ELCOM

Ein Compiler für eine funktionale Programmiersprache

Diplomarbeit

Technische Universität Chemnitz

Fachbereich Informatik

Vorgelegt von

Dirk Großmann

Geboren am 16. Juli 1966 in Hagenow

Betreuer: *Prof. Dr. rer. nat. habil. Klaus Mätzel*

---

1. Juli 1991

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die funktionalen Programmiersprachen stellen eine interessante Alternative zu den herkömmlichen Sprachen, wie *C* und *Pascal*, dar. Im Gegensatz zu diesen kennen die (reinen) funktionalen Sprachen keine Seiteneffekte und keine Kontrollstrukturen außer der Funktionsanwendung. Vielmehr sieht man ein funktionales Programm als eine Funktion aus der Menge der Ein- in die Menge der Ausgabedaten an, und die Abarbeitung besteht in der Berechnung des Wertes eines Ausdruckes.

Aufgrund des Fehlens von Seiteneffekten läßt sich in jedem Ausdruck ein Unterausdruck durch einen anderen, der den gleichen Wert liefert, ersetzen, ohne daß sich der Wert des Gesamtausdrucks ändert. Ein Ausdruck hat, unabhängig von der Auswertungsreihenfolge, immer denselben Wert, so daß mehrere Unterausdrücke auch gleichzeitig ausgewertet werden können. Ein Wert darf dabei nicht nur ein elementares Objekt, wie z. B. eine Zahl, sondern auch eine Funktion oder eine komplexe (eventuell sogar eine potentiell unendliche) Datenstruktur sein.

Die Familie der funktionalen Sprachen können wir nach dem Zeitpunkt der Auswertung der Argumente einer Funktion bei der Funktionsanwendung in Sprachen mit strikter und Sprachen mit verzögerter Auswertung unterteilen. Sprachen mit strikter Auswertung werten die Argumente zuerst aus und wenden dann die Funktion auf die erhaltenen Werte an.

Bei den Sprachen mit verzögerter Auswertung werden Argumente von Funktionen erst dann ausgewertet, wenn ihr Wert unbedingt benötigt wird, und nicht, bevor die Funktion angewendet wird. Konstruktoren werten ihre Argumente nicht aus, werden also wie nicht-strikte Funktionen behandelt. Damit eine Datenstruktur vollständig ausgewertet wird, muß sie in einen Kontext gebracht werden, der diese Auswertung erzwingt. Ein solcher Kontext ist etwa die „print“-Funktion, die die Datenstruktur druckt und dazu die Auswertung der Argumente des obersten Konstruktors erzwingt. Damit können wir auch unendliche Datenstrukturen als Ausgabe eines funktionalen Programms behandeln.

Die Umgebung des Programms bestimmt, inwieweit die unendliche Datenstruktur ausgewertet werden soll und wann das Programm terminiert.

Weiterhin können wir die funktionalen Sprachen in ungetypte und streng getypte einteilen. Eine Sprache ist streng getypt, wenn ihr Compiler garantieren kann, daß die von ihm akzeptierten Programme ohne Typfehler ausgeführt werden. Polymorphe Sprachen gestatten die Vereinbarung von polymorphen Funktionen. Der Typ einer solchen Funktion enthält Variablen, so daß diese Funktion auf Argumente unterschiedlichen Typs angewendet werden kann.

In dieser Arbeit beschäftigen wir uns mit dem Entwurf und der Implementation eines experimentellen Compilers für eine funktionale Programmiersprache mit verzögerter Auswertung. Dem Compiler liegt das Prinzip der trägen Superkombinator-Graphenreduktion zugrunde. Er läuft unter dem Betriebssystem UNIX[1] und ist in der Sprache *C* geschrieben.

Das zweite Kapitel gibt einen Überblick über die Quell- und Zielsprache des Compilers sowie über seine Funktionsweise.

Das dritte Kapitel definiert die Syntax der von unserem Compiler akzeptierten Quellsprache, einer Version des getypten erweiterten Lambda-Kalküls.

Im vierten Kapitel behandeln wir ausführlich den syntaktischen Aufbau und die Bedeutung der Instruktionen unserer Zielsprache, des G-Codes, sowie die G-Maschine, eine abstrakte Maschine, die den G-Code ausführt.

Im fünften Kapitel entwickeln wir die interne Repräsentation der Quellsprach-Ausdrücke und bauen den Syntaxanalysator mit Hilfe von **Lex** und **Yacc**.

Im sechsten und siebenten Kapitel behandeln wir zwei wichtige Transformationen auf der Quellsprachebene: die Umbenennung der gebundenen Variablen sowie die Transformation der Funktionsanwendungen.

Das achte Kapitel befaßt sich mit dem Lambda-Lifting, das unsere Quellprogramme in eine Superkombinator-Form überführt.

Im neunten Kapitel stellen wir den Codegenerator vor und betrachten die von ihm genutzten Übersetzungsschemata.

Das zehnte Kapitel beschreibt, wie der Compiler angewendet wird.

Das abschließende elfte Kapitel zeigt Möglichkeiten Verbesserung des Compilers auf.

Im Anhang finden wir ein Beispiel für eine Übersetzung mit unserem Compiler, das dessen Funktionsfähigkeit demonstrieren soll, sowie die *C*-Quelltexte, die **Lex**- und die **Yacc**-Eingabedatei.

[1] UNIX ist ein eingetragenes Warenzeichen der Firma AT & T.

Kapitel 2

Grobentwurf des Compilers

Die prozeduralen Programmiersprachen sind Abstraktionen der von NEUMANN-Rechnerarchitektur, so daß die Übersetzung dieser Sprachen in die Maschinensprachen der heute verwendeten Rechner relativ effizient ist und der Übersetzungsprozeß nur aus den Teilen Analyse und Synthese besteht [AHO 1988]. Der Analyse-Teil zerlegt das Quellprogramm in seine Bestandteile, erzeugt eine interne Repräsentation des Quellprogramms und führt statische Überprüfungen (z. B. Typprüfungen) durch. Der Synthese-Teil konstruiert das gewünschte Zielprogramm aus der internen Quellprogrammrepräsentation.

Bei der Entwicklung der funktionalen Sprachen wurde dagegen wenig Rücksicht auf die Fähigkeiten der herkömmlichen Rechner genommen, vielmehr ging es um eine einfache, klare Semantik und bequeme Handhabbarkeit für den Programmierer. Den funktionalen Sprachen liegt der funktionale Programmierstil zugrunde, der nicht direkt von den von NEUMANN-Maschinen unterstützt wird. Daher sind die Übersetzungsprinzipien völlig andersgeartet, und die Compiler, die solche Sprachen übersetzen, führen zwischen dem Analyse- und dem Syntheseprozeß bestimmte Transformationen auf der Quell- bzw. einer Zwischensprachebene aus. Diese Compiler kommen nicht mehr mit wenigen Pässen aus, zehn und mehr sind nicht selten.

Da die Compiler funktionaler Sprachen ziemlich komplexe Gebilde sind und es bei einem Experimentalcompiler besonders auf Eigenschaften wie gewisse Allgemeinheit, klare Struktur und gute Verfolgbarkeit der einzelnen Aktionen ankommt, beschäftigen wir uns in diesem Kapitel mit den Fragen

- Welche Quellsprache soll unser Compiler akzeptieren?
- Auf welche Art und Weise erfolgt die Übersetzung dieser Sprache?
- In welchen Code soll das Quellprogramm übersetzt werden?

deren Antworten die Grundentscheidungen für unsere weitere Arbeit darstellen. In diesem Kapitel wollen wir uns diesen Fragen der Reihe nach zuwenden.

2.1 Entwurf der Quellsprache

Die modernen funktionalen Sprachen sind komplizierte Instrumente, die neben einer Vielzahl syntaktischer Feinheiten (z. B. die Listennotation sowie die Bedeutung der Einrücktiefe in *Miranda*[1] [Turner 1985, 1987]) Fähigkeiten wie die Mustererkennung (*Hope* [Burstall 1980], *Standard-ML* [Wikström 1987], *Lazy-ML* [Augustsson 1987, 1989], *Miranda*) und die Behandlung von Listenverständnissen (*Miranda*) [Augustsson 1987, Peyton Jones 1987] aufweisen. Es würde den Rahmen dieser Arbeit sprengen, auf diese Möglichkeiten einzugehen.

Glücklicherweise sind die „höheren“ funktionalen Sprachen mehr oder weniger syntaktische Versionen voneinander, ohne wesentliche semantische Unterschiede. Sie lassen sich ohne Einbuße an Mächtigkeit (nur an Bequemlichkeit für den Anwender) in eine syntaktisch einfache funktionale Sprache, den *erweiterten Lambda-Kalkül* [Peyton Jones 1987], transformieren. Der erweiterte Lambda-Kalkül ist mächtig genug, alle berechenbaren Funktionen von beliebigem Typ mit beliebig vielen Argumenten (und damit alle funktionalen Programme) auszudrücken.

Wir verwenden daher den erweiterten Lambda-Kalkül als zu implementierende Quellsprache und überführen den Quell-Zeichenstrom in einem ersten Schritt mit Hilfe eines Lexik- und Syntaxanalysators in eine interne Repräsentation, den Syntaxbaum. Dieser Syntaxbaum ist der Ausgangspunkt für den eigentlichen Compiler. Unser einfacher Syntaxanalysator kann später gegen einen, der eine „höhere“ Sprache akzeptiert (und einen Syntaxbaum gleicher Struktur erzeugt), ausgewechselt werden.

Da die meisten funktionalen Sprachen streng getypt sind, nehmen wir in unsere interne Repräsentation Typinformationen auf und legen damit die Grundlage für die Typinferenz, die wir im Gegensatz zur Behandlung der Mustererkennung und der Listenverständnisse, die früher erledigt werden muß, im erweiterten Lambda-Kalkül vornehmen können. Die Typinferenz werden wir in dieser Arbeit jedoch nicht ausführen, sondern den Compiler nur dahingehend vorbereiten, daß die Typinferenz leicht eingefügt werden kann. Der Compiler geht deshalb davon aus, daß das zu übersetzende Programm korrekt getypt ist.

Der erweiterte Lambda-Kalkül legt von vornherein nicht fest, ob die Auswertung strikt oder verzögert erfolgen soll. Wir verwenden die verzögerte Auswertung, da sie aufgrund der Möglichkeit potentiell unendlicher Datenstrukturen und sauberer interaktiver Ein- und Ausgabe einen echten semantischen Fortschritt gegenüber der strikten Auswertung darstellt.

Damit haben wir festgelegt, daß die Quellsprache unseres Compilers eine getypte Version des erweiterten Lambda-Kalküls mit verzögerter Auswertung ist.

[1] Das *Miranda*-System ist ein eingetragenes Warenzeichen der Firma Research Software Ltd.

2.1.1 Strukturierte Typen und Muster

Ein *Typ* ist die Bezeichnung für eine Menge von Datenobjekten. Objekte eines strukturierten Typs sind zusammengesetzt — etwa im Sinne der *Records* und *Varianten-Records* von *Pascal*. Zu jedem strukturierten Typ gibt es *Konstruktoren*; das sind Funktionen, die Datenobjekte eines Typs erzeugen.

Wir unterscheiden zwischen *Summen-* und *Produkttypen*. Einen Produkttyp kann man sich als Äquivalent eines *Record*-Typs in *Pascal* vorstellen. Er hat die Struktur

$$Tp \quad = \quad c \quad T_1 \quad \cdots \quad T_m$$

wobei c der Typkonstruktor ist und die T_i Typen sind. Ein Summentyp ist eine Art Varianten-*Record* und wie folgt strukturiert.

$$Ts \quad = \quad T_1 \quad | \quad T_2 \quad | \quad \cdots \quad | \quad T_n$$

Ein Objekt vom Typ Ts kann demnach vom Typ T_1 oder vom Typ T_2 oder ... oder vom Typ T_n sein. Allgemein sieht ein strukturierter Typ als Summentyp aus Produkttypen dann so aus.

$$\begin{array}{rcl} T & = & c_1 \quad T_{1,1} \quad \cdots \quad T_{1,r_1} \\ & | & \ldots \\ & | & c_n \quad T_{n,1} \quad \cdots \quad T_{n,r_n} \end{array}$$

wobei die c_i Konstruktoren der Stelligkeit r_i und die $T_{i,j}$ Typen sind.

Einen strukturierten Typ vereinbaren wir, indem wir einen *typformenden Operator* einführen. Eine Typdefinition hat den folgenden syntaktischen Aufbau.

$$T \quad ::= \quad \langle \text{typf-Op} \rangle \quad \text{“}=\text{”} \quad c_1 \ \ldots \ \text{“}|\text{”} \ \ldots \ \text{“}|\text{”} \ c_n \ \ldots$$

Typformende Operatoren treten nur in Typausdrücken auf, Konstruktoren stehen nur in Wertausdrücken. Alle Konstruktoren aller definierten Typen müssen sich voneinander unterscheiden.

Als Beispiel für das eben Gesagte sehen wir uns die Definition des Typs „Liste aus ganzen Zahlen“ an. Eine solche Liste ist entweder die leere Liste **INIL** oder ein Paar **ICONS**, dessen Kopf eine ganze Zahl und dessen Schwanz eine Liste ganzer Zahlen ist.

type intList = *INIL* | *ICONS* integer intList **end**

An dieser Stelle müssen wir den Konstruktor *INIL* vom durch ihn konstruierten Objekt **INIL** (das vom Typ **intList** ist), unterscheiden.

Wenn wir nicht nur Listen ganzer Zahlen, sondern Listen mit Elementen eines beliebigen (aber festen) Typs haben wollen, müssen wir unser Typkonzept auf *parametrisierte Typen* ausdehnen. Parametrisierte Typen beschreiben Mengen von konkreten Typen. In der Definition eines parametrisierten Typs

erweitern wir den typformenden Operator um *schematische* oder *Typvariablen*, deren Namen mit einem Stern beginnen.

$$\textbf{type}\ \text{list}\ *a\ =\ \mathit{NIL}\ \mid\ \mathit{CONS}\ *a\ (\text{list}\ *a)\ \textbf{end}$$

Durch die Belegung der schematischen Variablen mit Typen kommen wir zu den konkreten Typen. In dieser Arbeit betrachten wir alle Typen als mit null oder mehr schematischen Variablen parametrisiert; und die Definition des Typs T hat den folgenden syntaktischen Aufbau.

$$\begin{array}{lllllll} T & ::= & \langle\text{typf-Op}\rangle & *a_1\ \ldots\ *a_m & \text{“}=\text{”} & c_1 & \{\ T_{c1}\ \mid\ \langle\text{schem-Var}\rangle\ \} \\ & & & & \text{“}\mid\text{”} & \ldots & \\ & & & & \text{“}\mid\text{”} & c_n & \{\ T_{cn}\ \mid\ \langle\text{schem-Var}\rangle\ \} \end{array}$$

$$\text{mit}\ \langle\text{schem-Var}\rangle\ \in\ \{*a_1\ \ldots\ *a_m\}$$

Die schematischen Variablen der Typen T_{ci} müssen belegt sein. Insbesondere dürfen sie mit schematischen Variablen aus $\{*a_1 \ldots *a_m\}$ belegt sein.

Mit Hilfe der Typen können wir nun die *Muster* definieren, die wir verwenden wollen, um Teilmengen der Objekte eines Typs zu bilden. Wir sagen, daß ein Datenobjekt zu der durch das Muster spezifizierten Teilmenge gehört, wenn das Muster auf das Objekt paßt. In dieser Arbeit betrachten wir nur einfache Muster, die wir brauchen, um zu testen, von welchem Konstruktor ein Objekt erzeugt wurde, und um strukturierte Objekte auseinanderzunehmen.

> Ein *einfaches Muster* für Objekte des Typs T ist von der Form $(c\ v_1\ \ldots\ v_r)$, wobei c ein Typkonstruktor von T der Stelligkeit r ist und die v_i Variablen sind.

Die Überführung von komplexen Mustern in einfache ist in [Peyton Jones 1987] sowie in [Augustsson 1987] erläutert.

2.1.2 Ausdrücke

Ein Programm im erweiterten Lambda-Kalkül besteht aus einer Liste von Typvereinbarungen, gefolgt von einem Ausdruck, dessen Wert berechnet werden soll. Wir betrachten zuerst die Ausdrücke im Lambda-Kalkül. Ein Ausdruck im Lambda-Kalkül ist entweder eine Konstante, eine Variable, eine Funktionsanwendung oder eine Lambda-Abstraktion.

⟨ausdruck⟩	::=	⟨konst⟩	Konstante
	|	⟨var⟩	Variable
	|	⟨ausdruck⟩ ⟨ausdruck⟩	Funktionsanwendung
	|	$\lambda\,\langle\text{var}\rangle\,.\,\langle\text{ausdruck}\rangle$	λ-Abstraktion

Die Semantik des Lambda-Kalküls geben wir in Form der *bezeichnenden Semantik* an, indem wir eine Funktion **Eval** von der Menge der Ausdrücke des Lambda-Kalküls in die Menge der Werte definieren.

$$\begin{aligned}
\mathbf{Eval}[k]\ p &= k \\
\mathbf{Eval}[f]\ p &= \text{Entsprechend der Semantik der} \\
&\quad\ \text{eingebauten Funktion } f \\
\mathbf{Eval}[x]\ p &= p(x) \\
\mathbf{Eval}[E_1\ E_2]\ p &= (\mathbf{Eval}[E_1]\ p)\quad(\mathbf{Eval}[E_2]\ p) \\
\mathbf{Eval}[\lambda\, x\,.\, E]\ p\ a &= \mathbf{Eval}[E]\quad p\,[x \leftarrow a]
\end{aligned}$$

Hierbei ist k eine Konstante, f der Name einer in den Lambda-Kalkül eingebauten Funktion, x eine Variable, a der Wert des Arguments einer Lambda-Abstraktion, E, E_1 und E_2 Ausdrücke sowie p die Umgebung (Kontextinformation), d. h. eine Funktion, die Variablennamen auf ihre Werte abbildet. Die Funktion $p\,[x \leftarrow a]$ benimmt sich für alle Namen $y \neq x$ wie p, sonst liefert sie den Wert a.

