Reallib", "General", "PopHandler"</td><td>Operator "mutate"

Libraries:
Lib.Path: "BinaryLib", "General", "PopHandler"</td></tr>
</table>

Abb. 5.11: Operatorbaum

5.5.4 Parser

Der Parser wandelt die Textdateien, in denen die Experimente, Verfahren und Operatoren in der Sprache LEA gespeichert sind, in die vom Preter und Linker benötigten SML-Datenstrukturen um. Aus der Beschreibung eines Operators in LEA werden für den Linker Listen mit den deklarierten Variablen erzeugt, die dieser verwendet, um die initiale Variablenumgebung vor dem Ausführen des Operators bereitzustellen. Der Anweisungsteil des Operators wird in eine Folge von Stackbefehlen umgewandelt, die dem Preter zur Ausführung übergeben wird.

5.5.4.1 Aufbau

Der Parser verwendet intern die folgenden Module:

- `stackcommands.sml` Die Definition der Befehle für die Stackmachine.
- `parsertypes.sml` Definiert die Typen, die von der Schnittstelle zwischen Parser und Preter/Library verwendet werden.
- `errors.sml` Enthält die Fehlermeldungen für den Scan- und den Parsevorgang.
- `scanner.sml` Wandelt eine Textdatei in eine Folge von lexikalischen Symbolen um. Enthält die Definition der lexikalischen Symbole von LEA.
- `environ.sml` Verwaltet die Listen der lokalen Deklarationen eines Operators.
- `parser.sml` Der eigentliche Parser mit der syntaktischen und semantischen Analyse der lexikalischen Symbole. Er erzeugt eine Beschreibung des Deklarationsteil und eine Liste mit Stackbefehlen, die dem Anweisungsteil des geparsten LEA-Programms entspricht.

Nach außen sind dabei nur die Module `parser.sml`, mit den eigentlichen Parserfunktionen und das Modul `stackcommands.sml` mit der Definition für die Befehle der Stackmachine sichtbar.

5.5.4.2 Verwendung

Das Modul Parser exportiert zwei Funktionen `parsfile1` und `parsfile2`. Diese werden nacheinander für eine Textdatei mit einem Experiment, Operator oder Verfahren in der Sprache LEA aufgerufen. Die erste Funktion parst nur den Deklarationsteil dieses Konstrukts und übergibt eine Beschreibung der dort deklarierten Elemente. In diesem Durchlauf können einige Prüfungen noch nicht durchgeführt werden (z.B. ob die verwendeten Unteroperatoren vorhanden sind). Diese werden erst möglich, wenn der erste Durchlauf für alle in dem Experiment (und den darin aufgerufenen Verfahren und Operatoren) verwendeten Operatoren durchlaufen wurde. Diese Prüfungen werden dann im zweiten Durchlauf durch die Funktion `parsfile2` durchgeführt in dem neben dem Deklarationsteil auch der Anweisungsteil geparst und Code dafür erzeugt wird.

5.5.4.3 Schnittstellen

Entsprechend der dort erlaubten Deklarationen sind auch die zurückgegebenen Datenstrukturen für Experimente und Verfahren bzw. Operatoren unterschiedlich.

Bei allen Operatoren, Verfahren und Experimenten werden die folgenden Elemente zurckgegeben:

- Der Name des Operators,
- eine Liste mit den verwendeten Bibliotheken (deklariert mit `USE`),
- eine Liste der verwendeten Operatoren (deklariert mit `OPERATORS`),
- eine Liste mit den lokalen Variablen (deklariert mit `VAR`),
- und eine Liste der Stackbefehlen, die dem Anweisungsteil entsprechen.

Bei einem Experiment werden zusätzlich noch die Elemente zurückgegeben, die nur dort deklariert werden dürfen:

- Der Name des verwendeten Problems
- und eine Liste mit den in diesem Experiment deklarierten Populationen (mit `POPULATIONS`).

In Verfahren und Operatoren sind die gleichen Deklarationen erlaubt. Hier gibt es noch folgende Elemente:

- Ob es sich um einen Operator oder ein Verfahren handelt, entsprechend der Kopfzeile (`OPERATOR` oder `ALGORITHM`),

- eine Liste der Parameter, die für diesen Operator gelten (deklariert mit `PARAMETER`),
- eine Liste der formalen Parameter, die in der Kopfzeile deklariert werden
- und der Typ für den Rückgabewert, ebenfalls in der Kopfzeile deklariert.

Typen und Werte werden mit der Bibliothek (Library) und dem Rahmen für den Interpreter (Frame) mit der Datenstruktur `Varvalue` ausgetauscht. Sie dient dazu die verschiedenen Typen, die im Interpreter verwendet werden, auf einen einzelnen Datentyp abzubilden. Auch alle Konstanten und Werte, die der Parser verwendet sind mit diesem Datentyp definiert. Mit der Datenstruktur `Variable` kann dazu noch der Name des Elements angegeben werden.

Die Listen von Variablen, formalen Parametern u.ä., die an das Rahmenprogramm für den Interpreter zurückgegeben werden, werden als Listen von Elementen dieses Datentyps übergeben. Dies hat den Vorteil, daß damit gleichzeitig der Name und der Typ der Variable sowie eine eventuelle Vorbelegung, wie sie bei lokalen Variablen möglich ist, übergeben werden kann. Auch Operatoren werden mit dieser Datenstruktur übergeben, wobei das zweite Element des Argumenttupels die Beschreibung für die in der Deklaration vorbelegten Parameter enthält.

Für die Beschreibung der Typen wird im Parser die Datenstruktur `Types` verwendet. Mit ihr können alle von LEA erlaubten Typen beschrieben werden:

```
datatype Types =
    tp_int
  | tp_real
  | tp_bool
  | tp_string
  | tp_ind
  | tp_indlist
  | tp_pop
  | tp_unknown
  | tp_notype
```

Der Typ `tp_unknown` wird im Parser (und in der Bibliothek) dazu verwendet, einen beliebigen Typ zu beschreiben. Z.B. beschreibt er bei Prüfung auf Typkorrektheit einen Ausdruck, dessen Typ noch nicht feststeht. Der Typ `tp_notype` wird verwendet, wenn an der Stelle kein Typ gültig ist. Z.B. bei einer Funktion, die keinen Rückgabewert hat. Zur Umrechnung der Datenstruktur `Varvalue` in die Datenstruktur `Types` gibt es die Funktion `typeofvalue`. Umgekehrt die Funktion `valuefromtype`.

Für den Problemnamen und die Kodierungen wird ein String übergeben, der die Bezeichnung in der Bibliothek enthält (dieser emtspricht dem Dateinamen ohne der Endung).

Die Übergabe von Parametern geschieht mit:

```
type ParameterDecl = Variable * Varvalue * Varvalue * string
```

Das erste Element des Tupels bestimmt den Namen, den Typ und den Standardwert des Parameters. Das zweite und dritte Element den Minimal- und den Maximalwert. Das letzte Element enthält schließlich die Beschreibung, die in der Textdatei für diesen Parameter angegeben ist.

Bei der Übergabe der Populationsdeklarationen muß übergeben werden, welche Kodierung für die Population verwendet wird, in welche Log-Datei geschrieben werden soll und wie die Population initialisiert wird:

```
datatype PopInit = loadfromfile of string |
                   randompop of int |
                   standardpop

(* declaration of populations *)
type PopDecl = string *
               string *
               string *
               PopInit
```

Die Stackbefehle, die für den Anweisungsteil erstellt werden, werden im Abschnitt Codegenerierung beschrieben.

5.5.4.4 Der Scanner

Der Scanner wandelt eine eingelesene Textdatei in eine Folge von Symbolen (den Token) um. Diese Token stehen für die Grundelemente und Schlüsselwörter von LEA und werden in der Datenstruktur `Token` definiert. Abgesehen von einigen Funktionen zum Arbeiten mit den Token besteht der Scanner nur aus den Funktionen `initscanner` und der Funktion `gettoken`. Die Funktion `initscanner` hat dabei nur die Funktion, den Scanner in den Anfangszustand zu versetzen.

Die Funktion `gettoken` liest solange Zeichen aus einer Textdatei mit einem LEA-Programm, bis ein Token erkannt wurde und gibt dieses mit seiner Anfangsposition zurück. Erkannt werden Ganzzahl-, Gleitkomma- und Stringkonstanten sowie alle Zeichen (wie z.B. `+` oder `:=`) und Schlüsselwörter (wie `BEGIN`), die in LEA verwendet werden. Andere Zeichenketten, die mit einem Buchstaben beginnen und aus einer Folge von Buchstaben und Zahlen, bestehen werden als Bezeichner behandelt. Eine Überprüfung, ob diese gültig sind wird erst bei der semantischen Analyse vorgenommen. Kommentare werden vom Scanner überlesen.

Schlüsselwörter erkennt der Scanner anhand einer Tabelle (`kw_list`). Diese enthält die Zeichenkette für ein Schlüsselwort und das Token, das erzeugt wird, wenn diese Zeichenkette gefunden wurde.

5.5.4.5 Der eigentliche Parser

Der Parser überprüft die Folge von Tokens auf ihre syntaktische Übereinstimmung mit der Sprache LEA überprüft und erstellt einen Syntaxbaum (wenn auch nur durch die Abfolge der Funktionsaufrufe). Da es sich trotz der oben

beschriebenen Aufteilung in zwei Durchläufe im wesentlichen um einen Ein-Pass-Compiler handelt, wird durch den Parser auch die Typprüfung und die Generierung des Codes durchgeführt.

Arbeitsweise Bei dem Parser handelt es sich um einen Recursive-Descent-Parser, d.h. er bildet die Struktur der EBNF mit Funktionen für jedes Nichtterminalzeichen nach, die sich entsprechend dem Syntaxbaum des Programms rekursiv aufrufen. Allerdings weicht die konkrete Realisierung in einigen Punkten von der EBNF ab, wie sie im Anhang dargestellt ist.

Beim Auftreten eines Fehlers wird der Parsevorgang sofort abgebrochen und die Nummer des Fehlers und seine Position (Zeile und Spalte in der Textdatei) zurückgegeben.

Der Parser durchläuft zuerst den Deklarationsteil des zu parsenden Operators (im weiteren sind damit meist auch die Verfahren oder Experimente gemeint) und erstellt aus den Deklarationen die lokale Umgebung für diesen Operator. Für die Blöcke des Deklarationsteils gibt es dazu jeweils eine Funktion (z.B. `parseparams`), in der wiederum die Unterfunktionen für die darin enthaltenen Elemente definiert sind. Die lokale Umgebung ist durch die folgende Datenstruktur realisiert, die im Modul `DefinedVars` in der Datei `environ.sml` definiert ist:

```
type definedvars =
  (ParameterDecl list *  (* parameters *)
   Variable list *       (* operators *)
   Variable list *       (* local variables *)
   Variable list *       (* formal parameters *)
   Types *               (* result *)
   PopDecl list *        (* populations *)
   string list)          (* used libs *)
```

Das Modul enthält auch die Funktionen, mit denen auf diese Datenstruktur zugegriffen wird. Die lokale Umgebung wird beim Parsen des Anweisungsteils dazu verwendet, zu überprüfen, ob ein Bezeichner deklariert ist (mit der Funktion `isvardefined`), was für eine Art von Element er bezeichnet (z.B. mit der Funktion `isinlocvars`, die überprüft, ob es sich um einen Bezeichner für eine lokale Variable handelt) und was für einen Typ er hat (mit der Funktion `getidenttype`).

Als Beispiel für den gesamten Parser wird hier kurz das Vorgehen beim Parsen eines Ausdrucks im Anweisungsteil dargestellt. Zu den Ausdrücken gehört der folgende Teil der EBNF:

expr = logexpr [(„AND“ | „OR“) expr] .

logexpr = algexpr
[(„=“ | „<>“ | „<“ | „>“ | „<=“ | „>=“) algexpr] .

algexpr = term [(„+“ | „-“) algexpr].

term = factor [(„*“ | „/“) term] .

factor = combident [„(“ expr { „,“ expr } „)“] |
constant |
„(“ expr „)“ |
„-“ factor |
„NOT“ factor .

Da SML keine wechselseitig rekursiven Aufrufe zuläßt und die Ausdrücke selber wieder beliebige Ausdrücke enthalten können (z.B. in Klammern), gibt es in der Realisierung in SML für alle Nichtterminale der Ausdrücke eine Funktion `parseexpr`, der das betreffende Nichtterminalzeichen mitgegeben wird.

```
datatype parseexprmode =
  expr | logexpr | algexpr | term | factor | combident
```

Die Funktion `parseexpr` wird aufgerufen, wenn in der Folge der Token als nächstes ein Ausdruck erwartet wird. Ihr wird die bisher errechnete Liste von Stackbefehlen übergeben, das erste Token und die Position des Anfangs des zu parsenden Programmstücks, der Typ, den der Ausdruck haben soll (oder `tp_unknown`, wenn dieser noch nicht bekannt ist) und das erwartete Nichtterminalzeichen (siehe oben). Da einige Teile dieser Funktion recht umfangreich sind, ist hier nur der Teil für `algexpr` vollständig aufgeführt.

```
fun parseexpr(cmdlist, (tok, pos), exprtype, combident) =
      ...
  | parseexpr(cmdlist, (tok, pos), exprtype, factor) =
      ...
  | parseexpr(cmdlist, (tok, pos), exprtype, term) =
      ...
  | parseexpr(cmdlist, (tok, pos), exprtype, algexpr) =
      let
        (* parse a term *)
        val (newcmdlist1, (tok1, pos1), rtntype) =
              parseexpr(cmdlist, (tok, pos), exprtype, term)

      (* main part of parseexpr(algexpr) *)
      in
        (* looking for a plus or minus symbol *)
        if (tok1 = tk_opsymbol(os_plus) orelse
            tok1 = tk_opsymbol(os_minus))
        then
          (* parse another algebraic expression with the *)
          (* same type as above and put a op-command in  *)
          (* the cmdlist                                 *)
          setcmdinret(
            parseexpr(newcmdlist1,
                      gettoken(instr),
```

```
                    rtntype,
                    algexpr),
            cmdforoptoken(tok1))
        else
          (newcmdlist1, (tok1, pos1), rtntype)
      end
  | parseexpr(cmdlist, (tok, pos), exprtype, logexpr) =
      ...
  | parseexpr(cmdlist, (tok, pos), exprtype, expr) =
      ...
```

Für `algexpr` wird zuerst ein Teilausdruck geparst (`term`), der nur stärker bindende Operatoren oder geklammerte Ausdrücke enthält (z.B. aber auch nur eine Konstante). Dabei wird der Code für diesen Ausdruck erzeugt und sein Typ ermittelt. Wenn das nächste Symbol ein Plus- oder ein Minuszeichen ist, wird die Funktion ein weiteres Mal für das nächste Token im Programmtext aufgerufen. Dann wird auch überprüft, ob die beiden Typen identisch sind (noch wird nicht überprüft, ob der angegebene Operator auf diesen Typ angewendet werden kann) und schließlich ein Stackbefehl erzeugt, der die Werte der beiden Teilausdrücke aufaddiert oder voneinander subtrahiert.

Von der Funktion `parsexpr` wird schließlich die neue Liste mit den Anweisungen, das nächste Token und seine Position sowie der ermittelte Typ des Ausdrucks zurückgegeben.

Die Funktion `parsestatements`, die für die Nichtterminale der Anweisungslisten zuständig ist, ist ähnlich aufgebaut.

Typprüfung Der Parser führt für die Ausdrücke eine Typprüfung durch (zur Zeit wird nur überprüft, ob die zwei Typen der Teilausdrücke gleich sind, auf die ein Operator angewendet). Für jedes Nichtterminalzeichen des Ausdrucks wird dazu sein Typ ermittelt und mit einem erwarteten verglichen.

Die Überprüfung dieses Typs entspricht einer L-Attributierung. Jeder Funktion wird mitgegeben, welcher Typ für das entsprechende Nichtterminalzeichen erwartet wird (siehe obiges Beispiel). Dieser Typ wurde beim Parsen des vorhergehenden Teils des Ausdrucks berechnet. Es handelt sich also um ein ererbtes Attribut. Wenn noch kein Typ ermittelt wurde, wird der Typ `tp_unknown` gesetzt. Der ermittelte Typ des Nichtterminalzeichens wird aus den Typen der Teilausdrücke errechnet. Hier handelt es sich um ein zusammengesetztes Attribut. Schließlich werden die beiden Typen verglichen und, wenn sie nicht übereinstimmen, eine Fehlermeldung erzeugt.

Die Typen für die verwendeten SML-Funktionen und Operatoren werden von der Bibliothek mit der Funktion `parameteroffkt` erfragt, die einen Typ für den Rückgabewert (oder `tp_notype`, wenn die Funktion keinen hat) und eine Liste von Typen für die Parameter der Funktion zurückliefert.

Codegenerierung Der Parser erzeugt für den Anweisungsteil der Operatoren Code, der auf einer im Preter simulierten Stackmachine ausgeführt wird. Die

verwendeten Befehle sind beim Preter beschrieben. Ein Stackprogramm ist eine Folge aus diesen Befehlen. Einzelne Befehle können über ihre Position in diesem Programm angesprungen werden.

Codegenerierung für Ausdrücke Für die Ausdrücke wird eine Folge von Befehlen erzeugt, nach deren Ausführung der Wert dieses Ausdrucks auf dem Stack liegt. Auch wenn während der Berechnung Zwischenergebnisse auf den Stack gespeichert werden, liegt am Ende nur der Wert des Ausdrucks auf dem ursprünglichen Stack.

Dazu wird für jede vom Parser gefundene Regel der entsprechende Befehl erzeugt, entsprechend den rekursiven Aufrufen der Parser-Funktionen für diese Regeln.

Für Konstanten, Variablen und Funktions- oder Operatoraufrufe wird ein Befehl eingefügt, der den entsprechenden Wert auf den Stack legt. Konstante Werte werden direkt auf den Stack gelegt (`sc_push`), Variablen ausgelesen und ihr Wert dann auf den Stack gelegt (ebenfalls mit `sc_push`). Bei einem Aufruf einer Funktion (oder eines Operators oder Verfahrens) wird zuerst der Code für die Ausdrücke erzeugt, deren Werte der Funktion als Parameter übergeben werden sollen. Da der Code in der Reihenfolge ihres Auftretens in der Parameterliste erzeugt wird, befinden sich die Parameterwerte in der richtigen Reihenfolge für die Befehle `sc_call_fct` or `sc_call_op` auf dem Stack (d.h. der letzte Parameter oben auf dem Stack). Für den eigentlichen Prozeduraufruf wird ein Befehl erzeugt, der die Parameter vom Stack nimmt, diese aufruft und das Ergebnis wieder auf den Stack legt. Bei einem Operator `sc_call_op` oder `sc_call_fct` für eine SML-Funktion.

Für zweistellige Operationen werden zuerst die beiden Teilausdrücke geparst und dabei der Code generiert, der den Wert dieser Ausdrücke berechnet. Dann wird ein Befehl erzeugt, der die gewünschte Operation auf die beiden Werte, die jetzt oben auf dem Stack liegen, ausführt. Für ein + wird z.B. der Befehle `sc_op(so_add)` erzeugt. Bei einstelligen Operationen funktioniert es genauso mit einem Teilausdruck.

Beispiel Für den Ausdruck $4 + (5 + 2) * 3$ wird das folgende Programmteil generiert:

```
sc_push(intval(4))
sc_push(intval(5))
sc_push(intval(2))
sc_op(so_add)
sc_push(intval(3))
sc_op(so_mul)
sc_op(so_add)
```

Codegenerierung für Anweisungen Der Code für einen LEA-Operator setzt sich zusammen aus dem Code, der für die einzelnen Anweisungen erzeugt wird, aus denen er besteht.

Zuweisung: Für eine Zuweisung wird zuerst der Code für den Ausdruck erzeugt (siehe oben), dessen Wert der Variable zugewiesen werden soll. Dieser Wert wird dann mit dem Befehl `sc_pop` in die Variable geschrieben.

Funktionsaufruf: Ein Funktionsaufruf als Anweisung funktioniert im wesentlichen wie ein Funktionsaufruf innerhalb eines Ausdrucks (siehe oben). Es muß allerdings der Rückgabewert, wenn nötig, wieder vom Stack genommen werden. Dazu wird der Befehl `sc_drop` hinzugefügt. Bei Operatoren ist dies nur nötig, wenn diese einen Wert zurckgeben. Da beim Aufruf von SML-Funktionen grundsätzlich ein Rückgabewert auf den Stack gelegt wird, muß danach immer `sc_drop` stehen.

If-Anweisung: Für eine If-Anweisung werden folgende Befehle erzeugt. Bei der Codeerzeugung wird unterschieden, ob die If-Anweisung einen Else-Zweig hat oder nicht.

If-Anweisung ohne Else-Zweig:

```
     <Code fuer Bedingung>
     sc_jmpF 1
     <Code fuer Anweisungsliste im Then-Zweig>
1:   ...
```

If-Anweisung mit Else-Zweig:

```
     <Code fuer Bedingung>
     sc_jmpF 1
     <Code fuer Anweisungsliste im Then-Zweig>
     sc_jmp 2
1:   <Code fuer Anweisungsliste im Else-Zweig>
2:   ...
```

Die Marken 1 und 2 stehen dabei für konkrete Positionen in der Liste der Anweisungen. Bei der Erzeugung der Sprunganweisungen sind diese Positionen noch nicht bekannt. Nach der Erzeugung des Codes für die Anweisungslisten sind diese Positionen bekannt und müssen in den Sprungbefehlen, deren Position sich der Parser gemerkt hat, nachgetragen werden (lookup).

While-Schleife: Der Code für die While-Schleife ist ähnlich wie der für die If-Anweisung, nur gibt es hier noch einen Befehl am Ende der Schleife, der einen Sprung zurück an den Anfang der Liste ausführt.

```
1:   <Code fuer Bedingung>
     sc_jmpF 2
     <Code fuer Anweisungsliste im Rumpf>
     sc_jmp 1
2:   ...
```

Repeat-Schleife: Entsprechend ist der Code für die Repeat-Schleife, nur daß hier die Bedingung hinter dem Schleifenrumpf steht und der erste Sprungbefehl entfällt.

```
1: <Code fuer Anweisungsliste im Rumpf>
   <Code fuer Bedingung>
   sc_jmpF 1
   ...
```

For-Schleife: Bei der For-Schleife muß die Bedingung selbst generiert werden. Auch muß im Schleifenrumpf die Zählvariable hochgezählt werden.

```
   <Code fuer Anfangswert>
   sc_pop Zaehlvariable
1: <Code fuer Endwert>
   sc_push <Zaehlvariable>
   sc_op(so_gte)
   sc_jmpF 2
   <Code fuer Anweisungsliste im Rumpf>
   sc_push <Zaehlvariable>
   sc_push 1
   sc_op(so_add)
   sc_pop <Zaehlvariable>
   sc_jmp 1
2: ...
```

Return-Anweisung: Bei der Return-Anweisung wird zuerst der Rückgabewert auf den Stack gelegt, wenn einer angegeben ist. Dann wird noch der Befehl für das Beenden des Operators erzeugt.

```
   <Code fuer Rueckgabewert>
   sc_return
```

5.5.4.6 Verbesserungen

Folgende Möglichkeiten zur Verbesserung des Parsers sollen hier noch kurz erwähnt werden.

Bei der Erzeugung von Fehlermeldungen müßten zusätzlich die FOLLOW-Mengen einiger Produktionen beachtet werden. Z.B. werden beim Auftreten eines Fehlers in einem der Deklarationsblöcke alle folgenden als leer angenommen und der Fehler erst erkannt, wenn das `BEGIN` des Hauptteils erwartet wird. Solche Fehler können wesentlich besser behandelt werden, wenn schon beim ersten Deklarationsblock erkannt wird, daß das gefundene Token nicht in der entsprechenden FOLLOW-Menge enthalten ist.

Der Parser ist eigentlich als Ein-Pass-Compiler geplant. Da das Aufbauen des Operatorbaums zwei Durchläufe des Parsers nötig macht, wäre es sinnvoll, beim ersten Durchlauf einen Strukturbaum des Operators zu erzeugen, aus dem im zweiten Durchlauf der Code erzeugt wird. Damit könnte der Parser in zwei wesentlich übersichtlichere Funktionen aufgeteilt werden. Ein Strukturbaum würde außerdem bessere Möglichkeiten zur semantischen Analyse bieten.

5.5.5 Bibliothek – *Library*

In der Library werden Funktionen und die geparsten Operatoren bereitgehalten. Operatoren werden in der Textform geladen (durch einen Aufruf an den Parser) und liegen dann als SML-Code in einer Struktur vor. Die Library liefert nach Übergabe eines Bezeichners an die Funktion `find_op` den dazugehörenden Operator an den Aufrufer, d.h. i.d.R. den Linker.

Operatoren werden über einen Namen (einen String) identifiziert. Heißen mehrere Operatoren gleich, so wird der zuletzt geparste verwendet.

Die Library verwaltet auch die Funktionen des Systems. Funktionen werden direkt in SML geschrieben und können selbst zwar andere Funktionen aufrufen, nicht aber Operatoren. Im System gibt es „interne Funktionen“ und „Benutzer-Funktionen“, wobei erstere vom System bereitgestellt werden und fundamentale Operationen aus führen (z.B. Ermittlung einer Zufallszahl). Benutzer-Funktionen dagegen werden vom Benutzer in Bibliotheken zusammengefaßt und sind zur Durchführung bestimmter evolutionärer Berechnungen vorhanden.

Während interne Funktionen beim Start des Systems bereits bereitstehen, müssen Bibliotheken mit Benutzer-Funktionen erst nachgeladen werden, sofern sie vom Experiment benötigt werden (s. Frame). Danach stehen diese Funktionen mit Verwaltungsdaten zur Verfügung, jeweils nach Bibliotheken gruppiert. Hierdurch können verschiedene Bibliotheken Funktionen gleichen Namens enthalten.

Bibliotheken lassen sich bei der Library registrieren, indem sie die Funktion `add_disp` aufrufen, die die Dispatcher-Funktionen der Bibliothek einträgt.

Um einen neuen Operator in die Library einzutragen, ruft der Parser die Funktion `add_opdata` auf, die neben dem Namen auch die Daten erhält. Der Namen, ein String, dient im folgenden auch zur Referenzierung des Operators.

Ebenso lassen sich die von Frame nachgeladenen Kodierungen registrieren, indem sie die Funktion `add_coding` aufrufen.

5.5.6 Frame

Der eigentliche Start eines Experiments erfolgt durch Aufruf der Funktion `start "Experimentname";` aus der Datei `start.sml`. Diese Funktion ist auf oberster SML-Ebene definiert, so daß das Nachladen von Programmteilen mit `use` möglich ist.

Das Modul Frame enthält die Strukturen `Prepare` und `TraceUses`. In `find_files` (aus `Prepare`) wird der Parser aufgerufen und festgestellt, welche Dateien in einem Experiment verwendet werden. Dies wird durch einen Abstieg in den Operator-Baum ermöglicht. Die Funktion `start` läd dann mit `use` die noetigen Dateien nach und anschließend kann der gesamte Operator-Baum vom Parser übersetzt werden, da nun alle Operatoren und Funktionen bekannt sind.

Die Namen der geladenen Dateien werden dabei in der Struktur `TraceUses` vermerkt, so daß sie nicht bei jedem Neustart des Experiments geladen werden müssen. Dies verkürzt die Anlaufphase beträchtlich. Die Entscheidung, ob eine bereits einmal geladene Datei erneut mit `use` eingebunden wird, wird anhand

des Dateidatums gefällt. Durch Aufruf der Funktion `TraceUses.use_all ()` kann erreicht werden, daß alle Dateien geladen werden.

Sind alle für das Experiment benötigten Daten vorhanden, so wird die Berechnung durch einen Aufruf der Linker-Funktion `start_experiment` angestoßen. Die Funktion `start` übernimmt dabei auch das Abfangen von Ausnahmesituationen wie z.B. Laufzeitfehlern.

5.5.7 Ausblick

Ausgehend vom augenblicklichen Stand des Systems ist es denkbar, das Parsen von Operatoren neu zu organisieren:

- Parser: Verwaltungsinformation, die zum Überprüfen der Typkorrektheit bei Operator und Funktionsaufrufen dient, wird bei den Operatordaten abgespeichert, z.B. Operator-/ Funktionsnamen und übergebene Typen für jeden Aufruf. Hierdurch würde es möglich, auf den zweiten Parse-Vorgang zu verzichten und die Typkontrolle beim Aufbau des Operatorbaums vorzunehmen. Außerdem könnte der Parser vom Laufzeitsystem getrennt werden.
- Linker: Der Linker ist in der Lage, operator-übergreifende Typprüfungen durchzuführen, da er alle Operatoren kennt. Er könnte Kompatibilitätsprüfungen zwischen Problem, Kodierungen und Verfahren anstoßen.
- Bibliothek (*Library*): Es wäre möglich, Operatoren in der geparsten Form zu speichern. Die SML-Strukturen, die der Parser liefert, müßten in Dateien geschrieben werden.

5.5.8 Wörterbuch

Erklärung/Definition einiger hier verwendeter Begriffe

(Variablen-) Umgebung: Jeder Operator kann auf Variablen zugreifen. Um auf deren Werte auch außerhalb der Lebenszeit des Operators Zugriff zu haben, werden sie in einer Struktur gespeichert, die beim Start des Experiments erzeugt wird. Die Verschachtelung der Operatoren wird durch die Verschachtelung der Umgebungen ineinander wiedergegeben.

Elementar-Anweisung: Befehle der „Stackmachine“, aus denen der Ablaufteil aller Operatoren aufgebaut ist. Die Befehle sind bedeutend simpler als die dem Benutzer zur Verfügung stehenden Statements.

Statements: Konstrukte der Programmiersprache. Werden vom Parser in Elementar-Anweisungen der Stackmaschine umgesetzt.

Stackmachine: Maschinenmodell, das zur Ausführung der Operatoren benutzt wurde. Hierbei hält ein Stapel die Operanden für Anweisungen bereit.

Parameter-Deklaration: Bei der Definition eines Operators muß angegeben werden, welchen Wertebereich seine Parameter annehmen dürfen; daneben kann eine Beschreibung in Form eines Texts angegeben werden. Der Wert eines Parameters kann von hierarchisch höher liegenden Operatoren verändert werden, daher werden sie in der Variablenumgebung gespeichert.

Formale Parameter: Diese Variablen werden bei der Definition eines Operators in der Kopfzeile angegeben, wobei ihr Typ festgelegt wird. Bei einem Aufruf des Operators wird ihnen ein aktueller Wert zugewiesen, der von den aktuellen Parametern bestimmt wird.

Aktuelle Parameter: Ausdrücke, die in Operator-/ Funktionsaufrufen an der Stelle der formalen Parameter stehen. Die Werte dieser Ausdrücke werden vor dem Aufruf berechnet und beim Aufruf selbst an die formalen Parameter des Operators zugewiesen.

Lokale Variablen: Diese Variablen werden bei der Definition eines Operators angegeben, wobei ihnen ein Typ und ggf. ein Wert zugewiesen wird. Diese Variablen sind nur im Operator sichtbar.

Operator: Operatoren sind Prozeduren, die in der Interpretersprache geschrieben sind und zusätzliche Verwaltungsdaten enthalten. Sie werden vom Preter auf der Stackmaschine ausgeführt.

Operatorbezeichner: Unter diesem Namen wird ein Operator in einem anderen aufgerufen. Ein Operator kann mehrere Bezeichner haben, mit denen jeweils andere Belegungen der Parameter und Unteroperatorumgebungen verbunden sind.

Operatorname: Unter diesem Namen wird ein Operator in der Library angesprochen. Es kann nur jeweils einen Operator unter einem Namen geben.

Operatorbaum: Struktur der Operatoren in einem Experiment. An der Wurzel steht der Experimentoperator, direkt unter ihm die Verfahrensoperatoren. Da Rekursion der Operatoren nicht erlaubt ist, ist die Baumform garantiert.

Funktion: Eine Funktion ist in SML geschrieben und kann zwar andere Funktionen aufrufen, nicht aber Operatoren. Es gibt interne und Benutzer-Funktionen.

interne Funktion: Diese Funktionen werden vom System zur Verfügung gestellt und übernehmen Basisaufgaben wie Ermittlung einer Zufallszahl.

Benutzer-Funktion: Diese Funktionen sind in Bibliotheken zusammengefaßt und werden je nach Experiment benötigt oder nicht. Die Bibliotheken werden von SML eingelesen und durch Aufrufe an die Library ins System integriert. Es ist möglich (und oft beabsichtigt), daß in verschiedenen Bibliotheken Funktionen gleichen Namens existieren.

externe Funktion: Funktion, die nicht ausschließlich im SML-System berechnet wird. Z.B. könnte eine C- oder Unix-Funktion so an das System angebunden werden.

Bibliothek: Sammlung von Benutzer-Funktionen. Können neben Benutzer-Funktionen zur Ausführung einer evolutionären Berechnung auch Problem und Kodierungen umfassen. Auch die Einbindung externer Funktionen ist über Bibliotheken möglich.

Dispatcher-Funktion: Funktion, die von einer nachgeladenen Bibliothek an die Library übergeben wird. Sie übersetzt die Parameter einer Funktion von der Interpreterdarstellung in die SML-Form, ruft die Funktion auf und wandelt das Ergebnis zurück.

Üblicherweise werden diese Funktionen am Ende des Bibliothekenmoduls in einem Aufruf der Funktion `Library.add_disp` definiert, wobei das „`fn`"-Konstrukt von SML benutzt wird.

Library: Bezeichnung für den Programmteil, der Operatoren und Funktionen verwaltet („Bibliothek").

Interpretersprache: Diese Sprache wird vom Parser erkannt. Verfahren, Operatoren und Experimente können in ihr formuliert werden, so daß sie in das System integriert werden können.

evolutionäre Berechnung: Der Ablauf eines Experiments berechnet für ein Problem durch evolutionäre Verfahren eine Lösungsmenge.

Verfahren: Implementation eines Algorithmus zur Bearbeitung eines Parameteroptimierungsproblems. Verfahren arbeiten auf einer Population und können Operatoren und Funktionen verwenden.

Experiment: Das Experiment legt fest, welches Problem optimiert werden soll und welche Verfahren hierzu eingesetzt werden. Im Experimentoperator werden die Populationen deklariert und Kodierungen angegeben, daneben kann im Experimentoperator die Migration von Individuen zwischen Populationen vorgenommen werden.

Kapitel 6

Erweiterungsmöglichkeiten

6.1 Kritischer Rückblick

Ein Problem bei der Entwicklung des Systems ist das Fehlen von systematischen Tests für die einzelnen Module. Das System wurde zwar in seiner Gesamtheit mit einer Reihe von Verfahren getestet, doch war die Entwicklung dieser Verfahren eher konstruktiv. Es wurde nicht versucht, wie eigentlich bei einem Test erforderlich, mögliche Probleme und Fehler des Systems aufzudecken. Um dies durchzuführen wäre es notwendig gewesen, für jedes der Module einen Testplan zu erstellen, anhand dessen es ausführlich getestet werden kann. Dies sollte von einer Person gemacht werden, die nicht bei der Entwicklung des Moduls beteiligt war.

Der Aufbau der Kodierung ist geprägt durch den Wunsch nach einer möglichst großen Flexibilität. Die Erstellung von eigenen Kodierungen und Problemen muß auf Ebene von SML erfolgen und ist nur nach einer Einarbeitung in relativ komplizierte Zusammenhänge möglich. Oft entsteht der Wunsch, eigene Probleme und damit auch eigene Kodierungen zu verwenden, jedoch schon vor dem, eigene Verfahren zu entwickeln. Es wäre daher sinnvoll, den Einstieg durch eine weitere Schicht zu erleichtern, die nur eine eingeschränkte Funktionalität bietet, jedoch einfach zu verstehen ist.

Etwas unglücklich ist die Trennung in Operatoren, die in LEA geschrieben sind und Funktionen in SML. Das dadurch entstandene Problem der Anbindung von SML-Funktionen und Strukturen an den Interpreter führte zu einem Bibliothekenkonzept, das zwar recht flexibel, aber auch umständlich ist und genauere Kenntnisse zu seiner Benutzung nötig macht.

Tiefergehende Kenntnisse sind auch erforderlich, wenn beim Arbeiten mit dem System ein Fehler auftritt. Da ein globales Konzept für das Behandeln von Fehlern fehlt, werden Fehlermeldungen, die auf unterster Ebene erzeugt werden, einfach nach oben weitergegeben, was eine Lokalisierung des Fehlers schwierig macht, vor allem, wenn nur mit den oberen Ebenen gearbeitet wird. Auch die Fehlermeldungen des Parsers sind größtenteils durch dessen internen Aufbau bestimmt und in manchen Fällen schwierig zu verstehen. In anderen Teilen

werden manche Fehler gar nicht vom System abgefangen, sondern werden erst beim Aufruf einer SML-Funktion erkannt. So ist es z.B. bei den Kodierungen nur möglich, bestimmte Klassen von Fehlern zu erkennen.

Die Auswertung der Daten, die bei einem Experiment anfallen, wird nur zu einem kleinen Teil vom System unterstützt. Es ist nur möglich die gesamte Population in eine Log-Dateien zu schreiben und später auszuwerten. Weiterhin ist nur eine Auswertung der Fitneß der gespeicherten Individuen möglich, andere Informationen werden ignoriert.

Ganz fehlt die in den Anforderungen gewünschte Anbindung von externen Problemen, die für die Anwendung der im System entwickelten Verfahren auf praxisnahe Probleme sehr hilfreich wäre. Oft wird die Fitneß für solche Probleme mit aufwendigen Algorithmen bestimmt, die schon als ausführbares Programm existieren. Beispiele dafür sind Simulatoren oder Finite Elemente Methoden.

6.2 Konkrete Erweiterungen

Im Rahmen der Projektgruppe sind viele Ideen entstanden, wie ein umfassendes System zur Unterstützung der Entwicklung von Evolutinären Algorithmen aussehen könnte. Wegen der geringen Zahl der Mitglieder und der am Ende doch etwas knappen Zeit konnten viele dieser Ideen nicht umgesetzt werden. Bei einigen war schon ziemlich früh klar, daß sie nicht mehr in den Rahmen der Projektgruppe passen würden (z.B. eine graphische Benutzungsoberfläche). Andere wurden angedacht und konnten nicht mehr durchgeführt werden oder sind erst beim Testen des Systems entstanden. Die wichtigsten Ideen sollen hier kurz beschrieben werden.