Beim Übergang auf den erweiterten Lambda-Kalkül nehmen wir die **let(rec)**-, **construct**-, **select-**, **case**- und **fatbar**-Ausdrücke hinzu, mit denen wir Lambda-Ausdrücke benennen, zusammengesetzte Datenstrukturen konstruieren und auseinandernehmen bzw. eine einfache Form der Mustererkennung realisieren können.

Die **let**- und **letrec**-Ausdrücke dienen dazu, ansonsten anonymen Lambda-Ausdrücken einen Namen zuzuordnen. Wir beginnen mit den **let**'s, die die folgende Syntax haben.

$$\textbf{let}\quad v\ =\ B\quad \textbf{in}\quad E$$

Dabei ist v eine Variable, und B sowie E sind Ausdrücke im erweiterten Lambda-Kalkül. Der **let**-Ausdruck führt eine Definition für eine Variable v ein, die v an B in E bindet. Die Definition $v = B$ ist in E, nicht aber in B sichtbar. Die Semantik wird durch die folgende Gleichung beschrieben.

$$\mathbf{Eval}[\textbf{let}\ v = B\ \textbf{in}\ E]\quad p\quad =\quad \mathbf{Eval}[E]\quad p\,[v \leftarrow (\mathbf{Eval}[B]\ p)]$$

Der Aufbau des **letrec**-Ausdruckes ist ähnlich dessen für **let**, nur daß wir hier mehrere Definitionen auf einmal einführen können.

$$\begin{aligned}
&\textbf{letrec} \\
&\qquad v_1 \quad = \quad E_1\,; \\
&\qquad \ldots \\
&\qquad v_n \quad = \quad E_n \\
&\textbf{in}\quad E
\end{aligned}$$

Das „*letrec*“ steht für „*rekursives let*“ und führt möglicherweise rekursive Bindungen für eine Menge von Variablen v_i ein. Die v_i sind demnach sowohl in

E als auch in den E_i sichtbar. Dadurch erhalten wir die Möglichkeit, direkte und indirekte Rekursion zu beschreiben. Die Semantikgleichung für die **letrec**'s sieht dann so aus.

$$\mathbf{Eval}\left[\begin{array}{c}\mathbf{letrec}\\ v_1 \;=\; E_1\,;\\ \ldots\\ v_n \;=\; E_n\\ \mathbf{in}\;E\end{array}\right]\;p \quad=\quad \mathbf{Eval}[E]\quad q$$

wobei $\quad q \;=\; \mathbf{fixpunkt}\,(\lambda\, p'\,.\,p\,[v_1 \leftarrow (\mathbf{Eval}[E_1]\;p'),\ \ldots\ ,\ v_n \leftarrow (\mathbf{Eval}[E_n]\;p')])$

Mit Hilfe der **construct**-Ausdrücke können wir strukturierte Datenobjekte aus einfacheren konstruieren. Diese Ausdrücke sind wie folgt aufgebaut.

$$\mathbf{construct}\,(k,\ E_1, \ldots,\ E_r)$$

Dabei ist k ein Konstruktor eines Typs T mit der Stelligkeit r. Die E_i müssen sich zu Datenobjekten der in der Definition von k angegebenen Typen auswerten. Die Bedeutung ist durch die nachstehenden Gleichungen festgelegt.

$$\begin{array}{l}\mathbf{Eval}[\mathbf{construct}(k,\ E_1,\ \ldots,\ E_r)]\;\;p\\ \quad=\quad (\text{STRUCT}\;k\;(\mathbf{Eval}[E_1]\;p)\;\ldots\;(\mathbf{Eval}[E_r]\;p)) \quad\in\quad T\end{array}$$

$$\begin{array}{lrcl}\text{wenn} & T & = & \ldots\ \mid\ k\ T_1\ \ldots\ T_r\ \mid\ \ldots\\ \text{und} & \mathbf{Eval}[E_1]\;p & \in & T_1\\ \text{und} & \ldots & & \\ \text{und} & \mathbf{Eval}[E_r]\;p & \in & T_r\end{array}$$

$$\mathbf{Eval}[\mathbf{construct}(k)]\;\;p \quad=\quad (\text{STRUCT}\;k) \quad\in\quad T$$

$$\text{wenn}\quad T \;=\; \ldots\ \mid\ k\ \mid\ \ldots$$

Die **select**-Ausdrücke dienen dazu, Komponenten aus zusammengesetzten Datenobjekten zu extrahieren.

$$\mathbf{select}\,(\mathit{index},\ E)$$

$$\mathbf{Eval}[\mathbf{select}(i,\ E)]\;\;p \quad=\quad x_i$$

$$\text{wenn}\quad \mathbf{Eval}[E]\;\;p \quad=\quad (\text{STRUCT}\;k\;x_1\ \ldots\ x_i\ \ldots\ x_r)$$

Die **case**-Ausdrücke führen wir ein, um eine einfache Form der Mustererkennung zu implementieren. Ein **case**-Ausdruck hat die folgende syntaktische Struktur.

$$\begin{array}{ll}\mathbf{case} & E\quad\mathbf{of}\\ & c_1\quad v_{1,1}\ \ldots\ v_{1,r_1}\quad\Longrightarrow\quad E_1\,;\\ & \ldots\\ & c_n\quad v_{n,1}\ \ldots\ v_{n,r_n}\quad\Longrightarrow\quad E_n\\ \mathbf{end} & \end{array}$$

Der Ausdruck E muß sich zu einem Objekt von einem bestimmten Typ auswerten, und die c_i sind Konstruktoren dieses Typs (es brauchen nicht alle zu sein). Ein **case**-Ausdruck testet ein Objekt eines festen Summentyps gegen eine bestimmte Anzahl von Konstruktoren dieses Typs ab. Paßt es auf einen der Konstruktoren, wählt **case** den Ausdruck, der mit diesem Konstruktor verbunden ist, zur Berechnung aus. In diesem Ausdruck kann über die Selektorvariablen $v_{i,j}$ Bezug auf Teilstrukturen genommen werden. Der Wert des **case**-Ausdrucks ist der Wert des ausgewählten Unterausdrucks.

$$\mathbf{Eval}\left[\begin{array}{cccc} & \textbf{case } E \textbf{ of} & & \\ c_{c1} & v_{1,1} \ldots v_{1,r_1} & \Longrightarrow & E_1\,; \\ & \cdots & & \\ c_i & v_{i,1} \ldots v_{i,r_i} & \Longrightarrow & E_i\,; \\ & \cdots & & \\ c_n & v_{n,1} \ldots v_{n,r_n} & \Longrightarrow & E_n \\ & \textbf{end} & & \end{array}\right] \quad p \quad = \quad \mathbf{Eval}[E_i]\ q$$

$$\text{wobei} \quad q \quad = \quad p\,[v_{i,1} \leftarrow k_{i,1},\ \ldots\ ,\ v_{i,r_i} \leftarrow k_{i,r_i}\,]$$

$$\begin{array}{lrcl} \text{wenn} & \mathbf{Eval}[E]\ p & = & (\text{STRUCT } c_i\ k_{i,1}\ \ldots\ k_{i,r_i}) \quad \in \quad T \\ \text{und} & T & = & \ldots \quad | \quad c_i \quad T_{i,1} \quad \cdots \quad T_{i,r_i} \quad | \quad \ldots \\ \text{und} & k_{i,j} & \in & T_{i,j} \qquad \forall\, j \quad (1 \le j \le r_i) \end{array}$$

Wenn das Objekt auf keinen der im **case**-Ausdruck aufgeführten Konstruktoren paßt, ist der Wert des **case**-Ausdrucks der spezielle Wert „FAIL“.

$$\mathbf{Eval}\left[\begin{array}{cccc} & \textbf{case } E \textbf{ of} & & \\ c_{c1} & v_{1,1} \ldots v_{1,r_1} & \Longrightarrow & E_1\,; \\ & \cdots & & \\ c_n & v_{n,1} \ldots v_{n,r_n} & \Longrightarrow & E_n \\ & \textbf{end} & & \end{array}\right] \quad p \quad = \quad \text{FAIL}$$

$$\begin{array}{lrcl} \text{wenn} & \mathbf{Eval}[E]\ p & = & (\text{STRUCT } c_i\ k_{i,1}\ \ldots\ k_{i,r_i}) \quad \in \quad T \\ \text{und} & T & = & c_1 \quad T_{1,1} \quad \cdots \quad T_{1,r_1} \\ & & | & \ldots \\ & & | & c_m \quad T_{m,1} \quad \cdots \quad T_{m,r_m} \\ \text{und} & c_i & \neq & c_{c1},\ \ldots\ ,\ c_i \quad \neq \quad c_n \end{array}$$

Die **if**-Ausdrücke mit der Gestalt **if** E **then** E_1 **else** E_2 sind nichts weiter als „syntaktischer Zucker“ für folgende **case**-Ausdrücke

$$\begin{array}{lrcl} \textbf{case} & E & \textbf{of} & \\ & \mathit{TRUE} & \Longrightarrow & E_1\ ; \\ & \mathit{FALSE} & \Longrightarrow & E_2 \\ \textbf{end} & & & \end{array}$$

wobei der Wert des Ausdrucks E vom Typ *boolean* ist, der durch die beiden Konstruktoren *TRUE* und *FALSE* gebildet wird.

Für die Übersetzung von komplexen Mustern benötigen wir eine letzte Ausdrucksart, die **fatbar**-Ausdrücke mit der Syntax **fatbar** $(E_1\ E_2)$. Sie werten den Ausdruck E_1 aus und prüfen, ob dessen Wert gleich FAIL ist. Wenn ja, ist der Wert des **fatbar**-Ausdrucks der Wert von E_2, sonst der Wert von E_1.

$$\begin{aligned}
\mathbf{Eval}[\mathbf{fatbar}(E_1\ E_2)]\ p &= a \\
\text{wenn}\quad \mathbf{Eval}[E_1]\ p &= a \neq \text{FAIL} \\
\mathbf{Eval}[\mathbf{fatbar}(E_1\ E_2)]\ p &= \mathbf{Eval}[E_2]\ p \\
\text{wenn}\quad \mathbf{Eval}[E_1]\ p &= \text{FAIL}
\end{aligned}$$

Somit ist unsere Quellsprache vollständig spezifiziert. In [Peyton Jones 1987] wird gezeigt, wie sich die in „höheren“ funktionalen Sprachen typischen Strukturen auf den hier definierten erweiterten Lambda-Kalkül abbilden lassen.

2.2 Übersetzungsprinzipien

Den heutigen Implementationen funktionaler Sprachen liegt eine der beiden Strategien zur Ausführung funktionaler Programme zugrunde, die *umgebungsbasierte Strategie* und die *Graphenreduktion*. Beim Entwurf eines Compilers legt die Entscheidung für eine der Abarbeitungsweisen die Übersetzungsprinzipien fest, weil der Compiler im Übersetzungsprozeß eines Quellprogramms Wissen darüber benutzt, wie der erzeugte Code abgearbeitet wird.

Umgebungsbasierte Abarbeitung und Graphenreduktion unterscheiden sich hinsichtlich der internen Repräsentation von Funktionsobjekten und von freien Variablen eines Lambda-Ausdrucks sowie in der Art des Ablaufs von Auswertungen. Bei der umgebungsbasierten Strategie greifen die Lambda-Ausdrücke auf Umgebungen zu, die die Wertbindungen ihrer freien Variablen enthalten. Ein Funktionsobjekt ist als *Closure*, einer Zusammenfassung der Repräsentation einer Lambda-Abstraktion mit der Menge der freien Bindungen, dargestellt. Zwei Bezüge auf die gleiche Variable veranlassen den Abruf derselben Bindung aus der Umgebung; auf diese Weise ersparen wir uns, die Bindung zu duplizieren. Bei der Anwendung einer Closure auf ihre Argumente wird die alte Umgebung gegen die in der Closure gespeicherten ersetzt und diese durch die Bindungen der formalen Parameter an die entsprechenden Argumente erweitert. In dieser neuen Umgebung wird der Körper der Lambda-Abstraktion ausgewertet. Danach ist die alte Umgebung wieder aktuell.

Die umgebungsbasierte Übersetzung eignet sich hervorragend für strikte Sprachen und führt dort zu sehr effizienten Implementationen. Für die Sprachen mit verzögerter Auswertung ist dieses Verfahren in seiner reinen Form weniger günstig, da hier die verzögerte Auswertung nur über die Definition und spätere

Anwendung von Lambda-Abstraktionen möglich ist. Erweitern wir unsere umgebungsbasierte Strategie um Möglichkeiten zum expliziten Verzögern und Veranlassen der Auswertung eines Ausdrucks (HENDERSON's träge SECD-Maschine [HENDERSON 1980]), sind wir auf dem besten Wege zur Graphenreduktion.

Die Grundlage der Graphenreduktion ist die Repräsentation des auszuwertenden erweiterten Lambda-Ausdrucks als gerichteter Graph. Die graphische Repräsentation kommt ohne Umgebungen aus: Mehrfache Bezüge zum gleichen Argumentausdruck sind durch mehrfache Kanten zu einem Argumentgraphen dargestellt. Der Prozeß der Graphenreduktion transformiert den Ausdrucksgraphen in eine Normalform[2], indem nacheinander Wurzeln von reduzierbaren Teilgraphen durch die entsprechenden Werte ersetzt werden. Wir sagen auch, daß der Graph zu einer Normalform reduziert wird.

Da wir mit Hilfe der Graphen in einfacher Form unausgewertete Ausdrücke repräsentieren können und wir die verzögerte Auswertung nicht über explizites Verzögern und Veranlassen der Auswertung, sondern nur über die Reihenfolge der Reduktionen realisieren, ist die Graphenreduktion ideal für die Implementation funktionaler Sprachen mit verzögerter Auswertung. Die Implementation der verzögerten Auswertung hat zwei Bestandteile.

- Argumente von Funktionen sollten nur dann ausgewertet (reduziert) werden, wenn ihr Wert gebraucht wird, und nicht, wenn die Funktion angewendet wird.
- Argumente sollten nur einmal ausgewertet werden, weitere Bezüge auf das Argument innerhalb der Funktion benutzen den Wert, der beim erstenmal berechnet wurde.

Den ersten Teil erledigen wir durch die Normalreduktion[3] des Graphen, wobei wir stoppen, wenn der Graph auf dem obersten Niveau keine reduzierbaren Ausdrücke enthält (Normalreduktion bis zur schwachen Kopf-Normalform). Der zweite Teil wird durch die Kombination von zwei Dingen erreicht.

- Bei der Ausführung von β-Reduktionen setzen wir Zeiger auf das Argument ein, anstatt es zu kopieren.
- Das Aktualisieren der Wurzel des reduzierten Ausdrucks mit der Wurzel des Ergebnisses sichert, daß weitere Verwendungen des Arguments von der getanen Arbeit profitieren.

Diese Implementationsstrategie bezeichnen wir als *träge Graphenreduktion*.

Eine detailierte Besprechung der umgebungsbasierten Abarbeitungsstrategie und der Graphenreduktion finden wir in [GROSSMANN 1990].

Nachdem wir uns für die träge Graphenreduktion als Abarbeitungsstrategie entschieden haben, betrachten wir die Übersetzungsprinzipien, die die diese Strategie bedingt.

[2] Wert des Ausdrucks

[3] Reduktion des jeweils ganz links stehenden, äußersten reduzierbaren Teilausdrucks zuerst

Bei der Ausführung der trägen Graphenreduktion ist die β-Reduktion neben dem Entfalten des Rückgrats die am häufigsten auftretende Operation, daher ist die Effizienz ihrer Ausführung entscheidend für die Effizienz der gesamten Programmabarbeitung. Wir setzen also hier mit der Compilation an und überführen jede Lambda-Abstraktion in eine feste Folge von Instruktionen, die bei ihrer Abarbeitung eine Instanz[4] des Lambda-Körpers konstruiert.

Nachteiligerweise eignen sich Lambda-Abstraktionen mit freien Variablen nicht direkt für die Compilation, wir müssen sie daher in eine für uns geeignete Form bringen. Dafür gibt es zwei Möglichkeiten.

1. Wir machen alle freien Variablen einer Lambda-Abstraktion zu zusätzlichen Argumenten und überführen somit alle Lambda-Abstraktionen in Superkombinatoren. Die nutzerdefinierten Funktionen werden als Ersetzungsregeln im Graphen verwendet. Die Transformation eines Programms in Superkombinator-Form heißt *Lambda-Lifting*.

2. Anstatt aus allen Lambda-Abstraktionen Superkombinatoren zu erzeugen, gehen wir von einer festen Menge Superkombinatoren (**S**, **K** und **I**) aus und transformieren das funktionale Programm in eine Form, die außer eingebauten Funktionen und Konstanten nur die Kombinatoren **S**, **K** und **I** enthält.

In unserem Compiler wollen wir die erste Methode verwenden, da die SK-Transformation unser Programm in zu viele kleine Abarbeitungsschritte zerlegt, so daß wir für die vielen Anwendungsknoten eine Menge Zwischenspeicher verbrauchen und das „Korn“ der Ausführung zu klein wird.

2.3 Die Zielsprache

Wenn wir Superkombinator-Körper in eine Folge von Instruktionen übersetzen wollen, benötigen wir eine Sprache, in der diese Instruktionen geschrieben werden. Die Maschinensprache eines bestimmten Rechners zu verwenden wäre schlecht, weil wir erstens immer wieder von vorn anfangen würden, wenn wir einen Codegenerator für eine andere Maschine brauchen, und uns zweitens der Gefahr aussetzen könnten, das Problem der Superkombinator-Compilation mit dem Problem, alle besonderen Möglichkeiten des verwendeten Rechners auszureizen, zu vermischen.

Wir verwenden daher als Zielsprache unseres Compilers einen Zwischencode, der aus Befehlen für eine abstrakte Graphenreduktionsmaschine, der G-Maschine, besteht. Die G-Maschine ist eine sequentielle Stack-Maschine mit „gewöhnlichen“ Instruktionen wie arithmetischen, logischen und Sprungbefehlen, aber auch Befehlen für die Konstruktion und die Manipulation von Ausdrucksgraphen. Sie wurde von T. Johnsson und L. Augustsson [Johnsson

[4] Kopie des Körpers, in der alle freien Vorkommen der formalen Parameter durch Zeiger auf das entsprechende Argument ersetzt sind

1987, Augustsson 1987, 1989] erfunden und dient dazu, die träge Graphenreduktion den konventionellen von Neumann-Rechnern näherzubringen.