6.2.1 Erweiterungen direkt am System

Im folgenden werden die Erweiterungen beschrieben, die an dem System vorgenommen werden können, ohne das sich wesentliche Konzepte ändern. Meistens sind nur einzelne Module des Systems betroffen. Weitere Erweiterungen, die nur geringere Auswirkungen auf den Rest des Systems haben, sind in der technischen Dokumentation für das jeweilige Teil beschrieben.

- Die Erzeugung von Log-Dateien und die Möglichkeiten zu deren Auswertung sind noch sehr eingeschränkt. Zur Zeit werden nur ganze Populationen in die Log-Dateien geschrieben, ausgewertet wird eigentlich nur die Fitneß der Individuen.

  Sinnvoll wäre die Möglichkeit, neben allgemeinen Bemerkungen auch die folgenden Daten zur späteren Auswertung in eine Log-Datei schreiben zu können:

  - Einzelne Individuen,
  - Zählerwerte der Verfahren,
  - eine Beschreibung des Experiments und

  - die Auswirkung von Operatoren auf Individuen.

- Für die Sprache LEA war eigentlich die Möglichkeit geplant, Parameter von untergeordneten Operatoren ändern zu können. Einige Teile des Interpreters sind bereits dafür ausgelegt. Es fehlt lediglich die Fähigkeit des Parsers, die dazugehörenden Konstrukte zu lesen und Funktionen der Bibliothek, die dem Parser die Möglichkeit geben, die Namen und Typen vorhandener Parameter zu ermitteln.

- Im Laufe der Berechnungen für ein Experiment sind viele Prüfungen möglich, z.B. auf Einhaltung des Wertebereichs bei Parametern. Damit dabei auftretende Fehlermeldungen einheitlich behandelt werden können, ist auch ein Konzept für die Behandlung von Laufzeitfehlern sowohl in der Sprache LEA als auch in den SML-Operatoren nötig.

- Die Fitneßfunktion, wie sie im System verwendet wird, ist eingeschränkt auf die Berechnung eines Real-Wertes für einen Phänotyp. Möglich wäre hier eine flexiblere Unterstützung von Verfahren, die mehr Informationen benötigen.

6.2.2 Weiterentwicklung des Systems

Für andere Erweiterungen ist die Erarbeitung von neuen Konzepten nötig, wie sie zur Zeit (z.B. für die Auswertung) noch nicht existieren, oder es sind umfangreiche Änderungen an mehreren Teilen des Systems nötig.

- Eine große Erleichterung für den Einstieg in das System könnte eine graphische Benutzungsoberfläche bieten. Dabei sind zwei Teile zu unterscheiden: Ein Teil, mit dem die Elemente der Bibliothek wie Probleme, Kodierungen, Operatoren, etc., erstellt, angezeigt und bearbeitet werden können. Dieser Teil dient dazu, ein Experiment zusammenzustellen. Mit dem zweiten Teil wird die Ausführung der Experimente gesteuert. Hier werden Einstellungen vorgenommen (z.B. Parameterwerte für die Operatoren) und Ergebnisse der Experimente verwaltet und angezeigt.

- Um Probleme und Kodierungen einfach erstellen zu können, sollten dafür einheitliche Darstellungen entwickelt werden. Wenn möglich so, daß die damit definierten Objekte immer gültig sind. Diese würden sich auch zur interaktiven Eingabe in einer Oberfläche eignen.

  Für diese Darstellungen sollten sich auch Prüfungen durchführen lassen, mit denen schon vor Programmstart entschieden werden kann, welche Probleme, Kodierungen und vielleicht auch Operatoren zueinander passen und welche nicht.

- Das Programmieren von Evolutionären Algorithmen wird dadurch erschwert, daß die Sprache LEA nur bis zur Ebene der Individuen verwendet werden kann und darunter SML-Operatoren geschrieben werden müssen, deren Anbindung an LEA etwas kompliziert ist. Um dies zu beseitigen, müßte LEA so erweitert werden, daß auch die Geno- und Phänostrukturen bearbeitet werden können. So könnten auch die Fitneßfunktionen in LEA geschrieben werden.

- Schon angedacht und auch teilweise schon unterstützt sind Haltepunkte und Einzelschrittmodus für den Interpreter. Um diese einfach benutzen zu können, ist die Verwendung einer (graphischen) Benutzungsoberfläche sinnvoll. Der Parser erzeugt zu den Stackprogrammen Debug-Code, der angibt, welche Position im Stackprogramm zu jeder Zeile des Programmtexts gehört. In der graphischen Oberfläche kann dann an einer Zeile ein Haltepunkt gesetzt werden (Break), an dem das System anhält. Einzelne Werte aus der Umgebung können für den bis dahin erreichten Zustand angezeigt und auch verändert werden. Genauso ist ein Einzelschrittmodus (Trace) möglich, bei dem nach jeder Zeile angehalten wird.

- Innerhalb einer graphischen Benutzungsoberfläche sind auch die unterschiedlichsten Möglichkeiten für eine graphische Anzeige von Informationen zum Ablauf eines Verfahrens denkbar. Es sollte daher möglich sein, für bestimmte Verfahren und Probleme spezielle Anzeigemodule einzubinden. So kann z.B. für bestimmte Probleme die Position von Individuen im Lösungsraum angezeigt werden. Für manche Verfahren könnte es auch sinnvoll sein, anzuzeigen, wie sich ein Individuum durch den Lösungsraum bewegt.

- Zur Anbindung externer Probleme sollte eine definierte Schnittstelle entwickelt werden, mit der Daten mit einem ausführbaren Programm ausgetauscht werden können, das dann die Fitneß für ein Individuum oder eine Gruppe von Individuen berechnet.

Kapitel 7

Bedienung

Um den Einstieg in das Arbeiten mit GENOM zu erleichtern, soll in diesem Abschnitt eine Einführung in die Bedienung des Systems gegeben werden. Diese Einführung erfolgt im Weiteren in mehreren Schritten, die den verschiedenen Schwierigkeitsgraden entsprechen, in denen mit dem System gearbeitet werden kann. Je tiefer eine Schicht liegt, desto größer ist der Umfang, in dem sie verändert werden kann, aber auch die Kenntnisse, die zu ihrem Verständnis nötig sind. Die einzelnen Schichten sind:

1. Das Verwenden von vorgefertigten Problemen und Verfahren. Diese können miteinander kombiniert und deren Parameter angepaßt werden.
2. Eigenen Operatoren und Verfahren zu schreiben.
3. Neue Probleme einzubinden, neue Kodierungen und die dazugehörenden Funktionen auf SML-Ebene zu erstellen.

Die im folgenden verwendete Einteilung in Experimente, Verfahren und Operatoren wird nur teilweise von LEA erzwungen. Das hier verwendete Konzept ist dazu gedacht, möglichst wiederverwendbare Verfahren und Operatoren zu ermöglichen und eine übersichtlichere Aufteilung zu bewirken. Es ist ratsam, diese vorgesehene Aufteilung auch bei der Erstellung eigener Verfahren zu berücksichtigen.

Das Vorgehen beim Arbeiten mit GENOM wird in den folgenden Abschnitten anhand eines Genetischen Algorithmus erklärt. Die dazugehörenden Dateien befinden sich im Bibliotheksverzeichnis des Systems.

7.1 Erste Schritte

7.1.1 Aufbau des Systems

Die einzelnen Komponenten, aus denen sich ein Experiment zusammensetzt, sind im Unterverzeichnis `lib` des Systemverzeichnisses gespeichert. Die verschiedenen Komponenten befinden sich in den folgenden Unterverzeichnissen:

Komponenten	Verzeichnis	Endung
Experimente	`/experiments`	`.exp`
Probleme	`/problems`	`.sml`
Kodierungen	`/coding`	`.sml`
Verfahren und Operatoren	`/operators`	`.eva`
SML-Operatoren	`/ml-operators`	`.sml`

Der Gesamtaufbau des Systems wird im entsprechenden Kapitel ausführlich beschrieben.

7.1.2 Laden des Systems

Um das System zu laden, muß zuerst in das Verzeichnis `sml` des Systems gewechselt werden (alle Pfade sind relativ zu diesem Verzeichnis). Dort wird der SML-Interpreter in der Version 1.09 geladen. Wie er aufgerufen wird, hängt von dessen Installation ab. Meist geschieht dies durch Eingabe von `sml` oder `sml-109`. Wenn der SML-Interpreter geladen ist, kann das System gestartet werden, indem nach dem Prompt `use "system.sml"` eingegeben wird. Dadurch wird die Datei `system.sml` ausgeführt, die die Befehle enthält, mit denen das System in die SML-Umgebung geladen wird.

7.1.3 Aufruf eines Experiments

Nachdem das System geladen ist, kann unter SML mit dem Befehl `start` ein Experiments ausgeführt werden. Die Dateien mit den Experimentdefinitionen befinden sich alle im Unterverzeichnis `lib/experiments` des Systemverzeichnisses. Der Name eines Experiments ist der Name der Datei, ohne die Endung `.exp`. Für das Experiment „TestGenAlg“ sieht der Aufruf so aus:

```
start("TestGenAlg");
```

7.1.4 Beenden von SML

Ein Experiment des Systems kann unter SML mit `Ctrl-C` abgebrochen werden, falls dies nötig sein sollte. Mit `Ctrl-D` wird SML verlassen.

7.2 Einführung in LEA

Für die nächsten Schritte sind Kenntnisse in der Sprache LEA nötig. In dieser Sprache werden die vom System verwendeten Experimentdefinitionen, Verfahren und allgemeinere Operatoren geschrieben. LEA ist eine prozedurale Sprache und stark an Sprachen wie PASCAL oder MODULA2 angelehnt. Wem eine dieser Sprachen geläufig ist, sollte auch gut mit LEA zurechtkommen. Eine Einführung in LEA befindet sich in der technischen Dokumentation.

7.3 Zusammenstellen von Experimenten

Wenn nur vorhandene Verfahren auf eine Testfunktionen angewendet werden soll, reicht es, eine neue Experimentdefinition zu schreiben (oder eine vorhandene abzuwandeln). Hier wird nur ein einfaches Experiment mit einer Population und einem Verfahren gezeigt. Das Zusammensetzen eines Experiment geschieht in mehreren Schritten.

1. Zuerst muß eine Textdatei im Verzeichnis `lib/experiments` des Systems angelegt werden, deren Namen dem des Experiments entspricht und zusätzlich die Endung `exp` hat. Die Kopfzeile des Experiments muß ebenfalls diesen Namen enthalten. Wenn das Experiment den Namen `TestGenAlg` haben soll, muß also eine Datei `TestGenAlg.exp` erstellt werden, deren Kopfzeile folgendermaßen aussieht:

   ```
   EXPERIMENT TestGenAlg;
   ```

2. Auswahl eines vorhandenen Problems: Die vorhandenen Probleme sind in den Dateien des Verzeichnisses `lib/problems` gespeichert. Sie haben die Endung `sml`. Ein Problem wird in einer Experimentdefinition mit dem Schlüsselwort `PROBLEM` angegeben. Der Name des Problems entspricht dann wieder dem Dateinamen ohne die Endung. Für das Problem „Hypersphere“ (aus der Datei `Hypersphere.sml`) sieht die entsprechende Zeile so aus:

   ```
   PROBLEM = "Hypersphere";
   ```

3. Deklaration einer Population: Die meisten Verfahren arbeiten auf einer Population; daher ist es ratsam eine zu verwendet, auch wenn sie nur ein Individuum enthalten soll. Populationen werden in einem eigenen Block deklariert, der mit dem Schlüsselwort `POPULATIONS` beginnt. Für eine Population müssen die folgenden Angaben gemacht werden:

   - eine Kodierung, mit der die Individuen kodiert sind,
   - eine Log-Datei, in die die Population geschrieben werden kann und
   - wie die Population initialisiert werden soll.

   Welche Kodierungen auf welche Probleme angewendet werden können, kann in der Tabelle bei der Beschreibung der Bibliothek im Anhang ersehen werden. Um eine Population mit 20 Individuen, die als Bitstrings kodiert sind, und der Log-Datei `GenAlg.log` zu erstellen, muß die Deklaration folgendermaßen aussehen:

   ```
   Pop CODED "GenAlgGrayCod" LOG "GenAlg" = RANDOMPOP (20);
   ```

4. Auswahl eines Verfahrens: Um ein Verfahren (oder einen Operator) in dem Experiment zu verwenden, muß ein Verweis darauf erstellt werden. Dabei werden den Parametern dieses Verfahrens eigene Werte zugewiesen

(ähnlich wie das Erzeugen der Instanz eines Objekts bei einer objektorientierten Sprache). Wenn einem Parameter nicht speziell ein Wert zugewiesen wird, erhält er seinen Default-Wert. Im Programmteil des Experiments kann das Verfahren über diesen Verweis aufgerufen werden. Verweise auf Verfahren und Operatoren werden in LEA unter dem Schlüsselwort `OPERATORS` deklariert.

```
Alg = GenAlg(mue: 100);
```

5. Wenn im Programmteil des Experiments bestimmte Funktionen benötigt werden, müssen die Bibliotheken, die diese enthalten, mit `USES` angegeben werden. Für manche Verfahren, die Genotypen mit verschiedenen Strukturen bearbeiten können, müssen die Bibliotheken angegeben werden, die die Funktionen enthalten mit denen die Struktur, die sich aus dem Problem ergibt, bearbeitet werden kann (z.B. Mutationsfunktionen). Bei dem Genetischen Algorithmus ist dies nicht nötig. Eine nähere Beschreibung der dazugehörenden Mechanismen findet sich in der technischen Dokumentation.

6. Im Anweisungsteil des Experiments kann für das Beispiel einfach der oben beschriebene Verweis aufgerufen werden. Alles weitere wird von dem Verfahren erledigt.

   ```
   Alg(Pop);
   ```

   Der Anweisungsteil für das Experiment ist in diesem Beispiel sehr kurz. Es können hier beliebige LEA-Programme stehen. Dazu stehen alle Konstrukte und Befehle von LEA zur Verfügung, die auch in den Operatoren vergewendet werden können.

Aus den oben beschriebenen Teilen setzt sich die gesamte Experimentdefinition zusammen.

```
EXPERIMENT TestGenAlg;
  PROBLEM = "Hypersphere";
  POPULATIONS
    Pop CODED "GenAlgGrayCod" LOG "GenAlg" = RANDOMPOP (20);
  OPERATORS
    Alg = GenAlg(mue: 100);
BEGIN
  Alg(Pop);
END
```

7.4 Erstellen von Verfahren

Der nächste Schritt ist die Erstellung von eigenen Verfahren. Verfahren und Operatoren können wie Funktionen (z.B. in PASCAL) Argumente erhalten und einen Rückgabewert zurückliefern. Ein Verfahren sollte eine oder mehrere Populationen als Argumente übergeben bekommen. Wenn sinnvoll, kann es auch weitere Argumente und einen Rückgabewert haben.

```
ALGORITHM GenAlg(POP Pop);
```

So wie die Experimentdefinition Verweise für Verfahren benützt, müssen in einem Verfahren Verweise für die dort verwendeten Operatoren erstellt werden. Auch hier können deren Parameter mit den gewünschten Werten belegt werden.

Ein Ziel bei der Erstellung von Verfahren (und auch Operatoren) ist, diese möglichst wiederverwendbar und unabhängig von Problem und Kodierung zu halten. Das Verhalten eines Verfahrens sollte daher durch eine Reihe von Parametern gesteuert werden. Für Parameter von Verfahren und Operatoren können in LEA ein Standard-, ein Minimal- und ein Maximalwert, sowie ein beschreibender Text angegeben werden. Da der Standardwert immer dann genommen wird, wenn nicht anderes angegeben ist, sollte dafür ein Wert gewählt werden, für den das Verfahren gute Ergebnisse liefert.

Ein Beispiel für einen geeigneten Parameter eines genetischen Algorithmus ist die Zahl der erzeugten Nachkommen `mue`.

```
PARAMETER
  INT mue = (20, 1, 1000000, "Number of Children");
```

Da die Log-Dateien für komplexere Individuen größere Ausmaße annehmen können, ist es sinnvoll, wenn jedes Verfahren einen Parameter hat, der angibt, ob in die Log-Datei geschrieben werden soll. Eine andere Möglichkeit ist ein Parameter, der angibt, nach welcher Zahl von Generationen in die Log-Datei geschrieben werden soll.

```
  BOOL writelog = (FALSE, FALSE, TRUE, "Write a Log");
```

Der Hauptteil eines Verfahrens besteht meist aus einer Schleife, innerhalb der die neue Generation berechnet wird. Um die Operatoren wie z.B. Crossover und Selektion auch auf Teilmengen von Populationen anwenden zu können, arbeiten diese meist nicht auf Populationen, sondern auf Listen von Individuen (Typ: `INDLIST`). Daher sollte am Anfang der Schleife die ganze Population in eine Individuenliste gelesen werden, die nach Bearbeitung durch die Operatoren wieder zurück in die Population geschrieben wird. Ein besonderer Fall sind die Abbruchbedingungen. Diese geben für ein Verfahren an, ob die Berechnung weit genug fortgeschritten ist. Da diese nicht im inneren Schleifenrumpf verwendet werden, arbeiten diese direkt auf der Population.

```
WHILE (NOT <Abbruchbedingung>(<Pop1>)) DO
  <IndList1> := get_Pop(<Pop1>);
  <IndList2> := <Operator1>(<IndList1>,...);
  ...
  <IndListn> := <Operatorn-1>(<IndListn-1>,...);
  set_Pop(<Pop1>, <IndListn>);
  incGenCount(<Pop1>);
  IF writelog THEN
    logPop(<Pop1>)
  FI;
OD
```

Nach Zurückschreiben der Individuen in die Population am Ende der Schleife wird der Generationszähler hochgezählt und die neue Generation in die Log-Datei der Population geschrieben. Dies ist notwendig, damit eine Auswertung, die vom System unterstützt wird, vorgenommen werden kann. Natürlich muß innerhalb der Hauptschleife keine Sequenz, wie oben gezeigt, eingehalten werden. Es können hier auch beliebige LEA-Konstrukte wie Schleifen und Verzweigungen verwendet werden. Für den Genetischen Algorithmus ergibt sich konkret:

```
ALGORITHM GenAlg(POP Pop);
  USES PopHandler, IndList;
  PARAMETER
    INT mue = (20, 1, 1000000, "Number of Children");
    BOOL writelog = (FALSE, FALSE, TRUE, "Write a Log");
  OPERATORS
    StopCond = CntGenStopCond(Generations: 500);
    Select = ElitistPropSelect(WorstIndFact: 0.2);
    Recomb = Crossover(Points: 2, nue: 1.0);
    Mutate = GAMutate(Prob: 0.005);
  VAR
    INDLIST IndList, Parents, Children, NewIndList;
    INT lambda;
BEGIN
  WHILE (NOT(StopCond(Pop))) DO
    IndList := get_Pop(Pop);
    lambda := length(IndList);
    Children := Recomb(IndList ,mue);
    Children := Mutate(Children);
    NewIndList := Select(merge(IndList, Children), lambda);
    set_Pop(Pop, NewIndList);
    incGenCounter(Pop);
    IF writelog THEN
      logPop(Pop)
    FI;
  OD;
END;
```

In diesem Verfahren werden die Bibliotheken `PopHandler` und `IndList` verwendet. Die erste Bibliothek stellt die Schnittstelle zur Populationsverwaltung dar. Sie enthält z.B. die hier verwendete Funktion `get_Pop`. Aus der zweiten stammen die Funktionen, die allgemeine Operationen auf Individuenlisten durchführen (z.B. `merge`).

7.5 Operatoren

Operatoren sind wie die Verfahren aufgebaut. Nur sollten hier keine Populationen übergeben, sondern nur mit Listen von Individuen oder einzelnen Individuen gearbeitet werden. Eine Ausnahme davon sind Operatoren, die eine Abbruchbedingung berechnen.

Bei Operatoren, die in mehreren Verfahren verwenden werden, sollte hier besonders auf die Verwendung von geeigneten Parametern geachtet werden.

Da in LEA Individuen nicht verändert werden können, stellen Operatoren die unterste Ebene des Operatorbaums dar, die in LEA programmiert wird. Wenn es sich um Operatoren handelt, die mit beliebigen Individuen auskommen, können diese auch komplett in LEA geschrieben werden. Operatoren dieser Art sind z.B. manche Selektionsoperatoren oder Abbruchbedingungen. Andernfalls müssen SML-Funktionen eingebunden werden, die mit den kodierten Individuen einer Population arbeiten können.

7.5.1 Operatoren auf Individuenlisten

Zu dieser Gruppe gehören, neben den Selektionsoperatoren, die aus einer Individuenliste eine Anzahl von Individuen für einen weiteren Bearbeitungsschritt auswählen, auch Operatoren, die eine Mutation oder Rekombination auf den Individuen der Liste durchführen. Für den in diesem Beispiel verwendeten GA sind das die Operatoren `GAMutate` und `Crossover`. Die Kopfzeile eines solchen Operators entspricht dem folgenden Schema:

```
OPERATOR <Name>(INDLIST IndList, ...): INDLIST;
```

Der Operator `GAMutate` besitzt den Parameter `Prob`, der angibt, mit welcher Wahrscheinlichkeit die einzelnen Bits eines Bitstrings umgedreht werden.

```
REAL Prob = (0.05, 0.0, 1.0, "Probability for mutation");
```

Um die Mutation der Individuen durchzuführen, ruft `GAMutate` die SML-Funktion `mutate` aus der Bibliothek `ga_mutate` auf. Die Aufgabe des Operators besteht darin, diese Funktion auf alle Individuen der Liste anzuwenden und den Parameter für die Mutationswahrscheinlichkeit zu definieren. Oft ist es sinnvoll auch für SML-Funktionen, die direkt aus einem Verfahren aufgerufen werden sollen, einen Operator zu schreiben, der deren Parameter definiert und sinnvolle Standardwerte vorgibt. Der vollständige Operator `GAMutate` sieht so aus:

```
OPERATOR GAMutate(INDLIST IndList): INDLIST;
  USES IndList, Math, ga_mutate;
  PARAMETER
    REAL Prob = (0.05, 0.0, 1.0, "Probability for mutate");
  VAR
    IND indiv;
    INT i;
BEGIN
  FOR i := 1 TO length(IndList) DO
    indiv := getListInd(IndList, i);
    indiv := mutate(indiv, Prob);
    IndList := setListInd(IndList, i, indiv);
  OD;
  RETURN IndList;
END;
```

7.5.2 Operatoren für Abbruchbedingungen

Eine Abbruchbedingung zeigt an, ob eine Population eine bestimmte Bedingung erreicht hat. Wenn diese wahr wird, wird die Berechnung von weiteren Generationen abgebrochen. Es gibt verschiedene Möglichkeiten diese Bedingung zu realisieren. So kann z.B. nach einer bestimmten Anzahl von Generationen abgebrochen werden oder wenn die Fitness der Individuen sich um weniger als einen vorgegebenen Wert unterscheidet. Der Abbruchbedingung wird eine Population übergeben. Sie gibt einen bool'schen Wert zurück, der angibt ob die entsprechende Bedingung erfüllt ist oder nicht.

```
OPERATOR <Abbruchbedingung>(POP <Pop>): BOOL;
```

Als Beispiel wird hier eine Abbruchbedingung gezeigt, die wahr wird, wenn eine bestimmt Anzahl von Generationen berechnet wurde:

```
OPERATOR CntGenStopCond(POP Pop): BOOL;
  USES PopHandler;
  PARAMETER
    INT Generations
         = (100, 0, 100000, "Stop after generation");
BEGIN
  IF (getGenCounter(Pop) > Generations) THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  FI;
END;
```

7.6 Anbinden von Funktionen in SML

Viele Operatoren enthalten Funktionen, die direkt auf den Genostrukturen der Individuen arbeiten müssen und nur in SML programmiert werden können. Auch bei langwierigeren Berechnungen sollten SML-Funktionen verwendet werden, da diese um einiges schneller sind. Um diese Funktionen in LEA verwenden zu können, müssen sie in einer Bibliothek in das System eingebunden werden. Auf die in einer Bibliothek enthaltenen Funktionen kann zugegriffen werden, wenn diese Bibliothek mit `USES` in diesem oder einem übergeordneten Operator deklariert wird.

7.6.1 Erstellen von Bibliotheken

Um eine SML-Funktion von LEA aus aufrufen zu können, muß sie in einer Bibliothek gespeichert werden, die in dem System verfügbar wird, wenn sie ein Experiment oder Verfahren bzw. Operatoren mit `USES` öffnet. Die zu einem bestimmten Verfahren gehörenden SML-Funktionen werden oft in einer Bibliothek zusammengefaßt.

7.6.1.1 Grundlagen

Der Interpreter kann Funktionen aus Benutzer-Bibliotheken nur dann auffinden und verwenden, wenn sich die Bibliothek beim Modul `Library` hat registrieren lassen. Dies kann sie durch einen Aufruf der Funktion `Library.add_disp` erreichen. Obwohl in einer Struktur beliebige Funktionen (z.B. Konstanten oder Hilfsfunktionen) enthalten sein können, werden nur die beim Modul `Library` registrierten exportiert und in LEA verwendbar.

Der Funktion `Library.add_disp` wird der Name der Bibliothek übergeben, d.h. ein String. Dieser Name muß derselbe sein, der in einem Operator im `USES`-Konstrukt deklariert wurde und er muß ebenfalls der Namen der Datei ohne die Endung `.sml` sein.

Das zweite Argument ist eine Liste aus Elementen vom Typ `Dispatchers`:

```
type Dispatchers = string * (Types list * Types) *
((Types.Varvalue list) -> Types.Varvalue)
```

Diese Tripel enthalten als erstes den Namen der Funktion. Unter diesem Namen wird die Funktion vom Interpreter aufgerufen. Danach wird ein Tupel angegeben, dessen erstes Element die Typen der Eingabeparameter beschreibt, das zweite Element beschreibt den Typ des Rückgabewerts. Soll die Funktion keine aktuellen Parameter erhalten, so muß eine leere Liste übergeben werden. Gibt sie keinen Wert zurück, so muß als Resultatstyp der Typ `tp_notype` angegeben werden und die Funktion selbst den Typ `notdeklared` liefern.

Eine Besonderheit ist die mögliche Polymorphie der Eingabewerte einer Funktion. Wird statt eines konkreten Typs der Typ `tp_unknown` angegeben, so akzeptiert der Parser jeden Typ. Es muß aber sichergestellt werden, daß die verwendete Funktion auch auf jeden Typ richtig reagieren kann!

Das dritte Element des Tripels ist eine Funktion, die eine Liste aus den obersten Elementen des Stacks bei Aufruf der Funktion erhält. Die Typen der Elemente wurden im zweiten Element als Eingabetypen deklariert. Die Funktion liefert einen Wert vom Rückgabetyp zurück.

Beispiel Die Berechnung der Fitness eines Individuums wird implementiert, indem die Funktion `Fitness` beschrieben wird. Sie erhält ein Individuum als Eingabeparameter, daher der Typ `tp_ind`, und liefert einen reellen Wert (`tp_real`). Wird die Funktion ausgeführt, so „schält“ sie das Individuum aus der einelementigen Liste (`indival()`) und übergibt es an die Funktion `Fitness` des Moduls `PopHandler`. Den resultierenden reellen Wert „wickelt“ sie in `realval()` ein, so daß der Interpreter ihn als Real erkennt. Wird der Funktion eine nicht passende Liste übergeben, so löst sie eine Exception aus, in diesem Fall `type_mismatch`.

```
("Fitness", ([tp_ind], tp_real),
fn [indival(ind)] => realval(PopHandler.Fitness(ind))
| _ => raise type_mismatch "Fitness"
```

7.6.1.2 Konventionen

Beispielhaft sei die Bibliothek `ListBasics` beschrieben, in der einige Funktionen zur Behandlung von Listen definiert werden.

- Header: Jede Bibliothek beginnt mit einigen Zeilen Information über die Bibliothek:

```
(* Author : Thomas Schmidt
 * Date   : 13.07.96, 25.07.96, 21.08.96
 * File   : lib/ml-operators/ListBasics.sml
 * Use for: Basic Functions on Lists
 *)
```

- Die Funktionen selbst werden in einer Struktur zusammengefaßt, so daß auf oberster SML-Ebene die neuen Funktionen keine Seiteneffekte durch Überlagerung anderer Funktionen produzieren können. Der Name der Struktur kann prinzipiell beliebig gewählt werden, sollte aber aus Gründen der Übersichtlichkeit identisch mit dem Namen der Bibliothek sein.

```
structure ListBasics = struct
```

- Praktischerweise kann innerhalb der Struktur das Modul `Types` geöffnet werden, wodurch die Notwendigkeit entfällt, jede Typangabe mit `Types.` einzuleiten:

```
  open Types
```

- Es ist weiterhin zweckmäßig, den Namen der Bibliothek in einer Variablen abzulegen und eine Exception als Funktion zu definieren:

```
  val libname = "ListBasics"
  fun type_mismatch s =
    Error.runtime_error ("SML-Operator", s,
                         "Type-Mismatch")
```

- Danach läßt sich die Bibliothek beim Modul `Library` registrieren. In diesem Fall können die eigentlichen Funktionen direkt angegeben werden; es wäre natürlich auch möglich diese Funktionen zunächst explizit zu formulieren. Es ist sinnvoll, zu jeder Funktion eine Beschreibung der Sematik anzugeben.

```
val _ = Library.add_disp
  (libname,
    (* Return an empty Individual-List *)
   [("empty_list", ([], tp_indlist),
     fn [] => indilistval(nil)
      | _ => raise type_mismatch(libname^".empty_list")),

    (* Append an Individual to a List of Individuals *)
    ("append", ([tp_ind, tp_indlist], tp_indlist),
     fn [indival(ein), indilistval(liste)]
          => indilistval(ein::liste)
      | _ => raise type_mismatch(libname ^ ".append")),

    (* Merge two Individual-Lists *)
```

```
("merge", ([tp_indlist, tp_indlist], tp_indlist),
 fn [indilistval(vorn),indilistval(hinten)]
      => indilistval(vorn@hinten)
  | _ => raise type_mismatch(libname ^ ".merge"))

])
```

- Schließlich muß die SML-Struktur noch geschlossen werden:

  ```
  end
  ```

7.7 Eigene Probleme

Wenn nicht nur die mit dem System mitgelieferten Probleme verwendet werden sollen, können auch eigenen Probleme erstellt werden. Dazu muß im Verzeichnis `lib/problems` eine Datei erstellt werden, die den gewünschten Namen des Problems und die Endung `sml` hat. Der Aufbau dieser Dateien wird im folgenden beschrieben.

7.7.1 Grundlagen

Ein Problem besteht im System aus einer Fitnessfunktion und einer Funktion, die Phänotypen liefert.

Die Fitneßfunktion erwartet einen Phänotyp und gibt eine reelle Zahl zurück. Es wird festgelegt, daß alle Probleme *Minimierungsprobleme* sind – jedes Maximierungsproblem kann durch Negation in ein Minimierungsproblem konvertiert werden.

Die Funktion zur Erzeugung eines Phänotyps muß die Funktion `PhenoType.init` benutzen, da Individuen im System als abstrakte Datentypen implementiert wurden. Diese Funktion erwartet zwei Parameter: zuerst eine Funktion, die für natürliche Zahlen jeweils eine Zelle liefert, dann die Anzahl der Zellen, aus der der Phänotyp besteht.

Wird nun ein Phänotyp benötigt, so wird die Funktion `PhenoType.init` aufgerufen, die für jede der Zahlen zwischen 1 und der angegebenen Zellenzahl die Zellen-Generierungsfunktion aufruft. Ergebnis ist ein Phänotyp.

Ein neues Problem muß sich bei der Populationsverwaltung registrieren lassen. Dies wird durch einen Aufruf der Funktion `PopHandler.Set_Problem` erreicht. Als Parameter müssen dabei die Phänotyp-Generierungsfunktion und die Fitnessfunktion übergeben werden:

```
val _ = PopHandler.Set_Problem
  (fn () => PhenoType.init (cell_n, number_of_cells),
   fn x => (evaluate x))
```

7.7.2 Konventionen

- Jedes Problemmodul sollte mit einigen Zeilen Informationen über die Datei beginnen:

```
(* Author : Thomas Schmidt
 * Date   : 14.06.96, 21.08.96
 * File   : lib/problems/Hypersphere.sml
 * Use for: A simple Problem (Hypersphere) incl. Phenotype
 *)
```

- Alle Funktionen werden in einer Struktur zusammengefaßt. Sinnvollerweise wird sie `Problem` genannt.

```
structure Problem =  struct
```

- Das Problem sollte einige Informationen deklarieren, durch die z.B. Kodierungen sich an das Problem anpassen können. Diese Information ist sinnvoll, jedoch nicht immer praktisch angebbar.

```
val number_of_cells = 20
val type_of_cells = "BoundRealAtom"
val name = "Hypersphere"
```

- Bei der Verwendung reeller Zahlen aus einem Intervall müssen dessen Grenzen angegeben werden.

```
val min_real = ~5.12
val max_real = 5.11
```

- Die eigentliche Berechnung der Fitness kann beliebig kompliziert von statten gehen. Im Beispiel „Hypersphäre“ besteht sie jedoch nur aus wenigen Zeilen.

```
fun hypersphere (nil) = 0.0
  | hypersphere (r::tl) = r * r + hypersphere(tl)
```

- Um das Problem an die Populationsverwaltung übergeben zu können, muß neben der Fitneßfunktion `problem` auch eine Generierungsfunktion für Phänotypen angegeben werden (`PhenoType.init`). Deren erstes Argument ist eine Funktion, die eine Zellen liefert. Im Falle der mathematischen Probleme auf reellen Zahlen existiert bereits eine solche Funktion in der Struktur `IndLib`, die an die Initialisierungsfunktion übergeben werden kann (`simple_real_ind`). Der zweite Parameter legt die Anzahl der Zellen im Phänotyp fest.

  Der zweite Teil des Problems ist eine Funktion, die ein Individuum bewertet (letzte Zeile). Dazu muß aber das Individuum in eine Form gebracht werden, die zur Problemfunktion paßt. Hier reicht es, `simple_reals2list` auf das Individuum anzuwenden, denn die Problemfunktion kann die entstehende Liste reeller Zahlen bearbeiten.

```
val _ = PopHandler.Set_Problem
    (fn () => PhenoType.init
     (IndLib.simple_real_ind (number_of_cells,
                              min_real, max_real),
      number_of_cells),
     fn x => (problem (IndLib.simple_reals2list x)))
```

7.8 Kodierungen

Manche Verfahren arbeiten nur mit einem Genotyp, der eine bestimmte Struktur hat. Ein Genetischer Algorithmus benötigt z.B. einen Bitstring. Um ein vorgegebenes Problem an dieses Verfahren anzupassen, wird eine Kodierung benützt. Im folgenden soll beschrieben werden, wie eigene Kodierungen erstellt werden können. Die Beschreibung gliedert sich in zwei Abschnitte: Im ersten Abschnitt wird die Erstellung eines Kodierungsschemas aus bereits vorhanden elementaren Kodierungsschemata erläutert. Für Standardanwendungen sollte dieses Vorgehen der Normalfall sein. Die Erzeugung elementarer Kodierungsschemata wird im zweiten Abschnitt beschrieben.

7.8.1 Kodierungsschema

Als Beispielanwendung dient ein TSP, als Optimierungsverfahren soll eine Evolutionsstrategie eingesetzt werden. Die Permutation der Länge 14 wird in eine gleichlange Liste von reellen Atomen kodiert, außerdem soll der Genotyp 14 ebenfalls reellwertige Strategieparameter enthalten. Folgende elementare Kodierungsschemata stehen zur Verfügung: `Perm2Reals` kodiert eine Permutation in eine Liste reeller Zahlen gleicher Länge, `Stratlist14` erzeugt eine Liste aus 14 reellen Atomen.

Der Phänotyp besteht aus einer Permutationszelle, der Genotyp aus zwei Listenzellen.[1] Die elementaren Kodierungsschemata im Verzeichnis `lib/coding` (hier: `Perm2RealsCoding.sml` und `Stratlist14Coding.sml`) müssen nachträglich geladen werden, dann stehen die Strukturen `Perm2Reals` und `Stratlist14` zur Verfügung. Mit der Anweisung

```
val meinkodsname = ("Name fuer diese Kodierung",
                    [(Perm2Reals.elemcodscheme, [1], [1]),
                     (Stratlist14.elemcodscheme, [],[2])];
```

kann das Kodierungsschema einer Variable zugeordnet werden. Dann können mit

```
val eingenotyp = Coding.codeind(einphaenotyp,
                                meinkodsname);
```

[1] Die Kodierungsfunktionen prüfen diesen Aufbau *nicht*. Der Benutzer ist allein dafür verantwortlich, die Individuen passend zu den von ihm gewählten Kodierungsschemata zu initialisieren

bzw.

```
val nocheinphaenotyp = Coding.decodeind(eingenotyp,
                                        meinkodsname);
```

Individuen kodiert bzw. dekodiert werden. Um diese Kodierung in Experimenten verwenden zu können, muß beim System angemeldet werden (dies geschieht mit der Funktion `Library.add_coding()`). Zweckmäßigerweise definiert man hierfür eine eigene Struktur:

```
(* elementare Kodierungsschemata laden *)
use "../lib/coding/Perm2RealsCoding.sml";
use "../lib/coding/Stratlist14Coding.sml";

(* Kodierungsschema definieren und anmelden *)
structure BeliebigerName =
    struct
        val _ = Library.add_coding
            ("Name fuer diese Kodierung",
             ("Name fuer diese Kodierung",
              [(Perm2Reals.elemcodscheme, [1], [1]),
               (Stratlist14.elemcodscheme, [],[2])]))
    end;
```

7.8.2 Elementare Kodierungsschemata

Ein elementares Kodierungsschema besteht aus einer Struktur zu folgender Signatur:

```
signature ELEMENTARY_CODING_SCHEME =
    sig
        val in_cell_names: string list
        val out_cell_names: string list
        val coding: CellTypes.cell_type list
                    -> CellTypes.cell_type list
        val decoding: CellTypes.cell_type list
                      -> CellTypes.cell_type list
    end;
```

`in_cell_names` und `out_cell_names` sind für Konsistenzprüfungen vorgesehen und werden momentan nicht verwendet. Da Strukturen nicht Elemente von Listen oder Tupeln sein können, muß der Inhalt der Struktur in ein Record geschrieben werden. Dazu dient der Funktor `GetElementaryCodingScheme` mit der Signatur

```
signature GET_ELEMENTARY_CODING_SCHEME =
    sig
        structure Coding: CODING
        val elemcodscheme: Coding.elementary_coding_scheme
    end;
```

Ein elementares Kodierungsschema sieht dann z.B. so aus:

```
structure Ident_cod : ELEMENTARY_CODING_SCHEME =
    struct
        val in_cell_names = ["any"]
        val out_cell_names = ["any"]
        fun coding x = x
        fun decoding x = x
    end;

structure Ident: GET_ELEMENTARY_CODING_SCHEME =
    GetElementaryCodingScheme(Ident_cod)
```

Das elementare Kodierungsschema kann dann als `Ident.elemcodscheme` in einem Kodierungsschema verwendet werden.