Wir können sagen, daß unser Compiler aus jedem funktionalen Programm einen G-codierten Superkombinator-Interpreter konstruiert. Da die Befehle der G-Maschine auf dem Assembler-Niveau der von Neumann-Maschinen liegen (jeder G-Maschineninstruktion entspricht eine Folge von Befehlen der Zielmaschine), können sie relativ leicht in die Maschinensprachen heutiger Rechner übersetzt werden.

Fradet und le Métayer beschreiben eine alternative Methode zur Übersetzung funktionaler Sprachen — die Programmtransformation innerhalb des funktionalen Rahmens [Fradet 1991]. Sie überführen Lambda-Ausdrücke in einfachere funktionale Ausdrücke und betrachten deren Reduktion als Abarbeitung auf einer traditionellen Maschine mit zwei Komponenten: dem Code und dem Stack. Dadurch sparen sie die Einführung einer abstrakten Maschine durch operationale Beschreibung der Zustandsübergänge während der Berechnung — der Ausdruck selbst ist ein „Maschinenzustand“, und die Auswertung wird durch die Definition von Basisfunktionen (Kombinatoren) spezifiziert. Der Vorteil dieser Methode besteht darin, daß Korrektheitsbeweise einfacher werden.

2.4 Die Grobstruktur des Compilers

Abschließend geben wir die Grobstruktur unseres Compilers an, die dadurch gegeben ist, daß die folgenden Operationen nacheinander ausgeführt werden müssen [Augustsson 1989, Wray 1989].

- Lexikalische und syntaktische Analyse des im erweiterten Lambda-Kalkül geschriebenen Programms und Aufbau des Syntaxbaumes.
- Umbenennung aller gebundenen Variablen derart, daß danach alle eingeführten Variablen unterschiedliche Namen tragen.
- Transformation von Funktionsanwendungen in eine solche Form, daß auf der linken Seite (als Funktion) nur Bezeichner stehen.
- Lambda-Lifting.
- Übersetzung in G-Code.

Die nächsten Kapitel beschäftigen sich mit der Definition der Quellsprache sowie der G-Maschine und ihrer Befehle. Im weiteren beschreiben wir die Implementation und Anwendung unseres Compilers.

Kapitel 3

Definition der Quellsprache

Dieses Kapitel gibt die Syntax der Quellsprache unseres Compilers in der erweiterten BACKUS-NAUR-Form an. Terminalsymbole sind in Anführungsstriche eingeschlossen, Nichtterminale in spitze Klammern.

⟨programm⟩ ::= [⟨typvereinb⟩] ⟨ausdruck⟩
⟨typvereinb⟩ ::= "**type**" ⟨vereinb⟩ { ";" ⟨vereinb⟩ } "**end**"
⟨vereinb⟩ ::= ⟨typf-Op⟩ { ⟨schem-Var⟩ } "="
⟨summentyp⟩ { "|" ⟨summentyp⟩ }
⟨summentyp⟩ ::= ⟨konstruktor⟩ { ⟨produkttyp⟩ }
⟨produkttyp⟩ ::= ⟨bezeichner⟩ | ⟨schem-Var⟩ |
"(" ⟨typf-Op⟩ { ⟨produkttyp⟩ } ")"
⟨schem-Var⟩ ::= "*" ⟨bezeichner⟩
⟨typf-Op⟩ ::= ⟨bezeichner⟩
⟨konstruktor⟩ ::= ("A" | ... | "Z") { ⟨Buchstabe⟩ | ⟨Ziffer⟩ }

⟨ausdruck⟩ ::= ⟨konst⟩ | ⟨var⟩ | ⟨anwendung⟩ | ⟨abstraktion⟩ |
⟨if-ausdr⟩ | ⟨case-ausdr⟩ | ' " ' ⟨Zeichenkette⟩ ' " ' |
⟨fatbar-ausdr⟩ | ⟨cons-ausdr⟩ | ⟨sel-ausdr⟩ |
⟨let-ausdr⟩ | ⟨letrec-ausdr⟩ | "**fail**"
⟨konst⟩ ::= ⟨Int-Zahl⟩ | ⟨Real-Zahl⟩ | "'" ⟨Zeichen⟩ "'"
⟨anwendung⟩ ::= "(" ⟨ausdruck⟩ ⟨ausdruck⟩ ")"
⟨abstraktion⟩ ::= "**lambda**" ⟨var⟩ { ⟨var⟩ } "." ⟨ausdruck⟩ "**end**"
⟨if-ausdr⟩ ::= "**if**" ⟨ausdruck⟩
"**then**" ⟨ausdruck⟩
"**else**" ⟨ausdruck⟩
"**end**"
⟨case-ausdr⟩ ::= "**case**" ⟨ausdruck⟩ "**of**" ⟨fall⟩ { ";" ⟨fall⟩ }
"**end**"
⟨fall⟩ ::= "(" ⟨konstruktor⟩ { ⟨var⟩ } ")" "=>" ⟨ausdruck⟩

⟨fatbar-ausdr⟩	::=	“**fatbar**” “(” ⟨ausdruck⟩ ⟨ausdruck⟩ “)”
⟨cons-ausdr⟩	::=	“**construct**” “(” ⟨konstruktor⟩ { “,” ⟨ausdruck⟩ } “)”
⟨sel-ausdr⟩	::=	“**select**” “(” ⟨Int-Zahl⟩ “,” ⟨ausdruck⟩ “)”
⟨let-ausdr⟩	::=	“**let**” ⟨var-Vereinb⟩ “**in**” ⟨ausdruck⟩ “**end**”
⟨var-Vereinb⟩	::=	⟨var⟩ “=” ⟨ausdruck⟩
⟨letrec-ausdr⟩	::=	“**letrec**” ⟨var-Vereinb⟩ { “;” ⟨var-Vereinb⟩ } “**in**” ⟨ausdruck⟩ “**end**”
⟨var⟩	::=	⟨bezeichner⟩
⟨bezeichner⟩	::=	(“a” | . . . | “z”) { ⟨Buchstabe⟩ | ⟨Ziffer⟩ }

Alle in den Typvereinbarungen eingeführten Namen müssen voneinander verschieden sein. Kommentare beginnen mit einem Prozentzeichen und erstrecken sich bis zum Zeilenende. Die Zeichenketten sind als Listen von Zeichen gespeichert und werden in Ausdrücke zur Konstruktion solcher Listen übersetzt. Wir benötigen daher folgende Typvereinbarung, wenn wir Zeichenketten benutzen.

type list $*a$ = NIL | CONS $*a$ (list $*a$) **end**

In unserer Sprache gibt es eine Reihe vordefinierter Bezeichner, die Namen von eingebauten Funktionen darstellen.

add, sub, mult, div, mod (Typ: *num* → *num* → *num*):
Mit dem Typnamen „*num*“ meinen wir einen der Wertebereiche „*integer*“ oder „*real*“. Der Funktionswert ist die Summe, die Differenz, das Produkt, der Quotient bzw. der Divisionsrest der beiden „*curried*“ angegebenen Argumente.

neg (Typ: *num* → *num*):
Der Wert ist das Argument mit umgekehrtem Vorzeichen.

lt, leq, eq, neq, geq, gt (Typ: $*a$ → $*a$ → *integer*, wobei $*a$ = *num* oder $*a$ = *char*):
Die Vergleichsoperationen. Das Ergebnis ist eine ganze Zahl, wobei wir 0 als Wahrheitswert „*falsch*“ und eine Zahl ungleich 0 als „*wahr*“ betrachten.

and, or (Typ: *integer* → *integer* → *integer*) und **not** (Typ *integer* → *integer*):
Die logischen Operationen.

ord (Typ: *char* → *integer*) und **chr** (Typ: *integer* → *char*):
Konvertierung von Zeichen in ganze Zahlen und umgekehrt.

seq (Typ: $*a$ → $*b$ → $*b$):
Diese Funktion wertet ihr erstes Argument aus und verwirft es. Der Wert ist das zweite Argument.

read (Typ: list *char* → list *char*):
Um diese Funktion benutzen zu können, benötigen wir wieder die Typvereinbarung für die Listen. Das Argument wird als Name der zu lesenden

Datei (bzw. **NIL** für die Standardeingabe) interpretiert. Der Wert ist der Inhalt dieser Datei als (natürlich verzögert ausgewertete) Liste von Zeichen.

Ein Beispiel soll unsere Sprachdefinition verdeutlichen. Betrachten wir das *Miranda*-Programm

```
from n            =  n : from (succ n)
                     where
                         succ n = n + 1
prod (x:xs) m     =  x , x >= m
                  =  x * (prod xs m)
fac n             =  prod (from 1) n
-------------------------------------
fac 10
```

das den Wert von (10!) berechnet. Dieses können wir wie folgt in unseren erweiterten Lambda-Kalkül übersetzen.

```
type list *a = NIL | CONS *a (list *a) end

letrec
    fac = lambda n .
            letrec
                from = lambda n .
                        construct(CONS, n, (from
                            (lambda n . ((add n) 1) end n)))
                       end;
                prod = lambda x m .
                        case x of CONS h t =>
                            if ((geq h) m)
                                then h
                                else ((mult h) ((prod t) m))
                            end
                        end
                       end
            in
                ((prod (from 1)) n)
            end
          end
in
(fac 10)
end
```

Kapitel 4

Definition der G-Maschine

Die G-Maschine [Johnsson 1987, Peyton Jones 1987, Loogen 1990] ist eine abstrakte Maschine zur Graphenreduktion von Superkombinator-Systemen. Sie besteht aus sechs Komponenten.

$$\langle S,\ G,\ C,\ D,\ V,\ O\rangle$$

Den Mittelpunkt bilden der *Graph* G, der den zu reduzierenden Programmgraphen darstellt, und der *Stack* S, über den wir auf den Graphen zugreifen. Wir fassen den Graphen als Abbildung aus der Menge der Knotennamen (Knotenadressen) in die Menge der Knoten auf und unterscheiden die folgenden Knotentypen.

- Datenknoten: (INT i), (REAL x), (CHAR ch) sowie (FAIL).
- Zusammengesetzte Datenobjekte (STRUCT $k\ n_1\ \ldots\ n_r$) von einem Summentyp T, wobei k die Nummer des entsprechenden Konstruktors c_k mit der Stellighkeit r ist und die n_i Zeiger auf die Wurzeln der Komponentengraphen sind.
- Anwendungsknoten (AP $n_1\ n_2$), wobei n_1 auf den Funktionsgraphen und n_2 auf den Argumentgraphen zeigt.
- Funktionsknoten (FUN $f\ n$), wobei f ein Kombinatorname und n die Anzahl der Argumente von f ist.
- Leerknoten (HOLE), die wir zur Konstruktion zyklischer Graphen benötigen.

Der Graph ist ein logisches Konzept, das durch einen *Heap* implementiert wird. Ein Knoten des Graphen belegt eine Zelle des *Heap*. Die Zellen brauchen nicht unbedingt gleich groß zu sein.

Für die Ausführung von Graphtransformationen nimmt die G-Maschine den Stack S zu Hilfe, auf dem Knotenadressen gespeichert werden können. Außerdem steht ein spezieller *Werte-Stack* V zur Verfügung, der bei Datenrechnungen (Anwendung von Basisoperationen auf Basiswerte) benutzt wird. Die Komponente C enthält den noch auszuführenden *Code*; wir stellen uns C als Befehlszähler vor. Zur Organisation von rekursiven Aufrufen wird der *Dump* D für die Rettung von Stack-Inhalt und Code beim rekursiven Abstieg verwendet. Der Dump ist ein Stack von Paaren (S, C). Die Komponente O (*Output*) ist ein Ausgabeband, auf das vom Programm Zahlen und Zeichen ausgegeben werden können.

Ein Zustand der G-Maschine ist ein Tupel $\langle S,\ G,\ C,\ D,\ V,\ O\rangle$. Wir beschreiben die Operationen der G-Maschine durch Zustandsübergänge. Bevor wir dazu kommen, führen wir Schreibweisen für jede Komponente ein.

Einen Stack mit dem Eintrag n an der Spitze schreiben wir $(n.S)$, wobei S ein Stack ist. Für den leeren Stack schreiben wir $[\,]$.

Eine Code-Folge mit I als erster Instruktion schreiben wir $(I.C)$, wobei C eine Code-Folge ist. Die leere Code-Folge bezeichnen wir mit $[\,]$.

Einen Dump, dessen Eintrag an der Spitze das Paar (S, C) ist, bezeichnen wir mit $((S, C).D)$, wobei D ein Dump ist. Einen leeren Dump kennzeichnen wir wieder mit $[\,]$.

Die Notation $G[n = \text{AP } n_1\ n_2]$ steht für einen Graphen, in dem der Knoten mit der Adresse n eine Anwendung von n_1 auf n_2 ist. Die Schreibweise $G[n = G\ n']$ steht für einen Graphen, in dem der Knoten n den gleichen Inhalt wie der Knoten n' hat (wir benötigen das nur für die Beschreibung des UPDATE-Befehls).

Die leere Ausgabe bezeichnen wir mit $[\,]$, und $(O; x)$ bezeichnet die Ausgabe O, gefolgt von der Ausgabe x.

4.1 Befehle zur Steuerung der Reduktionen

Am Anfang jedes Programms steht der Befehl BEGIN, der die Initialisierung des Systems vornimmt.

$$\langle S, G, (\text{BEGIN } \mathit{Prog}.C), D, V, [\,]\rangle \longrightarrow \langle (n.[\,]), [n = \text{FUN } \mathit{Prog}\ 0], C, [\,], [\,], [\,]\rangle$$

Unser Graph besteht jetzt aus nur einem Funktionsknoten, wobei der Körper des Kombinators *Prog* der Ausdruck ist, dessen Wert vom Programm berechnet werden soll. Danach starten wir mit dem EVAL-Befehl die Auswertung unseres Ausdrucks. Die Zustandsübergänge für EVAL richten sich nach dem Objekt, auf

das der Zeiger an der Stack-Spitze verweist.

$$\langle (v.S), G[v = \text{AP } v'\ n], (\text{EVAL}.C), D, V, O\rangle$$
$$\longrightarrow \quad \langle (v.[\,]), G[v = \text{AP } v'\ n], (\text{UNWIND}.C), ((S, C).D), V, O\rangle$$

$$\langle (n.S), G[n = \text{FUN } f\ 0], (\text{EVAL}.(C.\ \ldots\ .(f : .C'))), D, V, O\rangle$$
$$\longrightarrow \quad \langle (n.[\,]), G[n = \text{FUN } f\ 0], C', ((S, C).D), V, O\rangle$$

$$\langle (n.S), G[n = \text{INT } x/\text{REAL } x/\text{CHAR } x/\text{FAIL}], (\text{EVAL}.C), D, V, O\rangle$$
$$\longrightarrow \quad \langle (n.S), G[n = \text{INT } x/\text{REAL } x/\text{CHAR } x/\text{FAIL}], C, D, V, O\rangle$$

$$\langle (n.S), G[n = \text{STRUCT } k\ n_1\ \ldots\ n_r], (\text{EVAL}.C), D, V, O\rangle$$
$$\longrightarrow \quad \langle (n.S), G[n = \text{STRUCT } k\ n_1\ \ldots\ n_r], C, D, V, O\rangle$$

$$\langle (n.S), G[n = \text{FUN } f\ k], (\text{EVAL}.C), D, V, O\rangle \qquad \text{mit} \quad k > 0$$
$$\longrightarrow \quad \langle (n.S), G[n = \text{FUN } f\ k], C, D, V, O\rangle$$

Die letzten drei Regeln sagen aus, daß EVAL nichts tut, wenn das oberste Stack-Element auf ein Datenobjekt, einen FAIL-Knoten bzw. einen Funktionsknoten, der noch Argumente benötigt, zeigt.

Der UNWIND-Befehl initiiert eine neue Reduktion, indem das Rückgrat des Graphen entfaltet und die an der Spitze gefundene Funktion aktiviert wird, wenn genügend Argumente vorhanden sind. Befindet sich der Graph bereits in schwacher Kopf-Normalform, stellt UNWIND den im Dump geretteten Stack und Code wieder zurück.

$$\langle (n.[\,]), G[n = \text{INT/REAL/CHAR } x/\text{FAIL}], (\text{UNWIND}.C'), ((S, C).D), V, O\rangle$$
$$\longrightarrow \quad \langle (n.S), G[n = \text{INT/REAL/CHAR } x/\text{FAIL}], C, D, V, O\rangle$$

$$\langle (n.[\,]), G[n = \text{STRUCT } k\ n_1\ \ldots\ n_r], (\text{UNWIND}.C'), ((S, C).D), V, O\rangle$$
$$\longrightarrow \quad \langle (n.S), G[n = \text{STRUCT } k\ n_1\ \ldots\ n_r], C, D, V, O\rangle$$

$$\langle (v.S), G[v = \text{AP } v'\ n], (\text{UNWIND}.C'), D, V, O\rangle$$
$$\longrightarrow \quad \langle (v'.v.S), G[v = \text{AP } v'\ n], (\text{UNWIND}.C'), D, V, O\rangle$$

$$\langle (v_0.v_1.\ \ldots\ .v_k.S), G\left[\begin{array}{lcll} v_0 & = & \text{FUN } f\ k & \\ v_i & = & \text{AP } v_{i-1}\ n_i, & (1 \le i \le k)\end{array}\right],$$
$$(\text{UNWIND}.(C.\ \ldots\ .(f : .C'))), D, V, O\rangle$$

$$\longrightarrow \quad \langle (n_1.n_2.\ \ldots\ .n_k.v_k.S), G\left[\begin{array}{lcll} v_0 & = & \text{FUN } f\ k & \\ v_i & = & \text{AP } v_{i-1}\ n_i, & (1 \le i \le k)\end{array}\right],$$
$$C', D, V, O\rangle$$

$$\langle (v_0.v_1.\ \ldots\ .v_a.[\,]), G[v_0 = \text{FUN } f\ k], (\text{UNWIND}.C'), ((S, C).D), V, O\rangle$$
$$\text{mit} \quad a < k$$
$$\longrightarrow \quad \langle (v_a.S), G, C, D, V, O\rangle$$

Der RETURN-Befehl beendet die Ausführung einer Funktion, wenn bekannt ist, daß sich der Wert bereits in schwacher Kopf-Normalform befindet. Er stellt

den geretteten Stack und Code zurück, initiiert aber keine neue Reduktion des Graphen.

$$\begin{array}{l}\langle(v_0.v_1.\ \ldots\ .v_k.[\,]), G, (\text{RETURN}.C'), ((S,C).D), V, O\rangle \\ \longrightarrow \quad \langle(v_k.S), G, C, D, V, O\rangle\end{array}$$

Der Befehl JUMP veranlaßt einen unbedingten Sprung; bei JFAIL wird nur gesprungen, wenn der Zeiger an der Stack-Spitze auf einen FAIL-Knoten verweist.

$$\begin{array}{l}\langle S, G, (\text{JUMP } L.(C.\ \ldots\ .(L:\ .C'))), D, V, O\rangle \\ \longrightarrow \quad \langle S, G, C', D, V, O\rangle\end{array}$$

$$\begin{array}{l}\langle(n.S), G[n = n'\ \ (n' \neq \text{FAIL})], (\text{JFAIL } L.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n.S), G, C, D, V, O\rangle\end{array}$$

$$\begin{array}{l}\langle(n.S), G[n = \text{FAIL}], (\text{JFAIL } L.(C.\ \ldots\ .(L:\ .C'))), D, V, O\rangle \\ \longrightarrow \quad \langle S, G, C', D, V, O\rangle\end{array}$$

Der Befehl JFALSE betrachtet das Datenobjekt an der Spitze des Basiswert-Kellers V. Ist es gleich 0, wird gesprungen, sonst nicht. Wir wollen keinen eigenständigen Typ „*Boolean*“ einführen, sondern interpretieren die Zahl 0 als „falsch“ und eine Zahl ungleich 0 als „wahr“.

$$\begin{array}{l}\langle S, G, (\text{JFALSE } L.C), D, (x \text{ mit } x \neq 0.V), O\rangle \\ \longrightarrow \quad \langle S, G, C, D, V, O\rangle\end{array}$$

$$\begin{array}{l}\langle S, G, (\text{JFALSE } L.(C.\ \ldots\ .(L:\ .C'))), D, (0.V), O\rangle \\ \longrightarrow \quad \langle S, G, C', D, V, O\rangle\end{array}$$

Der CASEJUMP-Befehl sieht sich das Objekt an, auf das das oberste Stack-Element verweist. Dieses muß ein zusammengesetztes Datenobjekt der Art (STRUCT $k\ n_1\ \ldots\ n_r$) sein. Findet er die Konstruktor-Nummer k in der Liste aus Nummer-Marke-Paaren, bringt er Zeiger auf die Komponenten des Objekts auf den Stack und springt die entsprechende Marke an. Ist k nicht in der Liste enthalten, springt CASEJUMP zu der der Liste folgenden Marke.

$$\begin{array}{l}\langle(n.S), G[n = \text{STRUCT } k\ n_1\ n_2\ \ldots\ n_r], \\ \quad (\left(\begin{array}{l}\text{CASEJUMP } (k_1,\ L_1),\ \ldots\ ,\ (k, L^{(k)}),\ \ldots\ ,\ (k_m, L_m),\ L; \\ L_1:\ \ldots \\ \ldots \\ L^{(k)}:\ C' \\ \ldots\end{array}\right).C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_r.n_{r-1}.\ \ldots\ .n_1.S), G, C', D, V, O\rangle\end{array}$$

$$
\begin{array}{l}
\langle (n.S), G[n = \text{STRUCT } k\ n_1\ n_2\ \ldots\ n_r], \\
(\left(\begin{array}{l}
\text{CASEJUMP } (k_1, L_1), \ldots, (k_m, L_m), L; \\
L_1 : \ \ldots \\
\ldots \\
L_m : \ \ldots \\
L : \ C'
\end{array}\right) .C), D, V, O\rangle \\
\text{mit } k \neq k_1 \text{ und} \ldots \text{ und } k \neq k_m \\
\longrightarrow \quad \langle S, G, C', D, V, O\rangle
\end{array}
$$

Der Vollständigkeit halber geben wir an, daß die G-Maschine nichts tut, wenn sie auf eine Marke trifft.

$$\langle S, G, (L : .C), D, V, O\rangle \quad \longrightarrow \quad \langle S, G, C, D, V, O\rangle$$

Der PRINT-Befehl gibt den Inhalt des Knotens, auf den das Element an der Stack-Spitze zeigt, aus, wenn dieser Knoten ein einfaches Datenobjekt ist. Ist er ein zusammengesetztes Objekt, werden die Komponenten „von links nach rechts“ ausgewertet und deren Werte ausgegeben.

$$
\begin{array}{l}
\langle (n.S), G[n = \text{INT/REAL/CHAR } x], (\text{PRINT}.C), D, V, O\rangle \\
\longrightarrow \quad \langle S, G, C, D, V, (O; x)\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle (n.S), G[n = \text{STRUCT } k\ n_1\ n_2\ \ldots\ n_r], (\text{PRINT}.C), D, V, O\rangle \\
\longrightarrow \quad \langle (n_1.n_2.\ \ldots\ .n_r.S), G, (\underbrace{(\text{EVAL}; \text{PRINT})}_{r-\text{mal}} .C), D, V, O\rangle
\end{array}
$$

Der Befehl END beendet die Ausführung des G-codierten Programms.

4.2 Manipulation von Stack und Daten

Die PUSH-Befehle bringen Zeiger auf Graph-Knoten auf den Stack, POP entfernt Stack-Einträge.

$$
\begin{array}{l}
\langle (n_0.n_1.\ \ldots\ .n_k.S), G, (\text{PUSH } k.C), D, V, O\rangle \\
\longrightarrow \quad \langle (n_k.n_0.n_1.\ \ldots\ .n_k.S), G, C, D, V, O\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle S, G, (\text{PUSHINT } x/\text{PUSHREAL } x/\text{PUSHCHAR } x/\text{PUSHFAIL}.C), D, V, O\rangle \\
\longrightarrow \quad \langle (n.S), G[n = \text{INT } x/\text{REAL } x/\text{CHAR } x/\text{FAIL}], C, D, V, O\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle S, G, ((\text{PUSHFUN } f, k).C), D, V, O\rangle \\
\longrightarrow \quad \langle (n.S), G[n = \text{FUN } f\ k], C, D, V, O\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle (n_1.n_2.\ \ldots\ .n_k.S), G, (\text{POP } k.C), D, V, O\rangle \\
\longrightarrow \quad \langle S, C, D, V, O\rangle
\end{array}
$$

Der Befehl „SLIDE k“ streicht k Einträge aus dem Stack, beginnend mit dem zweiten von der Spitze aus.

$$\begin{array}{l}\langle(n_0.n_1.\ \ldots\ .n_k.S), G, (\text{SLIDE } k.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_0.S), G, C, D, V, O\rangle\end{array}$$

„SQUEEZE k, d“ streicht d Einträge aus dem Stack, beginnend beim $(k+1)$-ten.

$$\begin{array}{l}\langle(n_1.\ \ldots\ .n_k.m_1.\ \ldots\ .m_d.S), G, ((\text{SQUEEZE } k, d).C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_1.\ \ldots\ .n_k.S), G, C, D, V, O\rangle\end{array}$$

„UPDATE k“ überschreibt den Graph-Knoten, auf den der $(k+1)$-te Stack-Eintrag verweist, mit dem Inhalt des Knotens, auf den der Eintrag an der Stack-Spitze zeigt.

$$\begin{array}{l}\langle(n_0.n_1.\ \ldots\ .n_k.S), G, (\text{UPDATE } k.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_1.\ \ldots\ .n_k.S), G[n_k = G\ n_0], C, D, V, O\rangle\end{array}$$

ALLOC reserviert eine bestimmte Anzahl von Zellen auf dem *Heap*.

$$\begin{array}{l}\langle S, G, (\text{ALLOC } k.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_1.\ \ldots\ .n_k.S), G[n_1 = \text{HOLE}, \ \ldots\ , n_k = \text{HOLE}], C, D, V, O\ \rangle\end{array}$$

MKAP konstruiert einen Anwendungs-Knoten aus den beiden obersten Stack-Elementen.

$$\begin{array}{l}\langle(n_1.n_2.S), G, (\text{MKAP}.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n.S), G[n = \text{AP } n_1\ n_2], C, D, V, O\rangle\end{array}$$

Der Befehl „MKAP n“ bedeutet soviel wie eine Folge von n MKAP-Befehlen.

Der CONS-Befehl konstruiert ein zusammengesetztes Datenobjekt, das mit SELECT auseinandergenommen werden kann.

$$\begin{array}{l}\langle(n_1.n_2.\ \ldots\ .n_r.S), G, ((\text{CONS } k, r).C), D, V, O\rangle \\ \longrightarrow \quad \langle(n.S), G[n = \text{STRUCT } k\ n_1\ n_2\ \ \ldots\ \ n_r], C, D, V, O\rangle\end{array}$$

$$\begin{array}{l}\langle(n.S), G[n = \text{STRUCT } k\ n_1\ \ \ldots\ \ n_m\ \ \ldots\ \ n_r], (\text{SELECT } m.C), D, V, O\rangle \\ \longrightarrow \quad \langle(n_m.S), G, C, D, V, O\rangle\end{array}$$

4.3 Operationen auf dem V-Stack

Der Befehl GET lädt den Basiswert, auf den die Spitze des Stack zeigt, aus dem Graphen auf den Datenkeller V, auf dem die Datenrechnungen mittels der arithmetischen und logischen Befehle durchgeführt werden. „PUSHBASIC x“ legt den Basiswert x direkt auf V ab. Der Befehl MKBASIC lädt das Ergebnis der Datenrechnung zurück in einen Graph-Knoten und schreibt einen Zeiger auf

diesen Knoten auf die Spitze des Stack. UPDBASIC aktualisiert einen Graphknoten mit dem Wert an der Spitze von V.

$$\langle S, G, (\text{PUSHBASIC } x.C), D, V, O\rangle \longrightarrow \langle S, G, C, D, (x.V), O\rangle$$

$$\langle (n.S), G[n = \text{INT/REAL/CHAR } x], (\text{GET}.C), D, V, O\rangle \longrightarrow \langle S, G, C, D, (x.V), O\rangle$$

$$\langle S, G, (\text{NEG}.C), D, (x.V), O\rangle \longrightarrow \langle S, G, C, D, (-x.V), O\rangle$$

$$\langle S, G, (\text{ADD/SUB/MULT/DIV/MOD/AND/OR/ LT/LEQ/EQ/NEQ/GEQ/GT}.C), D, (x_1.x_2.V), O\rangle \longrightarrow \langle S, G, C, D, (x_1 \textbf{ op } x_2.V), O\rangle$$

$$\langle S, G, (\text{NOT}.C), D, (x\ (x \neq 0).V), O\rangle \longrightarrow \langle S, G, C, D, (0.V), O\rangle$$

$$\langle S, G, (\text{NOT}.C), D, (0.V), O\rangle \longrightarrow \langle S, G, C, D, (1.V), O\rangle$$

$$\langle S, G, (\text{ORD}.C), D, (\text{CHAR } x.V), O\rangle \longrightarrow \langle S, G, C, D, (\text{INT } x.V), O\rangle$$

$$\langle S, G, (\text{CHR}.C), D, (\text{INT } x.V), O\rangle \longrightarrow \langle S, G, C, D, (\text{CHAR } x.V), O\rangle$$

$$\langle S, G, (\text{MKBASIC}.C), D, (x.V), O\rangle \longrightarrow \langle (n.S), G[n = \text{INT/REAL/CHAR } x], C, D, V, O\rangle$$

$$\langle (n_0.n_1.\ \ldots\ .n_k.S), G, (\text{UPDBASIC } k.C), D, (x.V), O\rangle \longrightarrow \langle (n_0.n_1.\ \ldots\ .n_k.S), G[n_k \leftarrow \text{INT/REAL/CHAR } x], C, D, V, O\rangle$$

Wir müssen hier beachten, daß die arithmetischen Befehle sowohl mit ganzen als auch mit reellen Zahlen arbeiten können. Der Ergebnistyp richtet sich nach den Typen der Operanden. Zeichen lassen wir nur als Operanden für Vergleiche sowie ORD zu.

4.4 Eingabeoperationen

Einem G-codierten Programm müssen die Inhalte der Standardeingabe bzw. anderer Dateien als verzögert ausgewertete Liste von Zeichen zur Verfügung gestellt werden.

Der OPEN-Befehl veranlaßt die G-Maschine, die Liste, auf die das Element an der Stack-Spitze verweist, als Name einer Datei (bzw. der Standardeingabe

bei **NIL**) zu interpretieren. Diese Datei wird zum Lesen eröffnet und ein Zeiger auf den File-Descriptor auf den Stack gebracht.

$$\langle (n.S), G[n = \text{STRUCT } \ldots], (\text{OPEN}.C), D, V, O\rangle$$
$$\longrightarrow \quad \langle (m.S), G[m = \text{Descriptor}], C, D, V, O\rangle$$

Der READ-Befehl liest aus der Datei, auf dessen Descriptor der Eintrag an der Stack-Spitze verweist, das nächste Zeichen. Ist das möglich, legt der Befehl einen Zeiger auf den Knoten (CHAR c), wobei 'c' das gelesene Zeichen ist, auf den Stack. Ansonsten wird ein Zeiger auf (FAIL) gestapelt.

$$\langle (n.S), G[n = \text{Descriptor}], (\text{READ}.C), D, V, O\rangle$$
Gelesenes Zeichen: 'c'
$$\longrightarrow \quad \langle (m.S), G[m = \text{CHAR } c], C, D, V, O\rangle$$

$$\langle (n.S), G[n = \text{Descriptor}], (\text{READ}.C), D, V, O\rangle$$
Lesen nicht erfolgreich
$$\longrightarrow \quad \langle (m.S), G[m = \text{FAIL}], C, D, V, O\rangle$$

4.5 Aufbau der Ausgabedatei des Compilers

In der Ausgabedatei sind die G-Maschinenbefehle nicht durch Semikolons getrennt; jeder Befehl steht in einer Zeile. Namen für Marken (ein Buchstabe, gefolgt von Buchstaben bzw. Ziffern) beginnen in der ersten Spalte, Befehlcodes nicht (hier stehen Leerzeichen am Anfang). Leerzeilen sind erlaubt. Ein Prozentzeichen kennzeichnet den Beginn eines Kommentars, der sich bis zum Zeilenende erstreckt. Kommentare enthalten Informationen für einen eventuellen G-Code-Debugger und dienen zum Testen des Compilers.

Hier ist als Beispiel der aus dem Programmstück

letrec
from = **lambda** n . **construct** (CONS, n, (from ((incr n) 2))) **end**;
incr = **lambda** $n\ m$. ((add n) m) **end**;
...
in ...

erzeugte Code angegeben.

```
...
i6:
% Combinator (1 arg). Original: "from" (1 arg).
% Argument: n
    PUSHINT   2
    PUSH      1             % n
    PUSHFUN   i7, 2         % incr
```

```
    MKAP      2
    PUSHFUN   i6, 1         % from
    MKAP      1
    PUSH      1             % n
    CONS      2, 2          % CONS
    UPDATE    2
    POP       1
    UNWIND

i7:
% Combinator (2 args). Original: "incr" (2 args).
% Arguments: n m
    PUSH      1             % m
    EVAL
    GET
    PUSH      0             % n
    EVAL
    GET
    ADD
    UPDBASIC  2
    POP       2
    RETURN
...
```

Kapitel 5

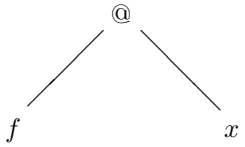
Syntaxanalyse und interne Repräsentation

Beginnend mit diesem Kapitel beschreiben wir die Realisierung unseres Compilers „ELCOM“. Als erstes betrachten wir den Syntaxanalyseprozeß und die Überführung des Quellprogramms in eine interne Repräsentation.

5.1 Entwurf der internen Programm-Repräsentation

Bevor wir den Syntaxanalysator realisieren können, müssen wir wissen, in welche Datenstruktur der Quell-Zeichenstrom übertragen werden soll. Diese Datenstruktur bildet — wie bereits gesagt — den Ausgangspunkt für die eigentliche Übersetzung und sollte so geartet sein, daß sich daraus jederzeit das Quellprogramm in druckbarer Form rekonstruieren läßt.

In der Literatur [Peyton Jones 1987, Joy 1988] ist die interne Darstellung des erweiterten Lambda-Kalküls als Graph mit verschiedenen Knotenarten nach dem folgenden Schema angegeben.



Diese Repräsentation werden wir im wesentlichen übernehmen. Die Knoten implementieren wir als *C*-Strukturen, in denen ein Feld die Knotenart festlegt. Wir wollen mit möglichst wenig verschiedenen Strukturtypen auskommen, da

sich dann die einzelnen Transformationen effizienter in einer getypten Programmiersprache implementieren lassen.

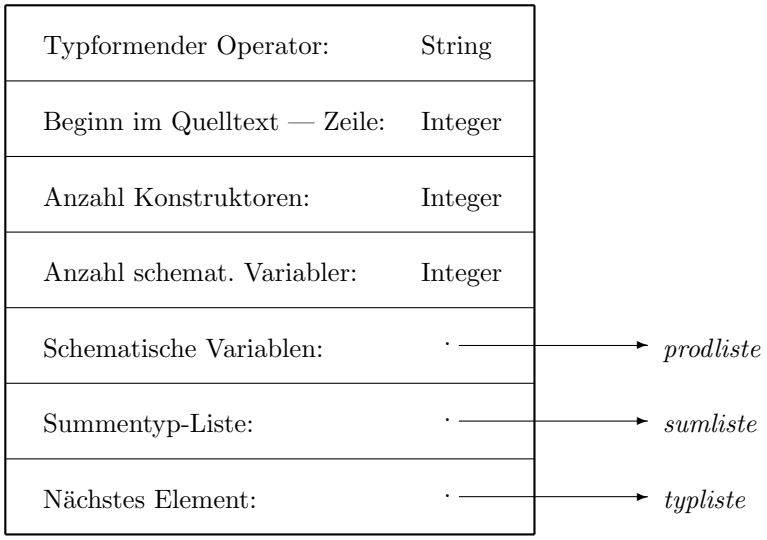
Weil wir keine Reduktionen auf unseren Ausdrucksgraphen ausführen müssen, kommen wir hier mit Bäumen aus, die nach dem Syntaxanalyseprozeß die Syntaxbäume unserer Quellprogramme darstellen. Programme bestehen im allgemeinen aus einem Typvereinbarungsteil und einem auszuwertenden Ausdruck. Wir brauchen also für deren Repräsentation zwei Datenstrukturen.