7.8.3 Parametrisierte elementare Kodierungsschemata

Gelegentlich ist es wünschenswert, elementare Kodierungsschemata zu parametrisieren, z.B. das Schema `stratlist14_cod` aus obigem Beispiel. Ein parametrisiertes elementares Kodierungsschema hat die Signatur:

```
signature PARAM_ELEMENTARY_CODING_SCHEME =
    sig
        type parameter
        val in_cell_names: string list
        val out_cell_names: string list
        val param_coding: parameter
                          -> (CellTypes.cell_type list
                              -> CellTypes.cell_type list)
        val param_decoding: parameter
                            -> (CellTypes.cell_type list
                                -> CellTypes.cell_type list)
    end;
```

Der Funktor heißt `GetParamElementaryCodingScheme` und hat die Signatur:

```
signature GET_PARAM_ELEMENTARY_CODING_SCHEME =
    sig
        structure Coding: CODING
        type parameter
        val paramelemcodscheme:
             parameter -> Coding.elementary_coding_scheme
    end;
```

Für das Beispiel:

```
structure Stratlist_cod : PARAM_ELEMENTARY_CODING_SCHEME =
    struct
```

```
        type parameter = int
        val in_cell_names = []
        val out_cell_names = ["list"]
        fun param_coding p = fn _ => ...
        fun param_decoding p = fn _ => []
    end;

structure Stratlist: GET_PARAM_ELEMENTARY_CODING_SCHEME =
    GetParamElementaryCodingScheme(Stratlist_cod)
```

Das Kodierungsschema wird dann mit

```
structure BeliebigerName =
    struct
        val _ = Library.add_coding
            ("Name fuer diese Kodierung",
             ("Name fuer diese Kodierung",
              [(Perm2Reals.elemcodscheme, [1], [1]),
               (Stratlist.paramelemcodscheme 14, [],[2])]))
    end;
```

erzeugt.

Anhang A

Systemfunktionen

GENOM stellt eine Reihe von Komponenten zur Verfügung, die von den Bibliotheken und eigenen Erweiterungen verwendet werden können. Dazu gehören Funktionen, die von LEA aus aufgerufen werden, Hilfsfunktionen, die zum Aufbau von evolutionären Algorithmen dienen und vordefinierte Zellen und Atome, aus denen die Individuen bestehen. Im Gegensatz zu den Bibliotheken sind sie ein fester Bestandteil des Systems.

A.1 LEA-Funktionen

Die folgenden Funktionen stellen eine Erweiterung von LEA um oft verwendete Funktionen dar. Dazu gehören neben allgemeinen und mathematischen Funktionen auch solche, mit denen Variablen vom Typ `INDLIST` bearbeitet werden können und die Anbindung an die Populationsverwaltung. Zur besseren Übersicht sind zusammengehörende Funktionen in Bibliotheken gruppiert. Die folgenden Funktionen werden mit dem System mitgeliefert und können in LEA aufgerufen werden, wenn die entsprechende Bibliothek mit `USES` geladen wurde.

A.1.1 Ausgabefunktionen, Output

Die Bibliothek `Output` enthält Funktionen, mit denen LEA-Variablen zur Standardausgabe geschrieben werden können.

- `write: UNKNOWN ->`
  Gibt den Inhalt einer Variable aus. Der Typ der Variable wird dabei erst zur Laufzeit überprüft. Unterstützt werden die folgenden Typen: `INT`, `REAL`, `BOOL`, `STRING` und `IND`.

- `writeln: UNKNOWN ->`
  Wie `write` mit anschließendem Zeilenumbruch.

A.1.2 Grundlegende Funktionen, Basefct

`Basefct` enthält die Funktionen zum Umwandeln zwischen den LEA-Typen. Ferner enthält sie Funktionen zur Berechnung des Betrags und des Maximums und Minimums.

- `inttoreal: INT -> REAL`
  Wandelt eine ganze in eine reelle Zahl um.
- `floor: REAL -> INT`
  Rundet eine reelle Zahl auf die nächstniedrigere ganze Zahl.
- `absi: INT -> INT`
  Berechnet den Betrag einer ganzen Zahl.
- `absr: REAL -> REAL`
  Berechnet den Betrag einer reellen Zahl.
- `inttostr: INT -> STRING`
  Wandelt eine ganze Zahl in eine Zeichenkette um.
- `realtostr: REAL -> STRING`
  Wandelt eine reelle Zahl in eine Zeichenkette um.
- `maxi: INT * INT -> INT`
  Gibt das Maximum zweier ganzer Zahlen zurück.
- `mini: INT * INT -> INT`
  Gibt das Minimum zweier ganzer Zahlen zurück.
- `maxr: REAL * REAL -> REAL`
  Gibt das Maximum zweier reeller Zahlen zurück.
- `minr: REAL * REAL -> REAL`
  Gibt das Minimum zweier reeller Zahlen zurück.

A.1.3 Mathematische Funktionen, Math

`Math` enthält einige mathematischen Funktionen, die hauptsächlich aus der SML-Struktur `Math` stammen. Dazu kommen noch die Funktionen zum Erzeugen von Zufallszahlen.

- `sqrt: REAL -> REAL`
  Berechnet die Quadratwurzel einer reellen Zahl.
- `sin: REAL -> REAL`
  Berechnet die Sinusfunktion.
- `cos: REAL -> REAL`
  Berechnet die Cosinusfunktion.
- `tan: REAL -> REAL`
  Berechnet die Tangensfunktion.

- `arctan: REAL -> REAL`
  Berechnet den inversen Tangens.
- `exp: REAL -> REAL`
  Berechnet eine Exponentialfunktion.
- `ln: REAL -> REAL`
  Berechnet den natürlichen Logarithmus.
- `random: -> REAL`
  Gibt eine reelle Zufallszahl zwischen 0 und 1 zurück.
- `randombound: REAL * REAL -> REAL`
  Gibt eine reelle Zufallszahl aus dem angegebenen Bereich zurück.
- `randomchoose: INT -> INT`
  Wählt eine ganze Zahl aus dem Bereich von 1 bis zur angegebenen Obergrenze aus. Diese Funktion kann dazu verwendet werden, zufällig einen Index aus einer Liste auszuwählen.
- `randombool: -> BOOL`
  Erzeugt zufällig einen bool'schen Wert.
- `randomstdnorm: -> REAL`
  Gibt eine normalverteilte Zufallszahl aus $N(0,1)$ zurück.
- `randomnorm: REAL * REAL -> REAL`
  Gibt eine normalverteilte Zufallszahl mit einem bestimmten Mittelwert und Varianz aus $N(\mu,\sigma^2)$ zurück.
- `pi: -> REAL`
  Kreiskonstante π.
- `e: -> REAL`
  Natürliche Zahl e.

A.1.4 Funktionen für Listen von Individuen, IndList

Mit den Funktionen aus der Bibliothek `IndList` können Listen von Individuen, die unter LEA den Typ `INDLIST` haben, bearbeitet werden. Zusätzlich zu den bekannten Funktionen für Listen gibt es hier auch Funktionen, die die maximale, die minimale oder die durchschnittliche Fitneß der Individuen einer Liste bestimmen. Bei allen Indizes, die im folgenden verwendet werden, bezeichnet 1 das erste Element. Ein falscher Index führt zu einer Exception.

- `emptyList: -> INDLIST`
  Gibt eine leere Individuenliste zurück.
- `isempty: INDLIST -> BOOL`
  Überprüft, ob es sich um eine leere Liste handelt.
- `length: INDLIST -> INT`
  Ermittelt die Länge einer Liste.

- `head: INDLIST -> IND`
  Gibt das erste Individuum einer Liste zurück. Bei einer leeren Liste wird eine Exception erzeugt.
- `tail: INDLIST -> INDLIST`
  Gibt den Rest der Liste, ohne das erste Element, zurück.
- `getListInd: INDLIST * INT -> IND`
  Gibt das Individuum an der angegebenen Position zurück.
- `setListInd: INDLIST * INT * IND -> INDLIST`
  Ersetzt das Individuum an einer bestimmten Position durch das angegebene Individuum.
- `removeListInd: INDLIST * INT -> INDLIST`
  Entfernt das Individuum an der angegebenen Position.
- `insertListInd: INDLIST * INT * IND -> INDLIST`
  Fügt ein Individuum an einer bestimmten Position ein. Der Index gibt die Position an, an der sich das Individuum nach dem Einfügen befindet.
- `append: IND * INDLIST -> INDLIST`
  Hängt ein Individuum vorne an die Liste an.
- `merge: INDLIST * INDLIST -> INDLIST`
  Fügt zwei Individuenlisten zu einer zusammen.
- `getBestFit: INDLIST -> REAL`
  Ermittelt die beste Fitneß in einer Individuenliste. Das System ist dafür ausgelegt, das Minimum der Fitneßfunktion zu suchen. Hier wird also die minimale Fitness zurückgegeben.
- `getWorstFit: INDLIST -> REAL`
  Ermittelt die schlechteste Fitneß.
- `getAvgFit: INDLIST -> REAL`
  Ermittelt die durchschnittliche Fitneß der Individuen.

A.1.5 Populationsverwaltung, PopHandler

Die Bibliothek `PopHandler` stellt die Schnittstelle von LEA zur Populationsverwaltung dar.

- `fitness: IND -> REAL`
  Berechnet die Fitneß eines Individuums.
- `get_ind: POP * INT -> IND`
  Gibt das Individuum mit dem angegebenen Index aus der Population zurück.
- `set_ind: POP * INT * IND ->`
  Ersetzt das Individuum mit dem entsprechenden Index durch das angegebene Individuum.

- `get_Pop: POP -> INDLIST`
  Gibt die Individuen in einer Population als Liste von Individuen zurück.
- `set_Pop: POP * INDLIST ->`
  Ersetzt alle Individuen in einer Population durch die aus der Individuenliste.
- `getGenCounter: POP -> INT`
  Ermittelt die Generation der Population.
- `incGenCounter: POP ->`
  Erhöht den Generationszähler einer Population.
- `logPop: POP ->`
  Schreibt die gesamte Population in die dazugehörende Log-Datei.

A.2 SML-Funktionen

Auch für die Teile des Systems, die in SML geschrieben werden, gibt es eine Reihe von Funktionen. Einige der Funktionen, die unter LEA verwendet werden können, sind in SML-Strukturen gespeichert, die beim Start des Systems geladen werden. Zusätzlich existiert eine Exception zur einheitlichen Fehlerbehandlung.

A.2.1 Fehlerbehandlung, Error

Die Struktur `Error` enthält die Exception `runtime_error`. Diese wird im System dazu verwendet, um Fehler, die zur Laufzeit auftreten, zu signalisieren. Auf oberster Ebene sollte bei Auftreten eines Fehlers nur diese Exception ausgelöst werden.

- `runtime_error: exception of string * string * string`
  Wird verwendet, um anzuzeigen, daß im System ein Fehler erkannt wurde. Die Strings beschreiben, in welchem Modul und welcher Funktion der Fehler erkannt wurde, sowie eine Beschreibung des Fehlers.

A.2.2 Zufallszahlen, Random

In der Struktur `Random` sind die Funktionen für Zufallszahlen enthalten. Diese entsprechen den Funktionen aus der Bibliothek `Math`.

- `random: unit -> real`
  Liefert eine reelle Zufallszahl, die gleichverteilt aus dem Intervall von 0 bis 1 entnommen wird.
- `randombound: real * real -> real`
  Liefert eine Zufallszahl aus einem Intervall, dessen obere und untere Grenze angegeben sind.

- `randomchoose: int -> int`
  Wählt eine ganze Zahl aus dem Bereich von 1 bis zur angegebenen Obergrenze aus.

- `randombool: unit -> bool`
  Erzeugt zufällig einen bool'schen Wert.

- `randseed: real -> unit`
  Setzt den Startwert für den Zufallszahlengenerator. Der Startwert muß eine reelle Zahl zwischen 0 und 1 sein.

- `randomstdnorm: unit -> real`
  Erzeugt eine normalverteilte Zufallszahl mit Erwartungswert 0 und Varianz 1.

- `randomnorm: real * real -> real`
  Erzeugt eine normalverteilte Zufallszahl, wobei der Erwartungswert und die Varianz angegeben werden kann.

A.2.3 Funktionen für Individuenlisten, IndList

Einige Funktionen der Bibliothek `IndList` sind in der gleichnamigen Struktur enthalten. Andere Funktionen dieser Bibliothek lassen sich durch SML-Bibliotheksfunktionen realisieren.

- `getBestFit: PopHandler.extInd_type list -> real`
  Gibt die beste Fitneß eines Individuums der Liste zurück.

- `getWorstFit: PopHandler.extInd_type list -> real`
  Gibt die schlechteste Fitneß eines Individuums der Liste zurück.

- `getAveFit: PopHandler.extInd_type list -> real`
  Berechnet die durchschnittliche Fitneß in der Individuenliste.

- `getListInd: (PopHandler.extInd_type list * int)`
  `-> PopHandler.extInd_type`
  Liefert das Individuum an der angegebenen Position.

- `setListInd: (PopHandler.extInd_type list * int *`
  `PopHandler.extInd_type) -> PopHandler.extInd_type list`
  Ersetzt das Individuum an der angegebenen Position.

- `removeListInd: (PopHandler.extInd_type list * int)`
  `-> PopHandler.extInd_type list`
  Entfernt ein Individuum aus der Liste.

- `insertListInd: (PopHandler.extInd_type list * int *`
  `PopHandler.extInd_type) -> PopHandler.extInd_type list`
  Fügt ein Individuum in die Liste ein.

A.3 Hilfsfunktionen für ev. Algorithmen

A.3.0.1 Ziel

Die Hilfsfunktionen erlauben, einfach Funktionen auf externen Individuen (die in LEA verwendet werden) zu implementieren. Sie bieten oft gebrauchte Konstrukte an, die für eine konkrete Aufgabe nur noch an- bzw. ineinandergefügt werden müssen. Die Funktion kann durch Einbinden in eine Bibliothek für Operatoren und Verfahren zugreifbar gemacht werden.

A.3.0.2 Konzept

Für jede Ebene des Individuums (Geno-/Phänotyp, Zellen, Atome) existieren Bibliotheken mit Funktionen, die häufig gebrauchte Konstrukte zur Verfügung stellen. Die Funktionen liegen im Verzeichnis `sml/evollib`.

- `IndLib.sml`: Diese Bibliothek unterstützt das Erzeugen von zufälligen Phänotypen, die Wandlung von Phänotypen in einfache SML-Strukturen und umgekehrt. Außerdem gibt es Funktionen, die Informationen über Individuen liefern und Funktionen auf externen Individuen (die LEA verwendet).
- `<Zelltyp>CellLib.sml`: Funktionen, die Zellen verändern.
- `<Atomtyp>AtomLib.sml`: Funktionen, die Atome verändern.

A.3.0.3 Beispiel

Es soll eine einfache Zelle eines externen Individuums, die ein reelles Atom enthält, normalverteilt mutiert werden.

Aus der Bibliothek `RealAtomLib` kann die Funktion `mutate_normal` entnommen werden, die ein einzelnes Atom mutiert.

Die Bibliothek `SimpleCellLib` enthält die Funktion `apply`, die eine Funktion auf eine Zelle anwendet.

Um das externe Individuum bearbeiten zu können, braucht man die Funktion `extcellapply` aus der Bibliothek `IndLib`.

Somit läßt sich die gesamte Funktion folgendermaßen definieren:

```
fun Mutate_extern_simple_real
    (Ext_Ind, Cell_Nr, sigma, expect) =
       IndLib.extcellapply
       (SimpleCellLib.apply
        (RealAtomLib.mutate_normal expect sigma))
       (Ext_Ind, Cell_Nr)
```

A.3.0.4 Bibliotheken

Individuen

- `IndLib`: allgemeine Funktionen auf Individuen.
  - `simple_ind: (unit -> AtomTypes.atom_type) -> int -> CellTypes.cell_type`
    Hiermit kann eine Funktion erzeugt werden, die Zellen liefert, wenn sie mit einer Zahl zwischen 1 und dem Maximalwert (1. `int`-Parameter) aufgerufen wird. Alle Zellen werden mit derselben Funktion erzeugt.
  - `simple_real_ind: int * real * real -> int -> CellTypes.cell_type`
    Vereinfachung von `simple_ind`: Die reellen Werte geben den Wertebereich an, den die simple-real Zellen des Individuums haben sollen.
  - `simple_int_ind: int * int * int -> int -> CellTypes.cell_type`
    `simple_bool_ind: int -> int -> CellTypes.cell_type`
    `simple_unbound_int_ind: int -> int -> CellTypes.cell_type`
    `simple_unbound_real_ind: int -> int -> CellTypes.cell_type`
    Analog zu `simple_real_ind`.
  - `simple_reals2list: Individuum.individuum_type -> real list`
    Konvertiert ein Individuum aus simple-real Zellen in eine Liste von Reals.
  - `by_functions: (unit -> 'a) list -> int -> 'a`
    Erzeugt eine Phänostruktur, wobei eine Funktionsliste übergeben und für die entsprechende Position jeweils die Funktion aufgerufen wird.
  - `from_real_list: real list -> int -> CellTypes.cell_type`
    Erzeuge ein Listen-Zellen-Individuum aus einer Liste reeller Zahlen.
  - `from_int_list: int list -> int -> CellTypes.cell_type`
    `from_bound_real_list: real * real -> real list -> int -> CellTypes.cell_type`
    `from_bound_int_list: int * int -> int list -> int -> CellTypes.cell_type`
    `from_bool_list: bool list -> int -> CellTypes.cell_type`
    Analog zu `from_real_list`.
  - `set_cells: (UnionOfCells.cell_type * int) list -> Individuum.individuum_type -> Individuum.individuum_type`
    Setze einige Zellen in einem Individuum.
  - `set: UnionOfCells.cell_type list -> Individuum.individuum_type -> Individuum.individuum_type`
    Ersetze alle Zellen eines Individuums durch den Inhalt einer Liste.
  - `number_of_cells: Individuum.individuum_type -> int`
    Liefert die Anzahl der Zellen in einem Individuum.