5.1.1 Repräsentation der Typvereinbarungen

Alle Typdefinitionen repräsentieren wir in einer Typliste: In jedem Element ist die Vereinbarung eines strukturierten Typs eingetragen. Vereinbarungen von Synonymen für andere Typen wie z. B. von "*string*" in

```
type  list *a  =  ...
      string   =  list char  end
```

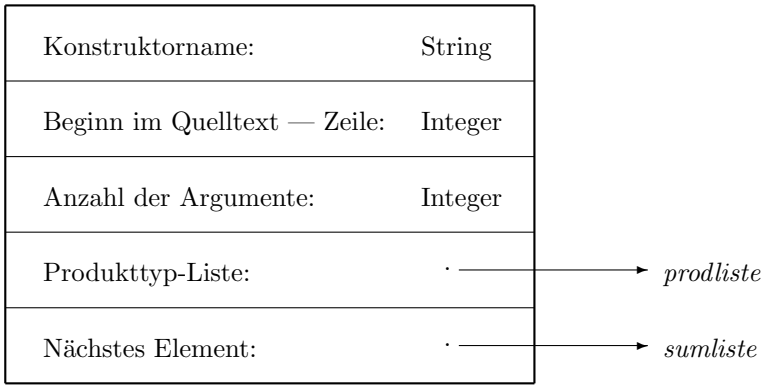
sind verboten. Ein Element der Typliste („*typliste*“) hat den folgenden Aufbau.



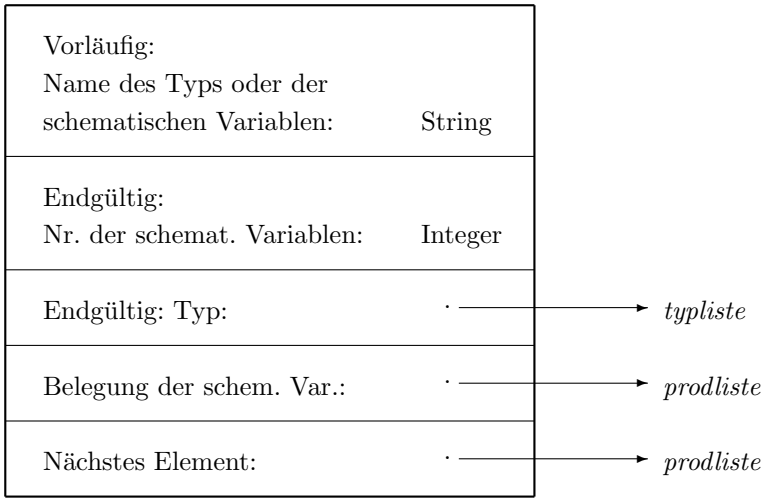
Die Zeilennummern haben wir in die Datenstruktur aufgenommen, um sie in eventuellen späteren Fehlernachrichten mit auszugeben. Für die Repräsentation der Liste der schematischen Variablen benutzen wir dieselbe Datenstruktur wie für die Argumente der Konstruktoren.

Jede Vereinbarung eines strukturierten Typs führt eine Reihe von Summentyp-Konstruktoren ein. Produkttypen betrachten wir als Summentypen mit nur einem Konstruktor. Die zu einer Typdefinition gehörenden Einzeltypen

sind in der Summentypliste aufgeführt, deren Elemente („*sumliste*“) wie folgt aufgebaut sind.



Die Argumente eines Konstruktors (die wir zusammen mit dem Konstruktor als Produkttyp auffassen können) repräsentieren wir in einer Produkttypliste. Ein Element dieser Liste („*prodliste*“) beinhaltet entweder den Bezeichner eines Standardtyps, eine schematische Variable oder einen Typbezeichner, dessen schematische Variablen durch die des zu definierenden Typs oder einen Typ belegt sind. An Hand des Aufbaus der Namen können wir schematische Variablen von Typbezeichnern unterscheiden.



Unmittelbar nach dem Syntaxanalyseprozeß haben die Produkttyp-Listenelemente erst eine vorläufige Form: Die beiden Objekte „Nummer der

schematischen Variablen“ und „Typ“ sind noch nicht belegt. Mittels einer Durchmusterung der Typliste können wir sie wie folgt belegen (in jedem Listenelement hat immer nur eine dieser beiden Komponenten einen „ordentlichen“ Wert).

Angenommen, unser Element bezeichnet eine schematische Variable. Da die schematischen Variablen eines typformenden Operators lokal in dessen Definition sind, reicht es aus, die Nummer der schematischen Variablen zu kennen (wir können aber den Namen für Debugging-Zwecke aufheben). Die schematischen Variablen sind so numeriert.

type	⟨typformender-Operator⟩	$*a$	$*b$	$*c$	...
		|	|	|	
		1	2	3	...

Weil wir außerdem die Anzahl der schematischen Variablen des typformenden Operators wissen, können wir, wenn wir mit der Behandlung einer Typvereinbarung fertig sind, die Liste der schematischen Variablen abhängen und den Speicherplatz wieder freigeben.

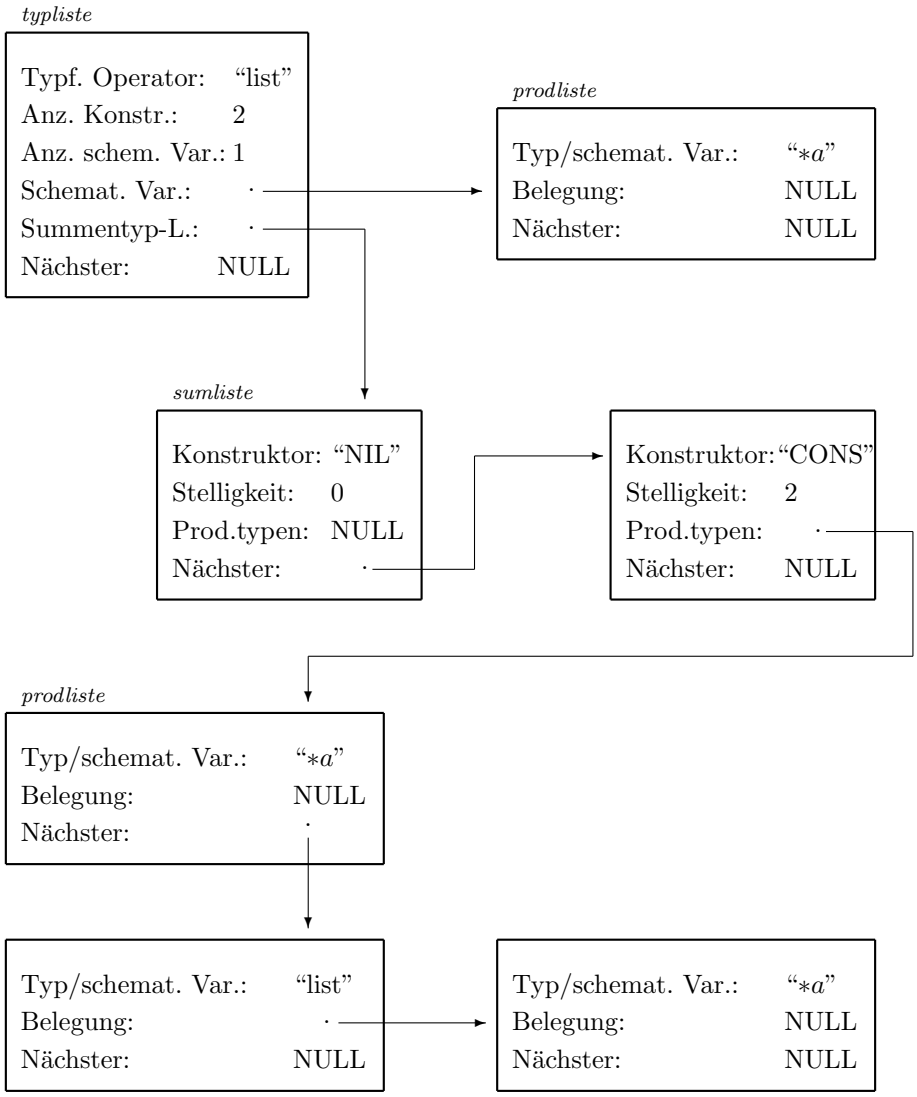
Wenn unser Listenelement dagegen den Namen eines Typs enthält, speichern wir statt dieses Namens einen Zeiger auf die Definition dieses Typs. Dadurch ersparen wir uns später, in der Typliste zu suchen, wenn wir diese Definition brauchen.

Die Typliste in der endgültigen Form benötigen wir nur für die Typinferenz, die wir in dieser Arbeit nicht ausführen wollen. Daher wird die endgültige Typliste nicht erzeugt. Wir überprüfen nur, ob sich alle definierten Konstruktoren voneinander unterscheiden. Das geschieht mit der Funktion „`ueberpruefe`“ in der **Yacc**-Eingabedatei “`el_syn.y`” (siehe Anhang). Die Operationen zum Aufbau der endgültigen Typliste können einfach ans Ende dieser Funktion angefügt werden.

Wir wollen die Verwendung der einzelnen Strukturen in der Typliste an einem Beispiel verdeutlichen. Angenommen, unser Compiler trifft auf die folgende Typvereinbarung.

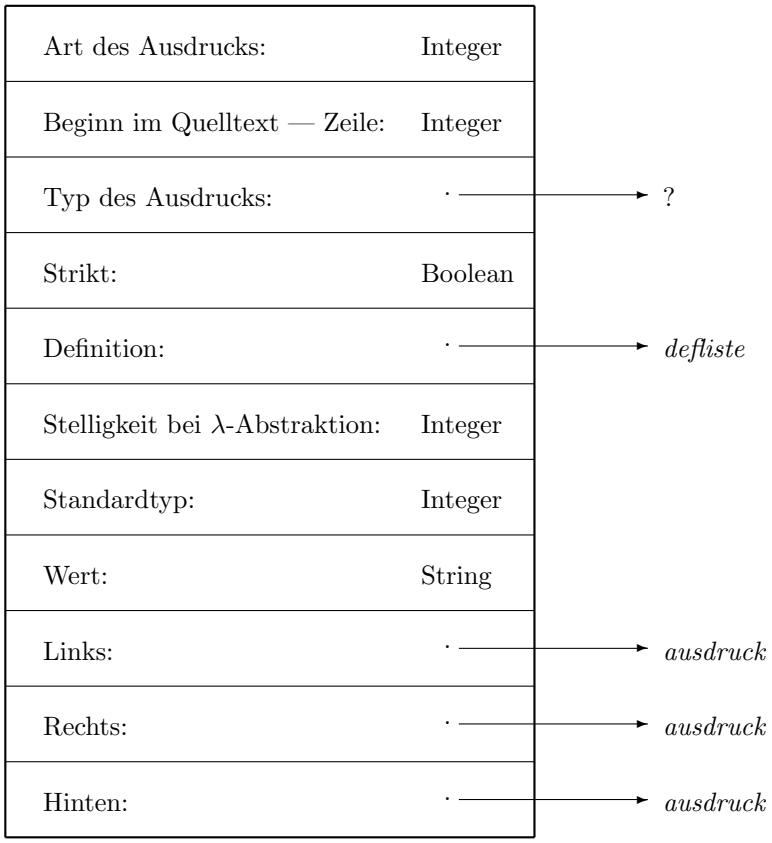
type list $*a$ = NIL | CONS $*a$ (list $*a$) **end**

Während der Syntaxanalyse erzeugt er daraus die nachstehende interne Repräsentation.



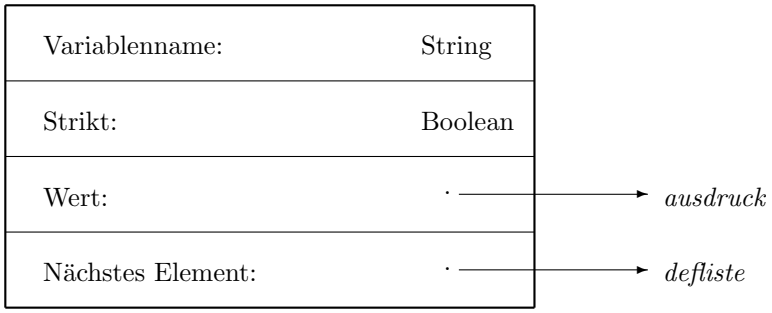
5.1.2 Repräsentation der Ausdrücke

Einen Ausdruck stellen wir durch die Datenstruktur „*ausdruck*“ dar, deren Aufbau im folgenden beschrieben ist.



Den Typ des Zeigers „Typ des Ausdrucks“ haben wir noch nicht festgelegt, weil der Zeiger nur für die Typinferenz gebraucht wird.

Ein Element der Definitionsliste („*defliste*“) hat das folgende Aussehen.



Die Variable „Strikt“ ist bei Anwendungen bzw. Lambda-Abstraktionen gleich 1, wenn das Argument in jedem Falle ausgewertet werden muß; ansonsten ist sie gleich 0.

Wir sehen uns jetzt die Repräsentation jeder einzelnen Ausdrucksart an je einem Beispiel an.

Konstante, z. B. “2.53”

Ausdrucksart:	Konstante
Definition:	NULL
Standardtyp:	Real
Wert:	“2.53”
Links:	NULL
Rechts:	NULL
Hinten:	...

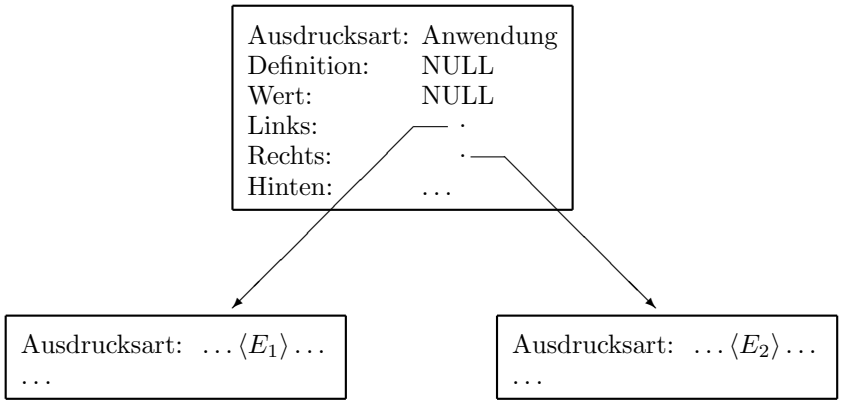
Variable, z. B. “from”

Ausdrucksart:	Variable
Definition:	NULL
Standardtyp:	0
Wert:	“from”
Links:	NULL
Rechts:	NULL
Hinten:	...

Der Ausdruck “FAIL”

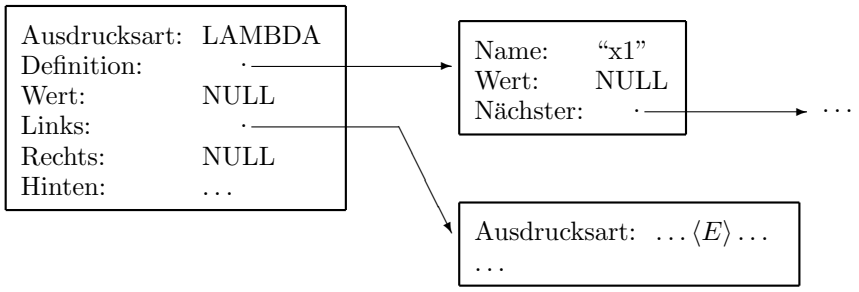
Ausdrucksart:	FAIL
Definition:	NULL
Standardtyp:	0
Wert:	NULL
Links:	NULL
Rechts:	NULL
Hinten:	...

Funktionsanwendung: $(\langle E_1 \rangle \ \langle E_2 \rangle)$

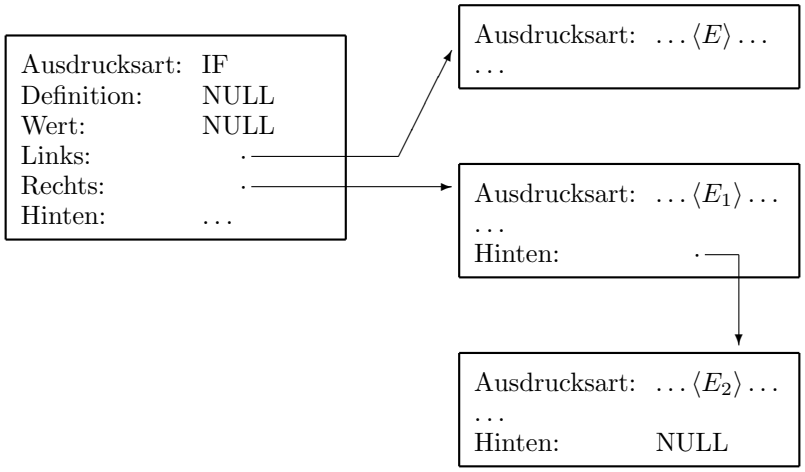


Genauso wie die Funktionsanwendungen repräsentieren wir die **fatbar**-Ausdrücke, nur daß als Ausdrucksart „FATBAR“ eingesetzt ist.