- `extract_data: int list -> (UnionOfCells.cell_type -> 'a) -> Individuum.individuum_type -> 'a list`
  Wende eine Funktion auf einige Zellen eines Individuums an.
- `conv_all_cells: (UnionOfCells.cell_type -> 'a) -> Individuum.individuum_type -> 'a list`
  Wende eine Funktion auf alle Zellen eines Individuums an.
- `ind2cells: Individuum.individuum_type -> UnionOfCells.cell_type list`
  Konvertiere ein Individuum in eine Liste von Zellen.
- `simple_reals2list: Individuum.individuum_type -> real list`
  Konvertiere ein Individuum aus Simple-Zellen, die Real-Atome enthalten, in eine Liste von Reals.
- `simple_ints2list: Individuum.individuum_type -> int list`
  Konvertiere ein Individuum aus Simple-Zellen, die Int-Atome enthalten, in eine Liste von Ints.
- `pair_apply: (UnionOfCells.cell_type * UnionOfCells.cell_type -> UnionOfCells.cell_type) -> PopHandler.extInd_type * PopHandler.extInd_type -> PopHandler.extInd_type`
  Wende eine Funktion auf alle Zellen-Paare zweier externer Individuen an und liefere die resultierenden Zellen im ersten Individuum zurück.
- `pair_apply_cell: (UnionOfCells.cell_type * UnionOfCells.cell_type->'a) -> PopHandler.extInd_type * PopHandler.extInd_type * int -> 'a`
  Wende eine Funktion auf je eine Zelle zweier externer Individuen an, wobei die Zellen sich an derselben Stelle befinden.
- `extapply: (PopHandler.Ind_type -> PopHandler.Ind_type) -> PopHandler.extInd_type -> PopHandler.extInd_type`
  Wende eine Funktion auf alle Zellen eines externen Individuums an und liefere das Ergebnis als externes Individuum zurück.
- `extcellapply: (UnionOfCells.cell_type -> CellTypes.cell_type) -> PopHandler.extInd_type * int -> PopHandler.extInd_type`
  Wende eine Funktion auf eine Zelle eines externen Individuums an und trage das Ergebnis in das Individuum ein.
- `extcellapplyall: (UnionOfCells.cell_type -> UnionOfCells.cell_type) -> PopHandler.extInd_type -> PopHandler.extInd_type`
  Wende eine Funktion auf alle Zellen eines externen Individuums an und gebe es verändert zurück.
- `extgetcell: PopHandler.extInd_type * int -> UnionOfCells.cell_type`
  Lese eine Zelle in einem externen Individuum aus.
- `extsetcell: PopHandler.extInd_type * UnionOfCells.cell_type * int -> PopHandler.extInd_type`
  Setze eine Zelle in einem externen Individuum.

  - `extconv: (PopHandler.Ind_type -> 'a) -> PopHandler.extInd_type -> 'a`
    Wende eine beliebige Funktion auf die Zellen eines externen Individuums an.

Zellen

- `CellLib`: Funktionen, die auf allen zur Zeit implementierten Zellen definiert sind. Diese Bibliothek muß nach Erweiterung des Systems um neue Zellentypen erweitert werden.
  - `first: CellTypes.cell_type -> AtomTypes.atom_type`
    Ermittelt das erste Atom in einer Zelle.
  - `last: CellTypes.cell_type -> AtomTypes.atom_type`
    Ermittelt das letzte Atom in einer Zelle.
- `SimpleCellLib`: Funktionen auf Simple-Zellen.
  - `apply: (AtomTypes.atom_type -> AtomTypes.atom_type) -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Funktion auf das Atom in der Zelle an.
- `PairCellLib`: Funktionen auf Paar-Zellen. Diese Zellen werden u.a. als Parameter mit zugehörigem Strategie-Parameter verwandt.
  - `apply: (AtomTypes.atom_type -> AtomTypes.atom_type) (AtomTypes.atom_type -> AtomTypes.atom_type) -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende je eine Funktion auf die beiden Atome an.
  - `mutate_real_normal: CellTypes.cell_type * int * real * real -> CellTypes.cell_type`
    Mutiere ein Real-Atom der Zelle normalverteilt, wobei der erste reelle Parameter der Erwartungwert, der zweite die Standardabweichung angibt. Zudem kann die Nummer des Atoms in der Zelle angegeben werden.
  - `get_real_strat: CellTypes.cell_type -> real`
    Ermittelt den zweiten Wert der Zelle, der oft als Strategiewert interpretiert wird. Diese Funktion arbeitet nur auf Real-Atomen.
  - `mutate_strat_real: real -> real -> int -> Individuum.individuum_type -> Individuum.individuum_type`
    Mutiere den Strategiewert in einer Paar-Zelle, sofern sie ein Real-Atom enthält.
  - `mutate_prob': CellTypes.cell_type -> CellTypes.cell_type`
    Mutiere den Problemwert mit Hilfe des Strategiewerts, wenn beide Atome reell sind.
  - `cross_pair: CellTypes.cell_type * CellTypes.cell_type * real -> CellTypes.cell_type`
    Mutiere zwei Paar-Zellen intermediär.

- `ListCellLib`: Funktionen auf Listen-Zellen.
  - `apply: (AtomTypes.atom_type -> AtomTypes.atom_type) -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Funktion auf alle Zellen an.
  - `applyL: (AtomTypes.atom_type list -> AtomTypes.atom_type list) -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Funktion auf die gesamte Liste an.
  - `apply_n: (AtomTypes.atom_type -> AtomTypes.atom_type) -> ListCell.index -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Funktion auf das n-te Element der Liste an.
  - `apply_list: (AtomTypes.atom_type ->AtomTypes.atom_type) list -> CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Liste von Funktionen auf die Liste an. Dabei wird bei Erreichen des Endes der Funktionenliste wieder die erste Funktion verwendet.
  - `pair_apply: (AtomTypes.atom_type * AtomTypes.atom_type -> AtomTypes.atom_type) -> CellTypes.cell_type * CellTypes.cell_type -> CellTypes.cell_type`
    Wende eine Funktion paarweise auf die Elemente zweier Listen an und liefere eine Liste zurück.
  - `list2cell: ('a -> AtomTypes.atom_type list) -> 'a -> CellTypes.cell_type`
    Wandle eine Liste in eine Zelle.
  - `reallist2cell: real list -> CellTypes.cell_type`
    Konvertiere eine Liste von Reals in eine Listenzelle.
  - `boundreallist2cell: real * real -> real list -> CellTypes.cell_type`
    Konvertiere eine Liste von Reals in eine Listenzelle von begrenzten Reals.
  - `intlist2cell: int list -> CellTypes.cell_type`
    analog zu `reallist2cell`.
  - `boundintlist2cell: int * int -> int list -> CellTypes.cell_type`
    analog zu `boundreallist2cell`.
  - `boollist2cell: bool list -> CellTypes.cell_type`
    analog zu `boollist2cell`.
  - `cell2list: (AtomTypes.atom_type list -> 'a) -> CellTypes.cell_type -> 'a`
    Wende eine Funktion auf den Inhalt einer Listenzelle an.
  - `cell2reallist: CellTypes.cell_type -> real list`
    Konvertiere eine Listenzelle aus Real-Atomen in eine Liste von Reals.
  - `cell2intlist: CellTypes.cell_type -> int list`
    `cell2boollist: CellTypes.cell_type -> bool list`
    analog zu `cell2reallist`.

  - `mutate_reverse: 'a list -> 'a list`
    Mutiert eine beliebige Liste durch Umkehren eines zufällig bestimmten Teils.
  - `mutate_rotate: 'a list -> 'a list`
    Mutiert eine beliebige Liste durch Rotation um eine zufällige Anzahl von Stellen.
  - `mutate_exchange: 'a list -> 'a list`
    Tauscht zwei zufällig bestimmte Elemente einer beliebigen Liste aus.

Atome

- `AtomLib`: Funktionen, die verschiedene Atome bearbeiten können.
  - `identity: 'a -> 'a`
    Funktion, die ein Atom nicht verändert.
    (Verwendbar z.B. in `PairCellLib.apply`.)
  - `a2real: real -> AtomTypes.atom_type -> real`
    Wandle ein Atom in einen reellen Wert, wobei bei Übergabe eines Atoms, das keinen Real-Wert darstellt, ein Default-Wert zurückgegeben wird.
  - `a2int: int -> AtomTypes.atom_type -> int`
    Analog zu `a2real` für Integer.
- `RealAtomLib`: Funktionen auf reellen Atomen, wobei nicht zwischen begrenzten und unbegrenzten unterschieden wird.
  - `set: AtomTypes.atom_type * real -> AtomTypes.atom_type`
    Setze den Wert eines Real-Atoms, wobei ggf. die Grenzen beachtet werden.
  - `gen_bound: real * real * real -> AtomTypes.atom_type`
    Erzeuge ein begrenztes reelles Atom, wobei die Grenzen überprüft werden.
  - `mutate_normal: real -> real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Atom normalverteilt.
  - `mutate_normal0: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Atom normalverteilt, mit Erwartungswert 0.
  - `set_normal: real -> real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Setze ein Atom auf einen normalverteilt ermittelten Wert.
  - `set_normal0: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Setze ein Atom auf einen normalverteilt ermittelten Wert, wobei der Erwartungswert 0 ist.

  - `mutate_uniform_interval: real -> real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Atom, wobei ein gleichverteilter Zufallswert aus einem Intervall auf den alten Wert addiert wird.
  - `mutate_uniform: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Atom, wobei ein gleichverteilter Zufallswert aus einem Intervall um 0 aufaddiert wird.
  - `mutate_uniform_onesided: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Atom; der aufaddierte Wert stammt aus einem Intervall zwischen 0 und einem Parameter.
  - `set_uniform_interval: real -> real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    `set_uniform: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Setzt den Wert eines Atoms, analog zu `mutate_ ...`.
  - `dyadic: (real * real -> real) -> AtomTypes.atom_type * AtomTypes.atom_type -> AtomTypes.atom_type`
    Wende eine Funktion, die zwei reelle Zahlen erwartet, auf zwei reelle Atome an und liefere das Ergebnis als Atom zurück. Nur wenn beide Atome unbegrenzt sind, wird ein unbegrenztes Atom geliefert.

- `IntAtomLib`: Funktionen auf Integer-Atomen, die ebenfalls keine Unterscheidung zwischen begrenzten und unbegrenzten machen.
  - Funktionen analog zu `RealAtomLib`.

- `BoolAtomLib`:
  Funktionen auf Bit-Atomen.
  - `mutate: real -> AtomTypes.atom_type -> AtomTypes.atom_type`
    Mutiere ein Bit mit einer gewissen Wahrscheinlichkeit.

A.3.0.5 LEA-Bibliotheken

Einige Funktionen für Permutationen sind auch direkt in LEA verfügbar.

A.3.0.6 Permutationen

`MutatePerm` enthält Funktionen, um Permutationen zu mutieren.[1]

- `setvalue: IND -> IND`
  Wählt zufällig zwei Atome der Permutation aus und verschiebt alle Atome dazwischen um eine Position nach links oder rechts. Eine genaue Beschreibung findet sich in [JW95].

[1] Tatsächlich enthält die Bibliothek nur eine Schnittstelle zu SML-Funktionen aus der Bibliothek `PermCellLib`, die beim Systemstart geladen wird.

- `xchange: IND -> IND`
  Tauscht zwei zufällig ausgewählte Atome gegeneinander aus. Auch diese Funktion ist in [JW95] beschrieben.

- `lin2: IND -> IND`
  Wählt zufällig zwei Atome der Permutation aus und kehrt die Reihenfolge der dazwischenliegenden Atome um.

A.4 Elementare Kodierungsschemata

Die elementaren Kodierungsschemata sind Grundbausteine der Kodierungen, können selbst aber von LEA aus nicht angesprochen werden.

- `Ident`
  - Eingabe: alle Zelltypen
  - Ausgabe: alle Zelltypen

  Identische Kodierung.

- `Perm2Ints`
  - Eingabe: eine Zelle, die eine Permutation aus natürlichen Zahlen zwischen 1 und der Länge der Permutation enthält
  - Ausgabe: eine Zelle, die eine Liste aus ganzen Zahlen enthält

  Die Permutation wird umgekehrt eindeutig in eine Liste gleicher Länge ganzer Zahlen kodiert. Umgekehrt läßt sich jede Liste ganzer Zahlen in eine Permutation gleicher Länge überführen. Die erzeugte Permutation enthält nur Zahlen aus dem Bereich zwischen 1 und der Länge der Permutation.

- `Perm2Bits`
  - Eingabe: eine Zelle, die eine Permutation aus natürlichen Zahlen zwischen 1 und der Länge der Permutation enthält
  - Ausgabe: eine Zelle, die eine Liste aus boolschen Werten enthält
  - Parameter: natürliche Zahl

  Diese Kodierung baut auf der vorherigen auf, jede Zahl der Ausgabeliste wird binär kodiert und die entstehenden Bitlisten konkateniert. Der Parameter bestimmt, wieviele Bits zur Kodierung jeder einzelnen Zahl verwendet werden. Er muß ausreichend groß sein, um die Länge der Permutation binär darstellen zu können (es wird standardbinär kodiert).

- `Perm2Gray`
  - Eingabe: eine Zelle, die eine Permutation aus natürlichen Zahlen zwischen 1 und der Länge der Permutation enthält
  - Ausgabe: eine Zelle, die eine Liste aus boolschen Werten enthält
  - Parameter: natürliche Zahl

Entspricht `Perm2Bits`, verwendet aber keine Binär-, sondern eine Gray-kodierung.

- `Int2Bits`
  - Eingabe: `SimpleCell`, die ein Integer-Atom enthält. Der Wert des Atoms muß ≥ 0 sein.
  - Ausgabe: Listenzelle aus Bool-Atomen

  Der Integer-Wert wird standardbinär kodiert. Die Anzahl der Bits in der Liste ist abhängig von der Größe der Zahl.

- `Int2Gray`
  - Eingabe: `SimpleCell`, die ein Integer-Atom enthält. Der Wert des Atoms muß ≥ 0 sein.
  - Ausgabe: Listenzelle aus Bool-Atomen

  Der Integer-Wert wird gray-kodiert, sonst wie oben.

- `RealList2FixedBits`
  - Eingabe: `BoundReal` Atome in einer `ListCell`
  - Ausgabe: boolsche Atome in einer `ListCell`
  - Parameter: Paar bestehend aus dem Parameter `interval` des `BoundReal` Atoms und einer natürlichen Zahl

  Die reellen Zahlen werden einzeln standardbinär kodiert und die entstehenden Bitlisten konkateniert. Die erste Komponente des Parameters gibt den Wertebereich der reellen Atome an, die zweite, wieviele Bits zur Darstellung einer einzelnen kodierten Zahl verwendet werden.

A.5 Zellen

Individuen (d.h. sowohl der Geno- als auch der Phänotyp) sind Listen fester Länge von Zellen. Zellen enthalten ihrerseits Atome, die in einer vom Zelltyp abhängigen Struktur angeordnet sind. Derzeit sind vier verschiedene Zelltypen implementiert: einfache Zellen, Zellen mit einem Paar von Atomen, Zellen mit Listen von Atomen und Zellen, die eine Permutation enthalten. Alle Zellen haben folgende Signatur (Erläuterungen dazu siehe Abschnitt 5.1.1 Seite 38):

```
signature CELL =
  sig
    type 'a rawstructure;
    type index;
    val name: string;
    val init: ((index -> AtomTypes.atom_type) * index list)
                      -> AtomTypes.atom_type rawstructure;
    val get_element: (AtomTypes.atom_type rawstructure * index)
                           -> AtomTypes.atom_type;
```

```
    val set_element: (AtomTypes.atom_type rawstructure *
                      AtomTypes.atom_type * index)
                          -> AtomTypes.atom_type rawstructure;
    val cell2rawstructure: CellTypes.cell_type
                          -> AtomTypes.atom_type rawstructure;
    val rawstructure2cell: AtomTypes.atom_type rawstructure
                          -> CellTypes.cell_type;
    val cell2string: AtomTypes.atom_type rawstructure
                                                  -> string;
    val string2cell: string
                          -> AtomTypes.atom_type rawstructure;
  end;
```

A.5.1 einfache Zellen

```
structure SimpleCell: CELL =
  struct
    type 'a rawstructure = 'a;
    type index = int;
    val name = "SimpleCell";
    ...
end;
```

- `init (f, il)` nimmt den ersten Wert `x` aus der Liste und liefert `f(x)` zurück.
- Sowohl `get_element` als auch `set_element` ignorieren das Argument vom Typ `index`, da bei einzelnen Elementen eine Indizierung keinen Sinn macht.
- `cell2string` liefert das in eine Zeichenkette umgewandelte atom zurück; `string2cell` kehrt diese Operation um.
- `rawstructure2cell x = CellTypes.simple_cell(x)`
- `cell2rawstructure CellTypes.simple_cell(x) = x`; bei anderen Argumenten wird eine Fehlermeldung erzeugt.

A.5.2 Zellen mit Paaren von Atomen

```
structure PairCell: CELL =
  struct
    type 'a rawstructure = 'a * 'a;
    type index = int;
    val name = "PairCell";
    ...
end;
```

- `init (f, [x, y, ...])` liefert das Paar `(f(x), f(y))` zurück.

- In Paaren hat das linke Element den Index 1, das rechte den Index 2. Dies gilt sowohl für `get_element` als auch `set_element`.
- `cell2string (x, y) = (`$\overline{\texttt{x}}$`, `$\overline{\texttt{y}}$`)`, wobei $\overline{\texttt{x}}$ und $\overline{\texttt{y}}$ die Darstellung der Atome `x` und `y` als Zeichenkette sind; `string2cell` kehrt diese Operation um.
- `rawstructure2cell(x, y) = CellTypes.pair_cell(x, y)`
- `cell2rawstructure (CellTypes.pair_cell(x, y)) = (x, y)`; bei anderen Argumenten wird eine Fehlermeldung erzeugt.

A.5.3 Zellen mit Listen von Atomen

```
structure ListCell: CELL =
  struct
    type 'a rawstructure = 'a list;
    type index = int;
    val name = "ListCell";
    ...
end;
```

- `init (f, [i1, .., in]) = [f(i1), .., f(in)]`
- `get_element(l, i)` liefert das `i`-te Element aus der Liste `i`; dabei hat das erste Element der Liste den Index 1.
- `set_element(l, x, i)` ersetzt in der Liste `l` das Element an der `i`-ten Position durch `x`.
- `cell2string` kodiert die Atome der Liste als Zeichenketten und trennt diese durch ein Komma; `string2cell` kehrt diese Operation um.
- `rawstructure2cell x = CellTypes.list_cell x`
- `cell2rawstructure (CellTypes.list_cell x) = x`; bei anderen Argumenten wird eine Fehlermeldung erzeugt.

A.6 Atome

`GENOM` stellt zwei verschiedene Klassen von Atomen zur Verfügung: Atome ohne Parameter und Atome, die zusätzlich noch ein Intervall enthalten. Jede Klasse von Atomen hat eine eigene Signatur. Atome ohne Parameter haben folgende Signatur:

```
signature ATOM =
    sig
        type base; (* Typ des Atoms *)
        val name: string; (* Bezeichner fuer diesen Atomtyp *)
        val base2atom: base -> AtomTypes.atom_type;
```

```
                    (* nach Vereinigungstyp *)
      val atom2base: AtomTypes.atom_type -> base;
                    (* von Vereinigungstyp *)
      val base2string: base -> string; (* nach string *)
      val string2base: string -> base; (* von string *)
      val init_random: real -> base; (* Zufallswert *)
  end;
```

Von dieser Signatur gibt es drei Strukturen (reelle Zahlen, ganze Zahlen und bool'sche Werte als Atome). Zu beachten ist, daß die reelle Zahl, die `init_random` als Argument erfordert, nicht verwendet wird:

1. ```
   structure RealAtom: ATOM =
     struct
       type base = real;
       val name = "RealAtom";
       ...
     end
   ```

   - `base2string` und `string2base` verwenden die SML-Funktionen `makestring` bzw. `Real.fromString` zur Umwandlung.
   - `init_random` ruft zweimal die Hilfsfunktion `Random.random` auf. Da mit dieser Zufallsfunktion nur Zahlen zwischen 0 und 1 erhalten werden können und nicht aus ganz $\mathbb{R}$, wird von der ersten Zufallszahl 0.5 abgezogen und das Ergebnis durch die zweite Zufallszahl geteilt. Das Argument vom Typ `real` wird nicht verwendet.
   - `base2atom x = AtomTypes.real_atom(x)`
   - `atom2base y` liefert `x` zurück, falls `y = AtomTypes.real_atom(x)`; andernfalls wird eine Fehlermeldung erzeugt.

2. ```
   structure IntegerAtom: ATOM =
     struct
       type base = int;
       val name = "IntegerAtom";
       ...
     end
   ```

   - `base2string` und `string2base` verwenden die SML-Funktionen `makestring` bzw. `Int.fromString` zur Umwandlung.
   - `init_random` ermittelt ähnlich wie in der Struktur `RealAtom` eine zufällige reelle Zahl und liefert davon die nächstgrößere ganze Zahl zurück. Das Argument vom Typ `real` wird nicht verwendet.
   - `base2atom x = AtomTypes.int_atom(x)`
   - `atom2base y` liefert `x` zurück, falls `y = AtomTypes.int_atom(x)`; andernfalls wird eine Fehlermeldung erzeugt.

3. ```
   structure BooleanAtom: ATOM =
     struct
   ```
   ```

```
  type base = bool;
  val name = "BooleanAtom";
  ...
end
```

- `base2string` und `string2base` verwenden die SML-Funktionen `makestring` bzw. `Bool.fromString` zur Umwandlung.
- `init_random` ruft die Hilfsfunktion `Random.random` auf und liefert `true` zurück, falls der Wert größer als 0.5 ist. Andernfalls wird `false` zurückgeliefert.
- `base2atom x = AtomTypes.bool_atom(x)`
- `atom2base y` liefert `x` zurück, falls `y = AtomTypes.bool_atom(x)`; andernfalls wird eine Fehlermeldung erzeugt.

Atome, deren Wert durch ein Intervall beschränkt ist, haben folgende Signatur:

```
signature INTERVALATOM =
    sig
        type base;
        type interval;
        val name: string;
        val base2atom: base -> AtomTypes.atom_type;
        val atom2base: AtomTypes.atom_type -> base;
        val base2string: base -> string;
        val string2base: string -> base;
        val init_random: interval -> real -> base;
    end;
```

Der Unterschied zu Atomen ohne Parameter besteht darin, daß zusätzlich ein Datentyp Intervall definiert wird und die Funktion `init_random` als ersten Parameter ein Intervall erfordert. Dahinter steckt die Idee, daß `init_random` nur einen Wert innerhalb des Intervalls liefert. Ob dies tatsächlich der Fall ist, hängt natürlich von der jeweiligen Struktur ab. Im Augenblick sind die folgenden zwei Strukturen implementiert:

1. ```
   structure BoundRealAtom: INTERVALATOM =
     struct
       type interval = real * real;
       type base = real * interval;
       val name = "BoundRealAtom";
       ...
     end
   ```

   - `atom2base` und `base2atom` arbeiten ähnlich wie bei `RealAtom`.
   - `base2string(x,(min, max))` liefert „$\overline{\texttt{x}}$ `in` $(\overline{\texttt{min}}, \overline{\texttt{max}})$“, dekodiert wird diese Zeichenkette mit `string2base`. $\overline{\texttt{x}}$, $\overline{\texttt{min}}$ und $\overline{\texttt{max}}$ erhält man aus `x`, `min` und `max` mit der SML-Funktion `makestring`.
   ```

- `init_random (min, max) seed` liefert eine zufällige reelle Zahl zwischen `min` und `max`.

2. `structure BoundIntegerAtom: INTERVALATOM =`

```
struct
  type interval = int * int;
  type base = int * interval;
  val name = "BoundIntegerAtom";
  ...
end
```

   - `atom2base` und `base2atom` arbeiten ähnlich wie bei `IntegerAtom`.
   - `base2string(x,(min, max))` liefert „$\overline{\texttt{x}}$ `in` $(\overline{\texttt{min}}, \overline{\texttt{max}})$“, dekodiert wird diese Zeichenkette mit `string2base`. $\overline{\texttt{x}}, \overline{\texttt{min}}$ und $\overline{\texttt{max}}$ erhält man aus `x`, `min` und `max` mit der SML-Funktion `makestring`.
   - `init_random (min, max) seed` liefert eine zufällige ganze Zahl zwischen `min` und `max`.

Anhang B

Bibliotheken

In GENOM werden verschiedene Arten von Bibliotheken unterstützt, durch die das System erweitert werden kann. Sie werden durch vorgegebene Mechanismen an das System angebunden und sind kein eigentlicher Teil davon. Im weiteren werden die Bibliotheken beschrieben, die mit dem System mitgeliefert werden. Sie können entweder als Teil von eigenen Erweiterungen verwendet werden oder als Vorlage für selbstgeschriebene Komponenten dienen.

B.1 Experimente

Die Experimentdefinitionen beinhalten alle Einstellungen, die zur Anwendung eines Verfahrens (oder mehrerer) auf ein Problem nötig sind. Sie können daher direkt aufgerufen werden und stellen die einfachste Möglichkeit dar, einen ersten Eindruck vom System zu gewinnen.

- `TestEvolStrat`
  Benutzt das Rastingin–Problem und wendet eine Evolutionsstrategie darauf an. Die einzelnen Generationen werden in die Datei `EvolStrat` geschrieben.

- `TestGenAlg`
  Führt einen Genetischen Algorithmus auf dem Hyperspähren-Problem durch. Dabei wird die Log-Datei `GenAlg.log` erzeugt.

B.2 Verfahren

Hier werden drei bekannte evolutionäre Verfahren bereitgestellt. Sie können in eigenen Experimenten und für selbstgeschriebene Probleme verwendet werden. Da sie eine bestimmte Form der Genostruktur voraussetzen, muß man allerdings meist eine passende Kodierung benützen.

B.2.1 Evolutionsstrategien

Evolutionsstrategien wurden durch zwei Operatoren ins System integriert:

- `EvolStrat: Pop -> ()`
  Eine Population mit Individuen, die aus reellen Paar-Zellen bestehen, wird durch eine Evolutionsstrategie optimiert. Parameter sind:
  - `mue: INT`; Anzahl der Nachkommen pro Generation.
  - `plus: BOOL`; Soll die „Plus-Strategie“ angewandt werden? Ist dieser Wert auf `FALSE`, dann wird die nächste Generation nur aus den Nachkommen ausgewählt („Komma-Strategie“).
- `EvolStrat1: Pop -> ()`
  Dieser Operator ist formal identisch mit `EvolStrat`, jedoch kann die Rekombination der Strategie- und der Problem-Parameter getrennt eingestellt werden.

Diese Algorithmen verwenden zwei besondere Operatoren für folgende Aufgaben:

- Rekombination: Aus der Elterngeneration werden Nachkommen erzeugt, indem man für jede Position reelle Werte ermittelt.
- Mutation: Die reellen Werte eines Individuums werden verändert.

Für die Rekombination stehen zwei Operatoren zur Verfügung:

- `Recombination: INDLIST * INT -> INDLIST`
  Erzeuge eine Anzahl neuer Individuen aus der Elternpopulation durch Rekombination. Diese kann durch Parameter gesteuert werden:
  - `chi: REAL`; Parameter für intermediäre Rekombination
  - `do_recomb: BOOL`; Soll überhaupt rekombiniert oder einfach ein Individuum zufällig gewählt werden?
  - `discreteF: BOOL`; Soll der neue Wert direkt von einem Elternindividuum übernommen werden? Ist dieser Wert auf `FALSE`, dann wird die intermediäre Rekombination verwendet.
  - `globalF: BOOL`; Sollen die Eltern-Individuen für jeden Wert neu aus der Population ausgewählt werden?
- `LEA_Recomb: INDLIST * IND * IND -> IND`
  Dieser Operator erzeugt durch Rekombination ein neues Individuum. Dabei werden im lokalen Fall die beiden übergebenen Individuen verwendet, sonst wird aus der Population ausgewählt. Es gibt dieselben Parameter wie oben, zusätzlich jedoch:
  - `atomnr: INT`; Position der zu rekombinierenden Atome in den Zellen.

Die Mutation kann durch folgenden Operator erreicht werden:

- `ESMutate: INDLIST -> INDLIST`
  Alle Individuen der Liste werden mutiert, indem zuerst der Strategie-Wert mit der Meta-Schrittweite mutiert wird, danach der Problem-Wert unter Verwendung des Strategie-Werts. Parameter ist:
  - `meta: REAL`; Meta-Schrittweite zur Mutation des Strategie-Werts.

Folgende Bibliotheken dienen zur Ausführung einzelner Operationen auf den Individuen:

- `es_mutate_pair`
  - `mutate_strat: IND * REAL -> IND`
    Mutiere alle Strategie-Parameter (d.h. den 2. Wert der Paar-Zelle) in einem Individuum, wobei der Parameter die Standardabweichung darstellt.
  - `mutate_prob: IND -> IND`
    Mutiere die Problem-Parameter eines Individuums (d.h. den ersten), wobei der Strategie-Wert steuert.
- `es_recomb_pair`
  - `no_recomb: INDLIST -> IND`
    Wähle ein Individuum zufällig aus.
  - `discrete: IND * IND -> IND`
    Rekombiniere zwei Individuen durch direkte Entnahme der Werte aus dem einen oder dem anderen Individuum.
  - `intermed: IND * IND * REAL -> IND`
    Diskrete Rekombination der beiden Individuen, wobei der Parameter den Ort des erzeugten neuen Werts, im Intervall der alten, festlegt.
  - `global_discrete: INDLIST -> IND`
    Erzeuge ein neues Individuum, indem für jeden Wert ein Individuum der Liste ausgewählt wird und dessen Wert an der fraglichen Stelle übernommen wird.
  - `global_intermed: INDLIST * REAL -> IND`
    Erzeuge ein neues Individuum, indem jeweils zwei Individuen der Liste ausgewählt werden, deren Werte zur Berechnung des neuen Werts verwendet werden. (Parameter analog zu `intermed`.)

B.2.2 Genetischer Algorithmus

Das Verfahren `GenAlg` realisiert einen Genetischen Algorithmus. Als Genostruktur wird eine Zelle mit einer Liste von Bits erwartet. Das Verfahren hat die beiden Parameter:

- `mue: INT`
  Mit diesem Parameter wird die Zahl der Nachkommen, die durch den Crossover erzeugt werden, angegeben.

- `writelog: BOOL`
  Gibt an, ob jede Generation in die Log-Datei geschrieben werden soll. Sonst wird überhaupt nicht in die Log-Datei geschrieben.

Der Algorithmus verwendet die Unteroperatoren `GAMutate` für die Mutation und `Crossover` für die Rekombination. Mit Hilfe des Operators `ElitistPropSelect` werden die Individuen ausgewählt, die in die nächste Generation übernommen werden.

- `GAMutate: INDLIST -> INDLIST`
  Mutiert die Bits der Individuen mit einer gewissen Wahrscheinlichkeit. Die Wahrscheinlichkeit der Mutation wird durch den Parameter `Prob` gesteuert.

  - `Prob: REAL`
    Die Wahrscheinlichkeit der Mutation eines Bits.

- `Crossover: INDLIST * INT -> INDLIST`
  Erzeugt durch einen Crossover aus einer Individuenliste die angegebene Zahl von Nachkommen.

  - `Points: INT`
    Die Anzahl der Stellen, an denen die Bitstrings gekreuzt werden.
  - `nue: REAL`
    Die Wahrscheinlichkeit, daß für zwei ausgewählte Individuen ein Crossover ausgeführt wird. Andernfalls wird zufällig einer der beiden Eltern übernommen.

B.3 Operatoren

Für oft verwendete Vorgehensweisen der Evolutionären Algorithmen gibt es eine Reihe von vorgefertigten Operatoren. Sie werden auch zum Teil in den oben beschriebenen Verfahren verwendet.

B.3.1 Selektionsoperatoren

Selektionsoperatoren dienen dazu, aus einer Individuenliste anhand der Fitneß eine Anzahl von Individuen auszuwählen.

- `BestSelect: INDLIST * INT -> INDLIST`
  Wählt die gewünschte Anzahl von Individuen mit den besten Fitneßwerten aus der Liste aus.

- `PropSelect: INDLIST * INT -> INDLIST`
  Wählt die Individuen proportional zu ihrer Fitneß aus. Die Individuen mit den besten Fitneßwerten werden dabei mit höherer Wahrscheinlichkeit ausgewählt, als die mit niedrigerer Fitneß. Der Operator hat den Parameter

  - `WorstIndFact: REAL`
    Dieser gibt an, wie stark die Fitneß des schlechtesten Individuums gegenüber der des besten Individuums gewichtet werden soll.

- `ElitistPropSelect: INDLIST * INT -> INDLIST`
  Entspricht `PropSelect`, nur daß das Individuum mit der besten Fitneß auf jeden Fall ausgewählt wird.

B.3.2 Abbruchbedingungen

Operatoren für Abbruchbedingungen liefern einen bool'schen Wert, der wahr wird, wenn die Population eine bestimmte Bedingung erfüllt.

- `CntGenStopCond: POP -> BOOL`
  Abbrechen, nachdem eine bestimmte Anzahl von Generationen berechnet wurden. Der Operator hat den folgenden Parameter:

  - `Generations: INT`
    Die Zahl der gewünschten Generationen.

- `IndDiffStopCond: POP -> BOOL`
  Dieser Operator wird wahr, wenn die maximale Differenz der Fitneßwerte unter einen bestimmten Wert sinkt.

  - `Difference: REAL`
    Die Differenz der Fitneß.

- `ResultStopCond: POP -> BOOL`
  Bricht ab, wenn eine gewünschte Fitneß erreicht wird. Die Parameter des Operators sind:

  - `BestFitness: REAL`
    Die beste bekannte Fitneß für das Problem.
  - `Factor: REAL`
    Gibt an, wie genau diese Fitneß erreicht werden soll.

B.4 Probleme

Einige mathematische Testfunktionen befinden sich im Verzeichnis `lib/problems` des Systems. Sie können verwendet werden, um das Verhalten eigener Verfahren zu bestimmen.

B.4.1 Mathematische Funktionen

Hierbei handelt es sich um die Umsetzung einiger bekannter Testfunktionen. Die Probleme arbeiten alle mit einer Phänostruktur, die aus Zellen von reellen Zahlen besteht.

- `Schwefel1`, Schwefelfunktion.

$$F(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right) ;\ Min(F(\vec{x})) = F(\vec{0})$$

- `SumDiffPow`, Summe verschiedener Potenzen.

$$F(\vec{x}) = \sum_{i=1}^{n} |x_i|^{i+1} ;\ Min(F(\vec{x})) = F(\vec{0})$$

- `Hyperellipse`, Achsenparallele Hyperellipsoide.

$$F(\vec{x}) = \sum_{i=1}^{n} (i \cdot x_i)^2 ;\ Min(F(\vec{x})) = F(\vec{0})$$

- `Hypersphere`, Hypersphäre.

$$F(\vec{x}) = \sum_{i=1}^{n} x_i^2;\ Min(F(\vec{x})) = F(\vec{0})$$

- `Griewank` Griewank's Funktion.

$$F(\vec{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1;\ Min(F(\vec{x})) = F(\vec{0})$$

- `Rastingin`, Rastingin Funktion.

$$F(\vec{x}) = 3.0 \cdot n + \sum_{i=1}^{n} x_i^2 - 3.0 \cdot \cos(2 \cdot \pi \cdot x_i);\ Min(F(\vec{x})) = F(\vec{0})$$

- `Schwefel2`, Weitere Schwefelfunktion.

$$F(\vec{x}) = 418.9829 \cdot n - \sum_{i=1}^{n} x_i \cdot \sin(\sqrt{|x_i|});$$
$$Min(F(\vec{x})) = F(420.9687, 420.9687, \ldots)$$

- `DeJong3`, De Jong's 3. Testfunktion.

$$F(\vec{x}) = real \sum_{i=1}^{n} integer(x_i)$$

B.5 Kodierungen

In GENOM wird bei den Individuen zwischen einer Darstellung als Genotyp und einer als Phänotyp unterschieden. Die Umwandlung von einem in den anderen Typ wird durch eine Kodierung bewerkstelligt. Für jede Population muß angegeben werden, welche Kodierung verwendet werden soll. Konkret setzen sich diese aus den elementaren Kodierungsschemata zusammen. Mit GENOM werden einige vorgefertigte Kodierungen mitgeliefert.

- `EvolStratPair`
  Die Kodierung `EvolStratPair` wandelt ein mathematisches Problem in eine Form um, die von der Evolutionsstrategie des System verwendet wird. Dazu wird aus jeder Real-Zelle eine Zelle mit einem Paar von Realatomen. Das erste davon beinhaltet den Wert der Zelle und das zweite den dazugehörenden Strategieparameter.
  - Phänostruktur: Besteht aus Real-Zellen.
  - Genostruktur: Für jede Real-Zelle der Phänostruktur eine Paar-Zelle.
- `GenAlgGrayCod`
  Diese Kodierung wird verwendet um mathematische Probleme, die aus Real-Zellen bestehen, für die Verwendung mit einem Genetischen Algorithmus in einen Bitstring umzuwandeln.
  - Phänostruktur: Besteht aus Real-Zellen.
  - Genostruktur: Eine Zelle, die eine Liste mit den erzeugten Bits enthält.
- `Identity`
  Diese einfachste Kodierung übernimmt alle Zellen des Problems in die Genostruktur. Sie wird verwendet, wenn ein Algorithmus direkt auf der Struktur des Problems arbeiten kann.
  - Phänostruktur: beliebig.
  - Genostruktur: entspricht der Phänostruktur.
- `Perm2Bits`
  Um einen Genetischen Algorithmus für ein Problem zu verwenden, das eine Permutation enthält, wird daraus eine Liste mit Bit-Atomen erzeugt. Die Kodierung verwendet den Wert `Problem.cities`, um zu ermitteln, wieviele Bits zur Darstellung der kodierten Permutation benötigt werden.
  - Phänostruktur: Eine Zelle mit einer Permutation.
  - Genostruktur: Eine Zelle mit einer Liste aus Bit-Atomen.
- `Perm2Gray`
  Wie `Perm2Bits`, nur daß hier die Bits eine Gray-Kodierung der Elemente der Permutation darstellen.
  - Phänostruktur: Eine Zelle mit einer Permutation.
  - Genostruktur: Eine Zelle mit einer Liste aus Bit-Atomen.