Lambda-Abstraktion: **lambda** $x_1 \ldots x_n \,.\, \langle E \rangle$ **end**



IF-Ausdruck: if $\langle E \rangle$ then $\langle E_1 \rangle$ else $\langle E_2 \rangle$ end

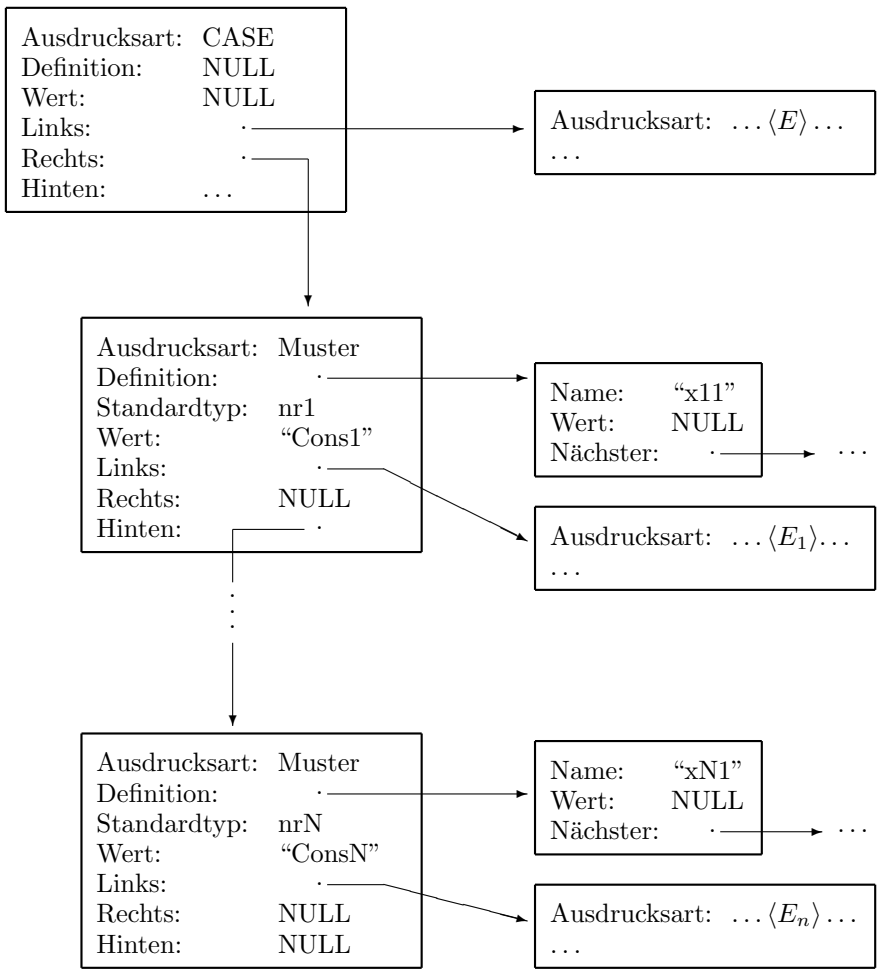


Die Argumente der Funktion "**if**" sind nicht *curried* gespeichert, da immer alle drei Argumente vorhanden sind und durch diese Darstellung eine effizientere Compilation möglich ist.

Der Zeiger „Hinten" ist bei geschachtelten **if**'s nur auf jedem zweiten Niveau belegt, daher gibt es keine Konflikte.

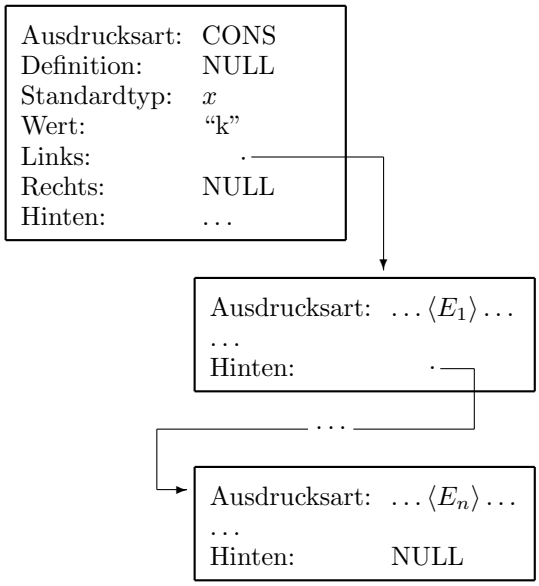
CASE-Ausdruck

$$
\begin{array}{l}
\textbf{case } \langle E \rangle \textbf{ of} \\
\quad \begin{array}{lll}
\text{Cons}_1 \ x_{1,1} \ \ldots & \Longrightarrow & \langle E_1 \rangle \, ; \\
\quad \ldots & & \\
\text{Cons}_n \ x_{n,1} \ \ldots & \Longrightarrow & \langle E_n \rangle
\end{array} \\
\textbf{end}
\end{array}
$$



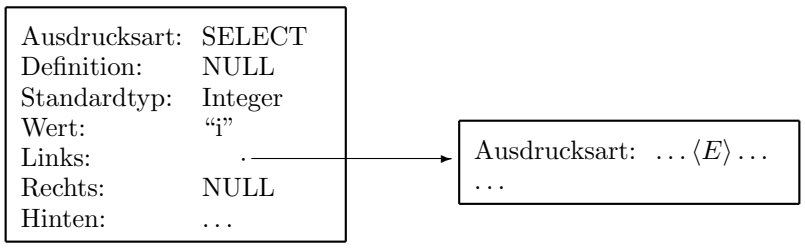
Die Werte der Komponenten „Standardtyp“ in den Mustern sind die Nummern der Konstruktoren in der entsprechenden Typvereinbarung.

CONSTRUCT-Ausdruck: construct(“k”, $\langle E_1\rangle, \ldots, \langle E_n\rangle$)

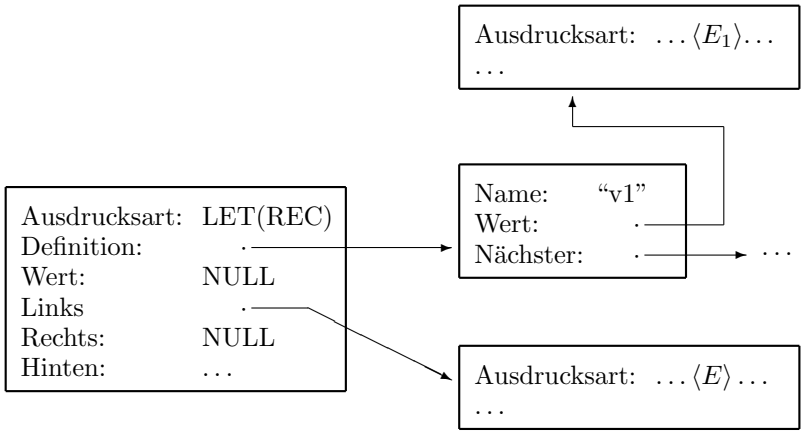


Der Wert der Komponente „Standardtyp“ ist wieder die Reihenfolge-Nummer des Konstruktors k in der entsprechenden Typvereinbarung.

SELECT-Ausdruck: select(i, $\langle E\rangle$)



LET(REC)-Ausdruck: let(rec) $v_1 \;=\; \langle E_1 \rangle \ldots$ **in** $\langle E \rangle$ **end**



Bei **let**-Ausdrücken darf die Definitionsliste nur ein Variablenname-Wert-Paar enthalten.

5.2 Der Syntaxanalysator

Unser mit **Yacc** geschriebene Syntaxanalysator hat nur die Aufgabe, die Zeichenfolge in der Quellprogramm-Datei als Satz unserer Sprache zu erkennen — oder die Syntaxfehler zu melden — und bei erfolgreicher Analyse die Typliste sowie die Repräsentation des auszuwertenden Ausdrucks zu erzeugen. Er ist daher relativ einfach gehalten.

Wir müssen die im dritten Kapitel angegebene Grammatik etwas umformen, da **Yacc** mit der erweiterten Backus-Naur-Form nichts anfangen kann. Die semantischen Aktionen bestehen einfach darin, den entsprechenden „Teilbaum“ zu konstruieren und einen Zeiger darauf dem Nichtterminal auf der linken Seite als Attribut zu übergeben. Auf diese Weise wird der Zeiger zu anderen Regeln „weitergereicht“ und kann dort in andere Datenstrukturen eingebaut werden.

Die Terminalsymbole erhält der Syntaxanalysator vom Lexikanalysator, der als **Lex**-Eingabedatei “`el_lex.l`” spezifiziert ist. Die nach dem **Lex**-en vorliegende Datei “`lex.yy.c`” lassen wir bei der Übersetzung von “`y.tab.c`” vom Präprozessor in den Quelltext einfügen.

Während der Syntaxanalyse führen wir zwei Prüfungen durch. Die erste haben wir schon bei der Behandlung der Typliste erwähnt. Die zweite testet, ob die innerhalb von **construct**- und **case**-Ausdrücken benutzten Konstruktoren überhaupt in der Typliste vorkommen. In einem **case**-Ausdruck müssen außerdem alle Konstruktoren aus derselben Typvereinbarung stammen und kein

Konstruktor darf mehrmals auftreten. Diese Prüfungen werden in den Funktionen „`b_cons`“ bzw. „`b_case`“ erledigt.

Nach der erfolgreichen Syntaxanalyse enthält die Variable „`defs`“ einen Zeiger auf die Typliste und die Variable „`prog`“ einen Zeiger auf den Ausdruck.

Sollte dieser einfache Syntaxanalysator gegen einen anderen ausgetauscht werden, ist zu beachten, daß einige Funktionen auch in anderen Quelltext-Dateien des Compilers verwendet werden.

War die Syntaxanalyse erfolgreich, kann man den Compiler dazu bringen, anzuhalten und sich in eine Kommandointerpreter-Schleife zu begeben. In dieser Schleife (Funktion „`stop_print`“ in der Datei “`el_print.c`”) können wir uns die erzeugten Datenstrukturen auf dem Bildschirm anzeigen lassen (das geschieht mittels rekursiver Durchmusterung). Dieses Stoppen (ziemlich ungewöhnlich für einen Compiler) wird später interessant, da wir alle Transformationen, die unser Compiler auf den Quellprogrammen ausführt, unmittelbar verfolgen können.

Die einzelnen Implementationsdetails entnehmen wir dem kommentierten Quelltext im Anhang.

Kapitel 6

Variablenumbenennung

Für die Ausführung der Transformationen ist es notwendig, daß kein Variablenname mehrfach vereinbart wird wie z. B. im Ausdruck.

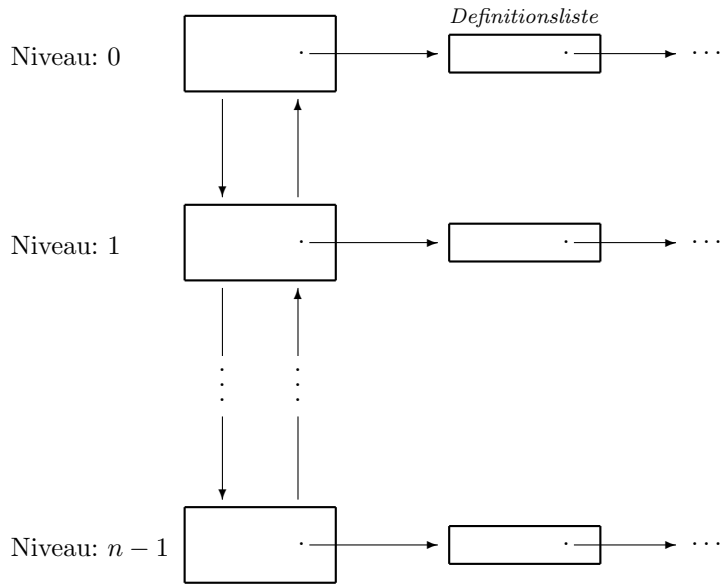
$$\begin{array}{l}\textbf{let}\ x\ =\ \ldots \\ \textbf{in} \\ \qquad \ldots\,\textbf{lambda}\ x\ \ldots\end{array}$$

Wir benennen daher alle Variablen um ("`el_scope.c`"), so daß hinterher jeder Name nur einmal gebunden ist, und sichern uns damit die folgenden Vorteile.

- Wenn wir neue Variablen einführen, können wir leicht solche Namen finden, die sich von allen im Programm bereits vorkommenden Namen unterscheiden.

- Reißen wir einen Teilausdruck aus seinem umgebenden Kontext und fügen ihn woanders wieder ein, gibt es keine Namenskonflikte.

Die Umbenennung läuft folgendermaßen ab. Wir durchmustern unseren Ausdrucksbaum rekursiv — beginnend bei der Wurzel „`prog`“ in Richtung der Blätter — und suchen nach Ausdrücken, die neue Variablen einführen und damit ein neues Bindungsniveau definieren. Die neuen Variablen befinden sich in den Definitionslisten der **let(rec)**'s, der **lambda**'s und der Muster in den **case**-Ausdrücken. Jetzt geben wir jeder Variable dieser Liste einen neuen Namen, der aus einem "i", gefolgt von einer Zahl, besteht, und merken uns den alten Namen. Die umbenannten Definitionslisten bauen wir für die Dauer der Gültigkeit der in ihr enthaltenen Vereinbarungen in eine globale Datenstruktur, die doppelt verkettete Bindungsniveau-Liste, ein. Diese Liste hat Ähnlichkeit mit den Datenstrukturen, die für die Bindung der Variablen in *Lisp* verwendet werden, und sieht nach n ineinandergeschachtelten Ausdrücken, die Variablen einführen, wie folgt aus (auf dem Wege dahin ist sie von oben nach unten gewachsen).



Treffen wir im Laufe der Durchmusterung auf Variablen, suchen wir sie in der Bindungsniveau-Liste, und zwar beginnend beim höchsten Niveau (in der Abbildung von unten nach oben). Wir hören sofort mit der Suche auf, sobald der Variablenname mit dem alten Namen in einer Definitionsliste übereinstimmt (damit sichern wir, die richtige Vereinbarung erwischt zu haben, wenn der Name mehrmals gebunden wurde). Jetzt benennen wir die Variable in den gefundenen neuen Namen um.

Bei **let(rec)**-Ausdrücken müssen wir auch die Ausdrücke der Variablenwerte umbenennen, und zwar bei **let**'s vor und bei **letrec**'s nach der Erweiterung der Bindungsniveau-Liste. Dabei erkennen wir Ausdrücke der Art

$$\begin{array}{l}\textbf{letrec}\\ \quad x \;=\; x\,;\\ \quad \ldots\\ \textbf{in}\;\; \ldots\end{array}$$

als fehlerhaft, da sie der Codegenerator nicht verarbeiten kann.

Tritt eine Variable nicht in der Bindungsniveau-Liste auf und ist sie kein vordefinierter Bezeichner, handelt es sich um eine ungebundene Variable, und wir melden einen Fehler.

Wenn wir den Gültigkeitsbereich eines Bindungsniveaus verlassen (den Ausdruck, der mit der entsprechenden Definitionsliste verbunden ist), hängen wir das Bindungsniveau-Listenelement (es muß immer das unterste sein) ab.

Bei der Umbenennung der Variablennamen in den Definitionslisten tragen wir die neuen Namen in der Hash-Tabelle ein. Ein Eintrag enthält noch weitere

Informationen, wie den alten Namen, das Bindungsniveau und (bei **let(rec)**'s) einen Zeiger auf den Wert. Da wir in den Definitionslistenelementen einen Zeiger auf den entsprechenden Hash-Tabelleneintrag speichern, haben wir einen schnellen Zugriff auf den alten Namen, der dann nicht im Definitionslistenelement stehen muß.

Die Hash-Tabelle ist ein Feld von Zeigern auf Tabelleneinträge. Beim Auftreten von Konflikten werden mehrer Einträge zu einer Liste verkettet.

Wenn wir unser Beispielprogramm aus dem dritten Kapitel als Datei "`prog.el`" eingeben und von ELCOM die Variablen umbenennen lassen, erhalten wir folgendes (die ursprünglichen Namen sind in eckige Klammern eingeschlossen).

```
type list *a = NIL | CONS *a (list *a) end

letrec
 i1[fac] = lambda i2[n].
            letrec
              i3[from] = lambda i5[n].
                          construct(CONS, i5[n],
                            (i3[from]
                             (lambda i6[n]. ((add i6[n]) 1) end
                              i5[n])))
                         end;
              i4[prod] = lambda i7[x] i8[m].
                          case i7[x] of
                          CONS i9[h] i10[t] =>
                            if ((geq i9[h]) i8[m])
                              then i9[h]
                              else ((mult i9[h])
                                    ((i4[prod] i10[t]) i8[m]))
                            end
                          end
                         end
            in
              ((i4[prod] (i3[from] 1)) i2[n])
            end
           end
in
 (i1[fac] 10)
end
```

Kapitel 7

Transformation der Anwendungen

Unser Codegenerator kann nur solche Funktionsanwendungen verarbeiten, bei denen *ganz links* nur ein Bezeichner steht; d. h. im folgenden Ausdruck muß E_1 ein Bezeichner sein.

$$((\ldots(E_1\ E_2)\ldots\ E_{n-1})\ E_n)$$

Da wir an dieser Stelle in unserer Quellsprache (fast) beliebige Ausdrücke zulassen, müssen wir sie in die gewünschte Form transformieren. Das geschieht wieder mittels rekursiver Durchmusterung durch die Funktion „`apptrans`“ in der Datei “`el_app.c`”.

Nebenbei bringen wir noch alle Lambda-Abstraktionen mit mehreren Parametern in eine einheitliche Form, da wir hierfür in unserer Quellsprache zwei völlig äquivalente Formen haben, nämlich z. B.

lambda x **. lambda** y **.** $\ldots$ **end end**

und

lambda $x\ y\ \ldots\ldots\ldots$ **end**

Wir transformieren alle **lambda**'s der ersten Form in die zweite (Funktion „`translambda`“), die (bei unserer Repräsentation) weniger Speicherplatz verbraucht. Das geht ohne weiteres, da nach der Variablenumbenennung alle vereinbarten Variablen unterschiedliche Namen haben (d. h. es kann nie “x” = “y” im Sinne der Namensgleichheit sein).

Wir betrachten jetzt nacheinander alle Ausdrücke, die wir (außer Variablen) an der Position von E_1 finden können.

7.1 Lambda-Abstraktion

Wenn wir ein **lambda** finden, führen wir eine β-Reduktion aus. Dabei substituieren wir die Vorkommen des entsprechenden Parameters nicht durch das Argument, weil wir dann unnötige Auswertungen riskieren, sondern führen einen **let**-Ausdruck ein, der die früher λ-gebundene Variable bindet. Wir betrachten hier drei Fälle.

1. Dem **lambda** stehen weniger Argumente zur Verfügung als es Parameter besitzt.

$$
\begin{array}{l}
(\ldots(\textbf{lambda}\ x_1\ \ldots\ x_n\,.\,E\ \textbf{end}\ \ E_1)\ \ldots\ \ E_m) \\
\text{mit}\quad n > m \quad \Longrightarrow \\
\textbf{let}\ x_1 = E_1 \\
\qquad \textbf{in let}\ x_2 = E_2 \\
\qquad\qquad \textbf{in}\ \ldots \\
\qquad\qquad\qquad \textbf{in let}\ x_m = E_m \\
\qquad\qquad\qquad\qquad \textbf{in lambda}\ x_{m+1}\ \ldots\ x_n\,.\,E\ \textbf{end} \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad \ldots \\
\textbf{end}
\end{array}
$$

2. Die Anzahl der im **lambda** gebundenen Variablen stimmt mit der Argumentanzahl überein.

$$
\begin{array}{l}
(\ldots(\textbf{lambda}\ x_1\ \ldots\ x_n\,.\,E\ \textbf{end}\ \ E_1)\ \ldots\ \ E_n) \\
\Longrightarrow \\
\textbf{let}\ x_1 = E_1 \\
\qquad \textbf{in let}\ x_2 = E_2 \\
\qquad\qquad \textbf{in}\ \ldots \\
\qquad\qquad\qquad \textbf{in let}\ x_n = E_n \\
\qquad\qquad\qquad\qquad \textbf{in}\ E \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad \ldots \\
\textbf{end}
\end{array}
$$

3. Es gibt mehr Argumente als das **lambda** Parameter hat.

$$
\begin{array}{l}
(\ldots(\textbf{lambda}\ x_1\ \ldots\ x_n\,.\,E\ \textbf{end}\ \ E_1)\ \ldots\ \ E_r) \\
\text{mit}\quad n < r \quad \Longrightarrow \\
\textbf{let}\ x_1 = E_1 \\
\quad \textbf{in let}\ x_2 = E_2 \\
\qquad \textbf{in}\ \ldots \\
\qquad\quad \textbf{in let}\ x_n = E_n \\
\qquad\qquad \textbf{in}\ (\ldots(E\ E_{n+1})\ \ldots\ E_r) \\
\qquad\qquad \textbf{end} \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

Nach diesen Transformationen sind einige **lambda**'s ganz verschwunden, so daß wir diese im nächsten Kapitel nicht mehr zu liften brauchen.

7.2 Let(rec)-Ausdruck

Da alle vereinbarten Variablen verschiedene Namen tragen, ist es legitim, den Gültigkeitsbereich eines **let(rec)**-Ausdrucks zu erweitern, ohne daß Namenskonflikte auftreten.

$$
\begin{array}{l}
(\ldots(\textbf{let(rec)}\ \ldots\ \textbf{in}\ E\ \textbf{end}\ \ E_1)\ \ldots\ \ E_n) \\
\Longrightarrow \\
\textbf{let(rec)}\ \ldots\ \textbf{in}\ (\ldots(E\ E_1)\ldots\ E_n)\ \textbf{end}
\end{array}
$$

7.3 Case-, if-, fatbar- bzw. select-Ausdruck

Aus dem gleichen Grund wie bei den **let(rec)**'s können wir bei den **case**-Ausdrücken die folgende Transformation ausführen.

$$
\begin{array}{l}
(\ldots(\quad \textbf{case } E \textbf{ of} \\
\qquad\quad \ldots \Rightarrow E_{c1}\,; \\
\qquad\quad \ldots \\
\qquad\quad \ldots \Rightarrow E_{cn} \\
\qquad\quad \textbf{end} \qquad\qquad\qquad\qquad E_1) \ldots\ E_m) \\
\Longrightarrow \\
\textbf{let} \\
\qquad x = \textbf{lambda}. \\
\qquad\quad \textbf{case } E \textbf{ of} \\
\qquad\quad\ \ldots \Rightarrow E_{c1}\,; \\
\qquad\quad\ \ldots \\
\qquad\quad\ \ldots \Rightarrow E_{cn} \\
\qquad\quad \textbf{end} \\
\qquad\quad \textbf{end} \\
\textbf{in} \\
\qquad (\ldots(x\ E_1) \ldots\ E_m) \\
\textbf{end}
\end{array}
$$

Mit dem Konstrukt „**lambda . ... end**" definieren wir eine parameterlose Funktion. Wir führen hier deshalb eine Funktion ein, damit wir sie im nächsten Kapitel liften können. x ist eine neue Variable, die im Programm noch nicht vorkommt. Da der Wert von x im Ausdruck nur einmal verwendet wird, riskieren wir keine unnötigen Reduktionen.

If-, **fatbar-** und **select**-Ausdrücke transformieren wir analog.

$$
\begin{array}{l}
(\ldots(\textbf{if } E \textbf{ then } F \textbf{ else } G \textbf{ end}\ E_1) \ldots\ E_m) \\
\Longrightarrow \\
\textbf{let } x\ =\ \textbf{lambda . if } E \textbf{ then } F \textbf{ else } G \textbf{ end end} \\
\textbf{in } (\ldots(x\ E_1) \ldots\ E_m) \textbf{ end}
\end{array}
$$

$$
\begin{array}{l}
(\ldots(\textbf{fatbar}(E\ F)\ E_1) \ldots\ E_m) \\
\Longrightarrow \\
\textbf{let } x\ =\ \textbf{lambda . fatbar}(E\ F) \textbf{ end} \\
\textbf{in } (\ldots(x\ E_1) \ldots\ E_m) \textbf{ end}
\end{array}
$$

$$
\begin{array}{l}
(\ldots(\textbf{select}(i,\ E)\ E_1) \ldots\ E_m) \\
\Longrightarrow \\
\textbf{let } x = \textbf{lambda . select}(i,\ E) \textbf{ end} \\
\textbf{in } (\ldots(x\ E_1) \ldots\ E_m) \textbf{ end}
\end{array}
$$

7.4 Beseitigung unnötiger let-Ausdrücke

Die Teilausdrücke in unseren transformierten Ausdrücken können wieder Funktionsanwendungen sein. Daher müssen sich auch alle Teilausdrücke der beschriebenen Prozedur unterziehen.

Waren einige Argumente der transformierten **lambda**'s nur Variablen, erzeugt ELCOM redundante Ausdrücke der Form

$$\textbf{let } x = y \textbf{ in } E \textbf{ end}$$

Diese Ausdrücke beseitigen wir durch Ausführen der entsprechenden Substitution in E (Funktion „`elim`“). Finden wir dabei auch (vom Programmierer eingebaute) **letrec**-Ausdrücke dieser Art, müssen wir auch in der Definitionsliste substituieren.

Abschließend berechnen wir mit der Funktion „`neuniv`“ für alle Variablenvereinbarungen die Bindungsniveaus neu, da wir sie im nächsten Kapitel brauchen.

Unser Beispielprogramm hat nach der Transformation der Funktionsanwendungen die folgende Gestalt.

```
type list *a = NIL | CONS *a (list *a) end

letrec
 i1[fac] = lambda i2[n].
             letrec
               i3[from] = lambda i5[n].
                           construct(CONS, i5[n],
                            (i3[from] ((add i5[n]) 1)))
                          end;
               i4[prod] = lambda i7[x] i8[m].
                           case i7[x] of
                           CONS i9[h] i10[t] =>
                             if ((geq i9[h]) i8[m])
                               then i9[h]
                               else ((mult i9[h])
                                      ((i4[prod] i10[t]) i8[m]))
                             end
                           end
                          end
             in
               ((i4[prod] (i3[from] 1)) i2[n])
             end
           end
in (i1[fac] 10)
end
```

Kapitel 8

Lambda-Lifting

Wenn wir den Code für ein funktionales Programm erzeugen, setzen wir voraus, daß es aus einer Menge von (möglicherweise wechselseitig rekursiven) Funktionsdefinitionen und dem auszuwertenden Ausdruck besteht. Das Programm hat demnach die Form

$$\begin{array}{lcl} f_1\ x_1\ \ldots x_{n_1} & = & E_1 \\ \quad\ldots & & \\ f_m\ x_1\ \ldots x_{n_m} & = & E_m \\ E_{Prog} & & \end{array}$$

wobei in den Funktionskörpern E_i keine Lambda-Abstraktionen, wohl aber **let(rec)**-Ausdrücke auftreten dürfen. Eine Funktionsdefinition

$$f\ x_1\ \ldots\ x_n \quad = \quad E$$

übersetzen wir in eine feste Code-Folge, die den Graphen der Anwendung dieser Funktion durch den Wert der rechten Seite ersetzt, wobei die Parameter x_i durch die Argumente E_{ai} substituiert sind.

$$(\ldots(f\ E_{a1})\ldots\ E_{an}) \quad \Longrightarrow \quad E\,[E_{a1}/x_1,\ \ldots\ ,\ E_{an}/x_n]$$

Da wir für jede Funktion eine *feste* Code-Folge erzeugen wollen, darf ihr Körper keine freien Variablen enthalten.

Die Technik des *Lambda-Lifting* macht die freien Variablen jeder Lambda-Abstraktion zu zusätzlichen Argumenten und hebt dann die Funktionsdefinition auf das globale Niveau. Da diese Funktionen jetzt keine freien Variablen und auch keine inneren Lambda-Abstraktionen mehr besitzen, bezeichnen wir sie als *Superkombinatoren*. Die Namen der Superkombinatoren betrachten wir nicht als freie Variablen, sondern als Konstanten, so daß die Superkombinatoren (wechselseitig) rekursiv sein dürfen.

Die Superkombinator-Form erhalten wir, indem wir unseren Ausdruck rekursiv durchmustern (Funktion „`lift`“ in der Datei “`el_lift.c`”) und die dort gefundenen Lambda-Abstraktionen sowie die **let**- und **letrec**-Ausdrücke dem Lifting unterziehen. Wir wollen uns diese drei Fälle nacheinander ansehen.

8.1 Liften der Lambda-Abstraktionen

In diesem Abschnitt liften wir die anonymen **lambda**'s. Lambda-Abstraktionen, die als rechte Seite einer Variablenvereinbarung in **let(rec)**-Ausdrücken vorkommen — also einen Namen haben —, behandeln wir an dieser Stelle nicht, da wir sonst bei der Abarbeitung des Programms mehr Reduktionen ausführen müßten. Betrachten wir dazu das Beispiel

let $a = 5$
in let $f =$ **lambda** x . ((add x) a) **end**
  in (f 3)
  end
end

Liften wir stur alle **lambda**'s, erhalten wir

combinators
  \$f y x = ((add x) y)
end

let $a = 5$
in let $f =$ (\$f a)
  in (f 3)
  end
end

Mit dem Konstrukt „**combinators** ⟨Definitionen⟩ **end**“ kennzeichnen wir die gelifteten Superkombinatoren. Bei der Reduktion des Ausdrucks (f 3) erhalten wir zunächst ((\$f a) 3), der dann weiter reduziert wird.

Liften wir dagegen die gesamte Vereinbarung von f und substituieren im Unterausdruck jedes Auftreten von f gegen den Namen des erzeugten Superkombinators, angewendet auf die freien Variablen, erhalten wir

combinators
  \$f y x = ((add x) y)
end

let $a = 5$
in ((\$f a) 3)
end

und sparen die Reduktion eines **let**-Ausdrucks (der jetzt verschwunden ist) sowie die Reduktion von $(f\ 3)$ aus der vorigen Variante. Trat f mehrfach im Unterausdruck auf, erzeugen wir einen längeren Code als in der ersten Variante, das Programm ist aber schneller.

Für das Liften eines anonymen **lambda**'s bestimmen wir zuerst die Liste seiner freien Variablen, in der die Variablen nach dem textlichen Bindungsniveau sortiert sind — die auf dem innersten Niveau gebundenen Variablen stehen hinten.[1]

Danach benennen wir die freien Variablen um (wobei wir die entsprechenden Substitutionen im Lambda-Körper vornehmen) und machen sie zu zusätzlichen Argumenten. Dem so erhaltenen Superkombinator geben wir einen Namen und fügen ihn ans Ende der Kombinator-Liste (auf die der Zeiger „`comb`“ verweist) an.

Das Vorkommen des nun gelifteten **lambda**'s im Ausdruck ersetzen wir durch die *curried* dargestellte Anwendung des neu gebildeten Superkombinators auf die vorher freien Variablen. Weil im Inneren des Superkombinators auch Lambda-Abstraktionen und **let(rec)**-Ausdrücke stehen dürfen, rufen wir die „`lift`“-Funktion rekursiv für den Körper auf — erst dann trägt der Superkombinator diesen Namen zu Recht.

Zur Verdeutlichung des Gesagten betrachten wir das folgende Beispiel.

```
let
     i1[a] = 7
in let
        i2[b] = 3
   in lambda i3[x]. ((add ((sub i3[x]) i2[b])) i1[a]) end
   end
end
```

Diesen Ausdruck transformieren wir in den folgenden.

```
combinators
    i6[--] i4[--] i5[--] i3[x] =
       ((add ((sub i3[x]) i5[--])) i4[--])
end

let
     i1[a] = 7
in let
        i2[b] = 3
   in ((i6[--] i1[a]) i2[b])
   end
end
```

[1] Das ermöglicht die Elimination von redundanten Superkombinatoren durch η-Reduktion, die wir jedoch nicht ausführen werden.

8.2 Transformation der let-Ausdrücke

Treffen wir einen **let**-Ausdruck, sehen wir nach, ob der Wert seiner gebundenen Variablen eine Lambda-Abstraktion ist. Wenn nicht, bearbeiten wir diesen Wert sowie den Unterausdruck und sind fertig.

Ansonsten bestimmen wir wieder die freien Variablen des **lambda**'s, benennen sie um und machen sie zu neuen Argumenten. Um eine einheitliche Behandlung aller **lambda**'s zu sichern, geben wir der Lambda-Abstraktion auch einen neuen Namen, bevor wir sie zum Superkombinator erheben. Anschließend substituieren wir im Unterausdruck jedes Auftreten des alten Namens durch die Anwendung des Superkombinators auf die früher freien Variablen und werfen die alte Definitionsliste des **let**-Ausdrucks weg. Das **let** mit der leeren Definitionsliste lassen wir erst einmal stehen, da wir es nach dem Lambda-Lifting mit Hilfe der uns schon bekannten Funktion „`elim`“ beseitigen können.

8.3 Transformation der letrec-Ausdrücke

Der komplizierteste Fall ist der **letrec**-Ausdruck, da hier mehrere Variablenvereinbarungen möglich sind, die noch dazu direkt oder indirekt rekursiv sein können.

Als erstes teilen wir die Definitionsliste in solche Vereinbarungen, die **lambda**'s zum Wert haben („`lambdas`“), und andere Vereinbarungen („`vars`“) auf. Bleiben nur Definitionen der letzteren Art übrig, tun wir nichts weiter, als die Werte der Variablen und den Unterausdruck der Lifting-Prozedur zu unterwerfen.

Anderenfalls setzen wir die Definitionsliste des **letrec** auf „`vars`“ und bestimmen die Menge der freien Variablen jeder Lambda-Abstraktion aus „`lambdas`“. Dabei erkennen wir keine Variable aus „`lambdas`“ als frei. Kommen in einem der Lambda-Körper der Definitionen aus „`lambdas`“ irgendwelche Variablen aus „`lambdas`“ vor, werden sie im weiteren Verlauf durch die Anwendungen des daraus erzeugten Superkombinators auf die freien Variablen ersetzt. Dadurch kann sich unser Lambda-Körper neue freie Variablen einhandeln. Wir müssen daher die Menge der freien Variablen jeder Lambda-Abstraktion aus „`lambdas`“ mit den Mengen der freien Variablen all derer Lambda-Abstraktionen aus „`lambdas`“ vereinigen, deren Namen im entsprechenden Lambda-Körper vorkommen.

Dieses Verfahren der Lösung von Mengengleichungen beschreibt Johnsson in seinem „dritten Versuch des Lambda-Lifting“ [Johnsson 1987]. Er geht vom folgenden Beispiel aus.

letrec
    f = **lambda** x **.** ... a ... g ... **end**;
    g = **lambda** x **.** ... b ... f ... **end**
in ...

Sei F_f die Menge der Variablen, die aus der Definition von f abstrahiert werden müssen. Dann ist

$$\begin{aligned} F_f &= \{a\} \cup F_g \\ F_g &= \{b\} \cup F_f \end{aligned}$$

Dieses Gleichungssystem lösen wir durch wiederholte Substitution.

Jetzt können wir in allen Lambda-Abstraktionen aus „`lambdas`“ die Namen aus „`lambdas`“ gegen die Anwendungen dieser Namen auf deren freie Variablen substituieren und die Lambda-Abstraktion in derselben Art und Weise wie bei den **let**-Ausdrücken zu Superkombinatoren machen. Danach müssen wir in allen Lambda-Abstraktionen aus „`lambdas`“ die alten Namen durch die neuen Superkombinator-Namen und in den übrigen Definitionen sowie im Unterausdruck die alten Namen durch die neuen, angewendet auf die früher freien Variablen, ersetzen.

Zuletzt wenden wir auf alle neuen Superkombinator-Körper, die übrigen Definitionen des **letrec** und den Unterausdruck die „`lift`“-Funktion an.

Zur Illustration des eben Behandelten wollen wir zwei Beispiele angeben. Nach dem Liften von

```
let
     i1[a] = 1
in let
         i2[b] = 2
   in letrec
           i3[f] = lambda i5[x]. (i5[x] (i4[g] i2[b])) end;
           i4[g] = lambda i6[x]. (i3[f] i1[a]) end
      in i3[f]
      end
   end
end
```

erhalten wir

```
combinators
    i7[f] i8[--] i9[--] i5[x] =
         (i5[x] (((i10[g] i8[--]) i9[--]) i9[--]));
    i10[g] i11[--] i12[--] i6[x] =
         (((i7[f] i11[--]) i12[--]) i11[--])
end

let
     i1[a] = 1
in let
        i2[b] = 2
```

```
   in ((i7[f] i1[a]) i2[b])
   end
end
```

Unser Beispiel aus dem dritten Kapitel hat nach dem Lambda-Lifting das folgende Aussehen.

```
type list *a = NIL | CONS *a (list *a) end

combinators
    fac n    = ((prod (from 1)) n);
    from n   = construct(CONS, n, (from ((add n) 1)));
    prod x m = case x of
               CONS h t =>
                 if ((geq h) m)
                     then h
                     else ((mult h) ((prod t) m))
                 end
               end
end

(fac 10)
```

Kapitel 9

Codegeneration

Der letzte Übersetzungsschritt unseres Compilers erzeugt aus dem in Superkombinator-Form vorliegenden Programm eine Datei mit dem entsprechenden G-Code. Dieser Schritt benimmt sich wie folgt.

Das entstehende G-Code-Programm setzt sich aus den folgenden Teilen zusammen.