- `Perm2Ints`
  Wandelt eine Problem, das aus einer Permutation besteht in eine Genostruktur mit einer Liste aus Ints um.
  - Phänostruktur: Eine Zelle mit einer Permutation.
  - Genostruktur: Eine Zelle mit einer Liste aus Integer-Atomen.
- `RealList2FixedBits`
  Wie `GenAlgGrayCod`, nur hat hier das Problem eine leicht andere Struktur. Zusätzlich wird keine Gray-Kodierung, sondern eine dezimale Kodierung der reellen Zahlen verwendet.
  - Phänostruktur: Eine Zelle mit einer Liste aus Real-Atomen.
  - Genostruktur: Eine Zelle mit einer Liste, die Bit-Atome enthält.

Welche Kodierung zu welchem Problem paßt, kann auch der Tabelle B.1 entnommen werden.

Problem	Kodierungen
CFunction	Identity
DeJong3	EvolStratPair, GenAlgGrayCod, Identity
Griewank	EvolStratPair, GenAlgGrayCod, Identity
Hyperellipse	EvolStratPair, GenAlgGrayCod, Identity
Hypersphere	EvolStratPair, GenAlgGrayCod, Identity
nqueens	Identity
Rastingin	EvolStratPair, GenAlgGrayCod, Identity
Schwefel1	EvolStratPair, GenAlgGrayCod, Identity
Schwefel2	EvolStratPair, GenAlgGrayCod, Identity
SumDiffPow	EvolStratPair, GenAlgGrayCod, Identity
TSP	Perm2Bits,Perm2Gray,Perm2Ints,Identity

Tabelle B.1: Kompatiblität von Problemen und Kodierungen

Anhang C

Durchgeführte Experimente

C.1 GA mit Schwefelfunktion

Für dieses Experiment wurde der vom System bereitgestellte Genetische Algorithmus auf eine der Schwefelfunktionen angewendet. Diese Funktion hat die Formel:

$$F(\vec{x}) = 419,829 \cdot n - \sum_{i=1}^{n} x_i \cdot \sin(\sqrt{|x_i|})$$

Das gesuchte Minimum hat die Fitneß 0 und ergibt sich für einen Vektor, dessen Komponenten alle den Wert $420,9687$ haben. Für das Experiment wurde die Dimension 10 verwendet. Da die Funktion, wegen der Periodizität der Sinusfunktion, viele lokale Minima hat, ist das Finden einer optimalen Lösung hier ziemlich schwierig.

Untersucht wurde das Verhalten des Systems bei Veränderung der folgenden Parameter:

- Die Anzahl der Generationen, die das Verfahren berechnet.
- Die Anzahl der Individuen in der Population.
- Wieviele Nachkommen `mue` für diese Individuen erzeugt werden sollen.
- Die Wahrscheinlichkeit, mit der ein einzelnes Bit der Genostruktur mutiert wird.
- An wievielen Stellen der Genostruktur ein Crossover durchgeführt wird.
- Die Anzahl der Bits, mit denen ein reeller Wert kodiert wird.

Der Einfluß der folgenden Parameter wurden nicht untersucht:

- Die Wahrscheinlichkeit `nue`, mit der ein beim Crossover erzeugter Nachkomme übernommen wird. Für sie wurde der Wert 0 verwendet.
- Bei der Selektion der Faktor, mit dem das schlechteste Individuum gegenüber dem besten gewichtet wird. Bei den durchgeführten Experimenten war der Wert $0,2$.

Da das Verhalten des Genetischen Algorithmus bei dieser Funktion stark von der Ausgangspopulation abhängt, sind allgemeine Aussagen über den Einfluß der einzelnen Parameter schwierig. Dieses Experiment zeigte sich allerdings gegenüber kleinen Veränderungen der Parameter recht unempfindlich.

Für die untersuchte Funktion haben sich die folgenden Ausgangswerte für die Parameter als günstig erwiesen:

- Generationen: 500
- Anzahl der Individuen: 30
- Anzahl der Nachkommen: 100
- Mutationswahrscheinlichkeit: 0,005
- Crossover an 2 oder 3 Punkten
- Kodierung mit 10 Bits pro reeller Zahl

Auch mit diesen Werten wird das Optimum oft nicht gefunden. So kann das Verfahren in lokalen Minima hängen bleiben. Einige Ergebnisse können der folgenden Tabelle entnommen werden.

Gen.	Indiv.	mue	MutWahr.	Punkte	Bits	Fitneß
500	30	100	0,005	2	10	82,382
500	30	100	0,005	2	10	3,1948
500	30	100	0,005	2	10	118,45
500	30	100	0,005	2	10	0,0063647
500	30	100	0,005	2	10	473,85
500	30	100	0,005	2	10	0,0063647

Tabelle C.1: Ausgangswerte

Ein wichtiger Faktor ist die verwendete Anzahl von Bits für die Kodierung. Ist die Anzahl zu groß, braucht das Verfahren zu lang und bleibt oft in lokalen Minima stecken. Ist die Zahl zu gering, kann das globale Minimum meist nicht gefunden werden. Im Gegensatz zu anderen Funktionen, wie z.B. der Rastingin-Funktion, lieferte die Schwefelfunktion auch bei einer größeren Anzahl von Bits (bis zu 25) gute Ergebnisse. Da aber dadurch die Zeit, die für das Berechnen einer Generation nötig ist, deutlich ansteigt, wurde meist mit einer Bitzahl von 10 gearbeitet. Für eine deutlich kleinere Zahl von Bits werden nur noch lokale Optima gefunden.

Gen.	Indiv.	mue	MutWahr.	Punkte	Bits	Fitneß
500	30	100	0,005	2	5	268,83
500	30	100	0,005	2	5	268,83
500	30	100	0,005	2	5	268,83
500	30	100	0,005	2	15	362,55
500	30	100	0,005	2	15	126,16
500	30	100	0,005	2	15	356,01
500	30	100	0,005	3	15	0,0030056
500	30	100	0,005	3	15	118,48
500	30	100	0,005	3	15	0,044655
500	30	100	0,005	3	20	236,88
500	30	100	0,005	3	20	1,3338
500	30	100	0,005	3	20	237,25
500	30	100	0,005	2	25	0,0096720
500	30	100	0,005	2	25	120,69

Tabelle C.2: Kodierung

Für die Mutationswahrscheinlichkeit waren kleinere Werte günstig. Bei viel zu großen Werten (0,5 oder 0,1) wurden schlechte Ergebnisse erzielt. Auch bei einer Mutationswahrscheinlichkeit von 0,05 wurde nicht das globale Minimum gefunden. Meist wurde daher der Wert 0,005 verwendet.

Gen.	Indiv.	mue	MutWahr.	Punkte	Bits	Fitneß
500	30	100	0,5	2	10	1672,9
500	30	100	0,5	2	10	1675,8
500	30	100	0,1	2	10	687,40
500	30	100	0,1	2	10	628,86
500	30	100	0,05	1	10	272,47
500	30	100	0,05	1	10	280,40
500	30	100	0,05	1	10	163,36

Tabelle C.3: Mutationswahrscheinlichkeit

Während z.B. bei der Rastingin-Funktion mit einem Crossover an mehreren Punkten meist bessere Ergebnisse erzielt wurden, konnte bei der Schwefelfunktion auch bei Crossover an einem oder zwei Punkten das optimale Minimum gefunden werden.

Wichtiger als die Anzahl der Individuen ist eine genügend große Zahl von erzeugten Nachkommen. Allerdings kann eine kleinere Zahl von Individuen dazu führen, daß sich die Individuen zu ähnlich werden und ein lokales Minimum

Gen.	Indiv.	mue	MutWahr.	Punkte	Bits	Fitneß
500	30	100	0,005	1	10	120,06
500	30	100	0,005	1	10	0,0063647
500	30	100	0,005	3	10	355,33
500	30	100	0,005	3	10	63,024
500	30	100	0,005	3	10	0,0063647

Tabelle C.4: Crossover

nicht mehr verlassen können.

Gen.	Indiv.	mue	MutWahr.	Punkte	Bits	Ergebnis
500	5	25	0,005	2	10	868,70
500	5	25	0,005	2	10	651,96
500	5	25	0,005	2	10	236,89
500	10	10	0,005	2	10	593,90
500	10	10	0,005	2	10	837,37
500	10	10	0,005	2	10	1579,3
500	10	10	0,005	2	10	789,73
500	10	100	0,005	2	10	118,45
500	10	100	0,005	2	10	118,45
500	10	100	0,005	2	10	118,45
500	10	100	0,005	2	10	118,45
500	20	100	0,005	2	10	236,89
500	20	100	0,005	2	10	476,02
500	20	100	0,005	2	10	355,34
500	30	30	0,005	2	10	149,35
500	30	30	0,005	2	10	667,21

Tabelle C.5: Individuen und Nachkommen

Anhang D

Syntax von LEA

Dieser Anhang enthält die EBNF, die die Syntax von LEA beschreibt. Weiterhin werden alle Schlüsselworte von LEA und die Operatoren, die in Ausdrücken verwendet werden können, aufgelistet.

D.1 EBNF von LEA

Hier ist die EBNF (Erweiterte Bachus-Naur-Form) dargestellt, die dem Parser für LEA zugrundeliegt. Sie enthält die Grammatikregeln, aus denen LEA aufgebaut ist.

```
leaprog =          experiment | oporalg .

experiment =       "EXPERIMENT" ident ";"
                   loadlib
                   problemdecl
                   populationdecl
                   operatordecl
                   vardecl
                   "BEGIN"
                     statement {";" statement}
                   "END" ";" .

oporalg =          ("ALGORITHM" | "OPERATOR") ident
                     ["("callparamdekl ")"]
                     [":" type] ";"
                   loadlib
                   paramdecl
                   operatordecl
                   vardecl
                   "BEGIN"
                     statement {";" statement}
                   "END" ";" .
```

```
type =              "INT" | "REAL" | "BOOL" | "IND" |
                    "INDLIST" | "POP" .

callparamdecl =     type ident ["," type ident] .

loadlib             ["USES" ident {"," ident} [";"]] .

operatordecl =      ["OPERATORS" opdecl {";" opdecl}] .

populationdecl =    ["POPULATIONS" popdecl {";" popdecl}] .

problemdecl =       "PROBLEM" "=" constant ";" .

opdecl =            [ident "=" ident ["(" paramdef ")"]] .

paramdef =          ident ":" constant
                      {"," ident ":" constant} .

popdecl =           [ident "CODED" constant "LOG" constant
                      [ = ("RANDOMPOP" "(" constant ")" |
                      "LOADFROM" "(" constant ")")]] .

operatordekl =      ident "=" ident
                      ["(" ident ":" constant
                        {"," ident ":" constant} ")"] .

paramdekl =         ["PARAMETER" pardekl {";" pardekl}] .

pardekl =           [type ident "=" "(" constant ","
                      constant "," constant "," constant ")"] .

localvardekl =      ["VAR" vardekl {";" vardekl}] .

vardekl =           [type ident [":=" constant]
                      {"," ident [" := " constant]}] .

combident =         ident (":" ident | {"." ident}) .

statementlist =     statement {";" statement}

statement =         combident [":=" expr]
                    "IF" expr "THEN" statementlist
                      ["ELSE" statementlist] "FI" |
                    "WHILE" expr "DO" statementlist "OD" |
                    "REPEAT" statementlist "UNTIL" expr |
                    "FOR" ident ":=" expr "TO" expr "DO"
                      statementlist "OD" |
                    "RETURN" expr .
```

```
expr =              logexpr [("AND" | "OR") expr] .

logexpr =           algexpr
                      [("=" | "<>" | "<" | ">" |
                        "<=" | ">=") algexpr] .

algexpr =           term [("+" | "-") algexpr].

term =              factor [( "*" | "/" ) term] .

factor =            combident ["(" expr {"," expr} ")"] |
                    constant |
                    "(" expr ")" |
                    "-" factor |
                    "NOT" factor .
```

D.1.1 Abweichungen

Der Parser weicht in einigen Punkten von der Sprache ab, die durch diese EBNF definiert ist. So werden einige Konstrukte, die diese EBNF zulassen würde, schon während des Parsens durch semantische Prüfungen ausgeschlossen. Der Parser läßt z.B. keine Zuweisung an einen Funktionsaufruf zu, was nach dieser Definition eigentlich möglich wäre.

Das Verwenden von Parametern in *combident* (ident { „.“ ident }) ist zur Zeit noch nicht realisiert.

D.1.2 Probleme

Wegen einer eleganteren Realisierung in SML sind die EBNF-Ausdrücke für *expr* und *algexpr* rechtsassoziativ und nicht linksassoziativ. Da dies aber nur die Reihenfolge beeinflußt, in der Terme aufaddiert bzw. voneinander subtrahiert und and- bzw. oder-verknüpft werden, dürften bei der jetzigen Semantik der Sprache keine merklichen Auswirkungen auftreten (eigentlich nur, wenn Fehler auftreten).

D.2 Schlüsselwörter

Folgende Schlüsselwörter werden in LEA verwendet und sind daher als Bezeichner nicht zugelassen:

```
ALGORITHM, AND, BEGIN, BOOL, BREAK, CODED, DO, ELSE, END,
EXPERIMENT, FALSE, FI, FOR, IF, IND, INDLIST, INT, LOADFROM,
LOG, NOT, OD, OPERATOR, OPERATORS, OR, RANDOMPOP,
PARAMETERS, POP, POPULATIONS, PROBLEM, REAL, REPEAT,
RETURN, STRING, THEN, TO, TRUE, UNTIL, USES, VAR, WHILE
```

D.3 Operatoren

Die folgende Tabelle zeigt die in LEA verwendeten Operatoren nach der Stärke geordnet, mit der sie die Operanden binden. Der oberste Operator bindet dabei am stärksten, der unterste am schwächsten. Operatoren mit gleicher Stärke sind hier in Gruppen zusammengefaßt.

Operator	Wirkung	Definiert für
`-` als Vorzeichen	Invertiert das Argument	real, int
`NOT`	Negiert das Argument	bool
`*`	Multiplikation	real, int
`/`	Division	real, int
`+`	Addition	real, int
`-`	Subtraktion	real, int
`=`	Test auf gleich	bool
`<>`	Test auf ungleich	bool
`<`	Test auf kleiner	bool
`>`	Test auf größer	bool
`<=`	Test auf kleiner oder gleich	bool
`>=`	Test auf größer oder gleich	bool
`AND`	Logisches Und	bool
`OR`	Logisches Oder	bool

Literaturverzeichnis

[AJJ+94] Frank Amos, Karsten Jung, Kurt Jaeger, Bernd Kawetzki, Wilfried Kuhn, Oliver Pertler, Ralf Reißing, and Markus Schaal. Zwischenbericht der Projektgruppe Genetische Algorithmen. Technical report, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1994.

[AJK+95] Frank Amos, Karsten Jung, Bernd Kawetzki, Wilfried Kuhn, Oliver Pertler, Ralf Reißing, and Markus Schaal. Endbericht der Projektgruppe Genetische Algorithmen. Technical Report FK95/1, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1995.

[BBSS88] H. Blohm, T. Beer, U. Seidenberg, and H. Silber. *Produktionswirtschaft.* Neue Wirtschaftsbriefe, Herne/Berlin, 1988.

[Ben92] Martin Philip Bendsøe. Optimisation topologique et les méthodes d'homogénéisation. In *Lecture notes, Advanced COMETT Course*, Liege, Belgien, 1992.

[Beu81] P. Beutel. *Das asymmetrische travelling salesman problem.* Hain, 1981.

[Bra90] H. Braun. On solving travelling salesman problems by genetic algorithms. *Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe?*, 1990.

[Bru93] Ralf Bruns. Direct chromosome representation and adanced genetic operators for production scheduling. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Matteo, 1993. Morgan Kaufmann Publishers.

[BS93] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. In *Evolutionary Computation*, pages 1–23. The Massachusetts Institute of Technology, 1993.

[Cla96] V. Claus. Naturanaloge Verfahren. Vorlesung an der Fakultät Informatik, 1995/96.

[CS88] V. Claus and A. Schwill. *Duden Informatik.* Engesser,H., BI, Mannheim, 1988.

[Dav91] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinold, New York, 1991.

[DJ75] K. De Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems.* Doctoral thesis, University of Michigan, Ann Arbor, 1975.

[DS90] Gunter Dueck and Tobias Scheuer. Threshold acceptance: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, (90):161–175, 1990.

[Due93] Gunter Dueck. New optimization heuristics for the Great Deluge Algorithm and the Record–to–Record Travel. In *Journal of Computational Physics*, volume 104, pages 86–92, 1993.

[GH91] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.

[GILS96] Matthias Großmann, Darko Ivančan, Alexander Leonhardi, and Thomas Schmidt. Zwischenbericht der Projektgruppe Evolutionäre Algorithmen. Technical report, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1996.

[GL85] D. Goldberg and R. Lingle. Alleles, loci and the travelling salesman. In J. Greffenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, San Matteo, 1985. Morgan Kaufmann Publishers.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, 1989.

[GWH90] Claas de Groot, Diethelm Würtz, and Karl Heinz Hoffmann. Optimizing complex problems by nature's algorithms: Simulated Annealing and Evolution Strategy - a comparative study. In *Parallel Problem Solving from Nature, 1st Workshop, PPSN I*, pages 445–454. Springer–Verlag, 1990.

[Hö79] Herbert Hörnlein. Ein Algorithmus zur Strukturoptimierung von Fachwerkkonstruktionen. Master's thesis, Ludwig-Maximilians-Universität, München, März 1979.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

[Joh73] K. J. Johnson. *Operations Research.* VDI-Verlag, Düsseldorf, 1973.

[JW95] Karsten Jung and Nicole Weicker. Funktionale Spezifikation des Software–Tools EAGLE. Technical Report FK 2/95, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1995.

[Kir90] Uri Kirsch. On singular topologies in optimum structural design. *Structural Optimization*, 2:133–142, 1990.

[Kir94] Uri Kirsch. Singular and local optima in structural optimization. *AIAA-94-4267-CP*, pages 150–160, 1994.

[LLRKS85] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D. Shmoys. *The Travelling Salesman Problem.* Lawler,E.L., 1985.

[Mau94] Kurt Maute. Topologieoptimierung kontinuierlicher Tragwerksstrukturen. In Kurt Maute, editor, *Topologie Workshop — Ein Ansatz zur Entwicklung alternativer Strukturen*, pages 107–127, Stuttgart, 1994. Sonderforschungsbereich 230 — Natürliche Konstruktionen.

[Mle92] H. P. Mlejnek. Some aspects of the genesis of structures. *Structural Optimization*, 5:64–69, 1992.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzbog, Stuttgart, 1973.

[Sch81] Hans-Paul Schwefel. *Numerical optimization of computer models.* John Wiley & Sons, 1981.

[WSF89] D. Whitley, T. Starkweather, and D'Ann. Fuquay. Scheduling problems and travelling salesmen: The genetic edge recombination operator. *ICGA'89*, pages 133–140, 1989.

[Zäp82] G. Zäpfel. *Produktionswirtschaft. Operatives Produktionsmanagement.* de Gruyter, Berlin, 1982.