- Einem Segment mit Code, der die Initialisierung des Systems vornimmt.
- Einem Segment mit Code, der den Ausdruck E_{Prog} auswertet und dessen Wert druckt.
- Für jede Superkombinator-Definition einem Segment mit dem entsprechenden G-Code. Den Ausdruck E_{Prog} betrachten wir als Körper des Superkombinators „Main“, der keine Argumente erwartet. Jedes dieser Segmente beginnt mit einer Marke (dem Namen des Superkombinators).
- Markierten Segmenten von G-Code für jede eingebaute Funktion (wie z. B. **add**). Diese Segmente bilden die Laufzeitbibliothek, da sie für alle Programme gleich sind.

All diese Segmente werden nacheinander von der Funktion „`codegen`“ aus der Datei “`el_code.c`” ausgegeben. Die Codefolge für die ersten beiden Segmente ist ziemlich einfach und besteht aus den folgenden Befehlen.

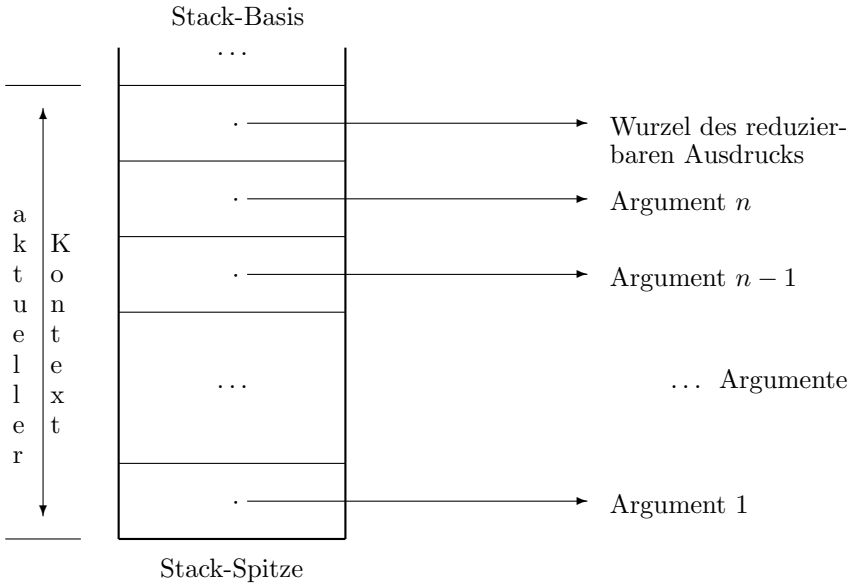
```
BEGIN     Main   % Markiert den Beginn des Programms
EVAL             % Wertet "Main" aus
PRINT            % Druckt das Ergebnis
END              % Programm-Ende
```

Die verbleibenden Segmente behandeln wir in den nächsten beiden Abschnitten.

9.1 Übersetzung einer Superkombinator-Definition

Für die Übersetzung der Definition eines Superkombinators bedient sich „`codegen`“ der nachstehenden Schemata, die als gleichnamige Funktionen in der Datei “`el_code.c`” zu finden sind.

- Das **R**-Schema generiert den Code für den Körper E des Superkombinators f. Wenn f eine Funktion von n Argumenten ist, erwartet der von **R** erzeugte Code, daß der Stack der G-Maschine folgendermaßen aussieht.



- Das **RS**-Schema vervollständigt **R**, indem es den Code für Ausdrücke der Art $(E_1\ E_2)$ generiert.

- Das **C**-Schema generiert den Code zur Erzeugung einer Instanz eines Ausdrucks E und hinterläßt einen Zeiger auf den erzeugten Graphen an der Stack-Spitze.

- **C** wird vom **CS**-Schema in analoger Weise vervollständigt wie **R** von **RS**.
- Das **E**-Schema generiert den Code zur Auswertung eines Ausdrucks E und hinterläßt einen Zeiger auf den Wert an der Spitze des Stack.
- Das **ES**-Schema vervollständigt **E**.
- Das **B**-Schema erzeugt den Code zur Berechnung des Wertes des Ausdrucks E, wenn bekannt ist, daß dieser Wert ein einfaches Datenobjekt sein muß, und hinterläßt den Wert auf dem Datenstack V.

Während der Übersetzung eines Superkombinators muß der Compiler ein Modell des Stacks halten, weil er wissen muß, an welcher Stack-Position (von der Spitze aus) der Zeiger auf den Wert einer Variablen zu finden ist. Daher erhalten die Übersetzungsschemata zwei zusätzliche Argumente:

- p, eine Liste aus Variablenname-Stackposition-Paaren, wobei die Stackposition von der Basis des aktuellen Kontextes aus gezählt wird (das Baiselement hat die Position 0); und
- d, die Tiefe des aktuellen Kontextes minus 1.

Die Funktion „`codegen`“ ruft nun für jede Superkombinator-Definition

$$f\ x_1\ x_2\ \ldots\ x_n\ =\ E$$

die Funktion **R** wie folgt auf.

$$\mathbf{R}\ (E,\ [(x_1 = n),\ (x_2 = n-1),\ \ldots\ ,\ (x_n = 1)],\ n)$$

9.1.1 Das R-Schema

Das Schema **R** erzeugt den Code für die Anwendung eines Superkombinators auf seine Argumente. Dieser Code hat vier Dinge zu tun:

1. Konstruktion einer Instanz des Superkombinator-Körpers, wobei die Parameter auf dem Stack verwendet werden.
2. Aktualisieren der Wurzel des zu reduzierenden Ausdrucks mit einer Kopie der Wurel des Ergebnisses. Besteht das Ergebnis nur aus einer Variablen, wird sie vorher ausgewertet (ansonsten würden wir zur Laufzeit unnötige Reduktionen riskieren).
3. Entfernen der Parameter vom Stack.
4. Veranlassen der nächsten Reduktion.

Das Aussehen des erzeugten Codes hängt von der Art des Ausdrucks ab; wir betrachten daher alle auftretenden Fälle.

R (i, p, d) i ist eine Konstante
= **B** (i, p, d); UPDBASIC d; POP d; RETURN

R ($E = x$/**construct**(...)/**select**(...), p, d) x ist eine Variable
= **E** (E, p, d); UPDATE ($d+1$); POP d; UNWIND

R (**fail**, p, d) = PUSHFAIL; UPDATE ($d+1$); POP d; RETURN

R ($E =$ (**not/neg/ord/chr** E_1), p, d)
= **B** (E, p, d); UPDBASIC d; POP d; RETURN

R ($E =$ ((**add/sub/.../or** E_1) E_2), p, d)
= **B** (E, p, d); UPDBASIC d; POP d; RETURN

R ($E =$ ((...(f E_1)... E_{n-1}) E_n), p, d)
f ist ein Superkombinator mit n Argumenten
= **C** (E_n, p, d);
   C (E_{n-1}, p, $d+1$);
   ...
   C (E_1, p, $d+n-1$);
   SQUEEZE n, d; JUMP f

R ($E = (E_1\ E_2)$, p, d) = **RS** (E, p, d, 0)

R (**if** E_1 **then** E_2 **else** E_3 **end**, p, d)
= **B** (E_1, p, d); JFALSE L; (L ist eine neue Marke.)
      R (E_2, p, d);
   L:
      R (E_3, p, d)

R (**case** E_c **of**
      $\text{Cons}_{i_1}\ x_{1,1}\ \ldots\ x_{1,r_1} \Longrightarrow E_1$;
      ...
      $\text{Cons}_{i_n}\ x_{n,1}\ \ldots\ x_{n,r_n} \Longrightarrow E_n$
   end, p, d)
die i_j sind die Reihenfolge-Nummern der Konstruktoren in der entsprechenden Typvereinbarung
= **E** (E_c, p, d);
   CASEJUMP (i_1, L_1), ... , (i_n, L_n), L
   L_1:
      R (E_1, **Xr** ($[x_{1,1}\ \ldots\ x_{1,r_1}]$, p, d), $d+r_1$);
   ...
   L_n:
      R (E_n, **Xr** ($[x_{n,1}\ \ldots\ x_{n,r_n}]$, p, d), $d+r_n$);
   L:
      PUSHFAIL; UPDATE ($d+1$); POP d; RETURN

$\mathbf{R}$ (**fatbar**(E_1 E_2), p, d)
= $\mathbf{E}$ (E_1, p, d); JFAIL L;
UPDATE ($d+1$); POP d; UNWIND
L: ; $\mathbf{R}$ (E_2, p, d)

$\mathbf{R}$ (**let** $x = E_x$ **in** E_b **end**, p, d)
und der Wert von x wird in E_b immer benötigt
= $\mathbf{E}$ (E_x, p, d); $\mathbf{R}$ (E_b, **append**(p, [($x = d+1$)]), $d+1$)

$\mathbf{R}$ (**let** $x = E_x$ **in** E_b **end**, p, d) sonst
= $\mathbf{C}$ (E_x, p, d); $\mathbf{R}$ (E_b, **append**(p, [($x = d+1$)]), $d+1$)

$\mathbf{R}$ (**letrec** $x_1 = E_1$; ... ; $x_n = E_n$ **in** E_b **end**, p, d)
= **CLetrec**([$x_1 = E_1$, ... , $x_n = E_n$], p', $d+n$);
$\mathbf{R}$ (E_b, p', $d+n$)
wobei $p' = \mathbf{Xr}$ ([x_1 ... x_n], p, d)

Die sechste Gleichung besagt, daß „Schwanz-Rufe“ mit der richtigen Anzahl von Argumenten in Sprünge übersetzt werden. Insbesondere überführt das **R**-Schema Schwanzrekursion in Schleifen.

Die Funktion „**CLetrec**“ nimmt eine Liste von wechselseitig rekursiven Definitionen, konstruiert Instanzen von jedem der Körper und hinterläßt die Zeiger auf die Instanzen auf dem Stack.

$$\mathbf{CLetrec}\left(\left[\begin{array}{lcl} x_1 & = & E_1 \\ \ldots & & \\ x_n & = & E_n \end{array}\right], p, d\right) \quad = \quad \begin{array}{l} \text{ALLOC } n; \\ \mathbf{C}\,(E_1, p, d);\ \text{UPDATE } n \\ \mathbf{C}\,(E_2, p, d);\ \text{UPDATE } n-1 \\ \ldots \\ \mathbf{C}\,(E_n, p, d);\ \text{UPDATE } 1 \end{array}$$

Der Wert der Funktion „**Xr**“ ist der um die Definitionen erweiterte Kontext p (**append** liefert die Verkettung zweier Listen).

$$\mathbf{Xr}\ ([x_1\ \ldots\ x_n], p, d) \quad = \quad \mathbf{append}\left(p, \left[\begin{array}{c} (x_1 = d+1) \\ \ldots \\ (x_n = d+n) \end{array}\right]\right)$$

9.1.2 Das RS-Schema

Das **RS**-Schema erwartet neben dem Ausdruck E sowie p und d noch ein viertes Argument n. **RS** vervollständigt eine Superkombinator-Reduktion, in der sich die letzten n Rippen des Körpers bereits auf dem Stack befinden.

Der von **RS** generierte Code konstruiert Instanzen der Rippen von E und bringt Zeiger darauf auf den Stack. Danach wird die Reduktion in derselben Weise wie bei **R** beendet.

RS (f, p, d, n) f hat m Argumente
$=$ PUSHFUN f, m; MKAP n; UPDATE $(d-n+1)$;
POP $(d-n)$; UNWIND

RS (x, p, d, n)
$=$ PUSH $(d-pos)$ wobei $(x, pos) \in p$;
EVAL; MKAP n; UPDATE $(d-n+1)$;
POP $(d-n)$; UNWIND

RS $((E_1\ E_2), p, d, n)$
$=$ **C** (E_2, p, d); **RS** $(E_1, p, d+1, n+1)$

Da wir im siebenten Kapitel die Transformation der Funktionsanwendungen vorgenommen haben, kann **RS** an der Spitze der Anwendungen nur Namen von Variablen bzw. Superkombinatoren finden.

9.1.3 Das C-Schema

Das **C**-Schema erhält den zu übersetzenden Ausdruck sowie p und d, die angeben, wo im Stack die Argumente des Superkombinators zu finden sind, als Argumente. Das Ergebnis ist eine G-Code-Folge, die, wenn sie ausgeführt wird, eine Instanz des Ausdrucks konstruiert.

C (i, p, d) $=$ PUSHINT/PUSHREAL/PUSHCHAR i

C (f, p, d) f hat n Argumente
$=$ PUSHFUN f, n

C (x, p, d)
$=$ PUSH $(d-pos)$ wobei $(x, pos) \in p$

C (**fail**, p, d) $=$ PUSHFAIL

C (**fatbar**$(E_1\ E_2)$, p, d)
$=$ **E** (E_1, p, d); JFAIL L_1; JUMP L_2;
L_1: ; **C** (E_2, p, d); L_2:

C $(E = (E_1\ E_2), p, d)$ $=$ **CS** $(E, p, d, 0)$

C (**if** E_1 **then** E_2 **else** E_3 **end**, p, d)
$=$ **B** (E_1, p, d); JFALSE L_1;
C (E_2, p, d); JUMP L_2
L_1:
C (E_3, p, d)
L_2:

C (**case** E_c **of**
$\quad \text{Cons}_{i_1}\ x_{1,1}\ \ldots\ x_{1,r_1} \Longrightarrow E_1\,;$
$\quad \ldots$
$\quad \text{Cons}_{i_n}\ x_{n,1}\ \ldots\ x_{n,r_n} \Longrightarrow E_n$
end, p, d)
= **E** (E_c, p, d);
CASEJUMP (i_1, L_1), ... , (i_n, L_n), L
L_1:
C (E_1, **Xr** ([$x_{1,1}$ ... x_{1,r_1}], p, d), $d + r_1$);
SLIDE r_1; JUMP L_e;
...
L_n:
C (E_n, **Xr** ([$x_{n,1}$ ... x_{n,r_n}], p, d), $d + r_n$);
SLIDE r_n; JUMP L_e;
L:
PUSHFAIL;
L_e:

C (**construct**(Cons$_i$, E_1, ... , E_{n-1}, E_n), p, d)
= **C** (E_n, p, d);
C (E_{n-1}, p, $d + 1$);
...
C (E_1, p, $d + n - 1$);
CONS i, n

C (**select**(i, E_i, p, d)
= **E** (E_i, p, d); SELECT i

C (**let** $x = E_x$ **in** E_b **end**, p, d)
und der Wert von x wird in E_b immer benötigt
= **E** (E_x, p, d); **C** (E_b, **append**(p, [($x = d + 1$)]), $d + 1$);
SLIDE 1

C (**let** $x = E_x$ **in** E_b **end**, p, d) sonst
= **C** (E_x, p, d); **C** (E_b, **append**(p, [($x = d + 1$)]), $d + 1$);
SLIDE 1

C (**letrec** $x_1 = E_1$; ... ; $x_n = E_n$ **in** E_b **end**, p, d)
= **CLetrec**([$x_1 = E_1$, ... , $x_n = E_n$], p', $d + n$);
C (E_b, p', $d + n$); SLIDE n
wobei $p' =$ **Xr** ([x_1 ... x_n], p, d)

9.1.4 Das CS-Schema

Das **CS**-Schema erwartet die gleichen Argumente wie **RS** und vervollständigt die Konstruktion des Graphen eines Ausdrucks, wobei sich die letzten n Rippen bereits auf dem Stack befinden.

Der von **CS** generierte Code konstruiert Instanzen der Rippen von E und bringt Zeiger darauf auf den Stack. Danach wird die Reduktion in derselben Weise wie bei **C** beendet.

CS (f, p, d, n) f hat m Argumente
= PUSHFUN f, m; MKAP n

CS (x, p, d, n)
= PUSH ($d - pos$) wobei (x, pos) $\in p$;
EVAL; MKAP n

CS ((E_1 E_2), p, d, n)
= **C** (E_2, p, d); **CS** (E_1, p, $d+1$, $n+1$)

9.1.5 Das E-Schema

Der beim Aufruf „**E** (E, p, d)“ erzeugte Code liefert bei seiner Abarbeitung dasselbe Ergebnis wie der von „**C** (E, p, d)“ erzeugte Code, gefolgt von der Instruktion EVAL. Bei der Verwendung von **E** wird aber die Konstruktion einiger Graph-Knoten gespart und der Code ist kürzer.

E (i, p, d) = PUSHINT/PUSHREAL/PUSHCHAR i

E (f, p, d) f hat n Argumente
= PUSHFUN f, n

E (x, p, d) = PUSH ($d - pos$) wobei (x, pos) $\in p$; EVAL

E (**fail**, p, d) = PUSHFAIL

E (**fatbar**(E_1 E_2), p, d)
= **E** (E_1, p, d); JFAIL L_1; JUMP L_2;
L_1: ; **E** (E_2, p, d); L_2:

E (E = (**not/neg/ord/chr** E_1), p, d)
= **B** (E, p, d); MKBASIC

E (E = ((**add/sub/. . . /or** E_1) E_2), p, d)
= **B** (E, p, d); MKBASIC

E (E = (E_1 E_2), p, d) = **ES** (E, p, d, 0)

E (**if** E_1 **then** E_2 **else** E_3 **end**, p, d)
= **B** (E_1, p, d); JFALSE L_1;
E (E_2, p, d); JUMP L_2
L_1:
E (E_3, p, d)
L_2:

E (**construct**(Cons$_i$, E_1, ... , E_{n-1}, E_n), p, d)
= **C** (E_n, p, d);
C (E_{n-1}, p, $d+1$);
...
C (E_1, p, $d+n-1$);
CONS i, n

E (**select**(i, E_i, p, d)
= **E** (E_i, p, d); SELECT i; EVAL

E (**case** E_c **of**
Cons$_{i_1}$ $x_{1,1}$ ... x_{1,r_1} $\Longrightarrow$ E_1 ;
...
Cons$_{i_n}$ $x_{n,1}$ ... x_{n,r_n} $\Longrightarrow$ E_n
end, p, d)
= **E** (E_c, p, d);
CASEJUMP (i_1, L_1), ... , (i_n, L_n), L
L_1:
E (E_1, **Xr** ([$x_{1,1}$ ... x_{1,r_1}], p, d), $d+r_1$);
SLIDE r_1; JUMP L_e;
...
L_n:
E (E_n, **Xr** ([$x_{n,1}$ ... x_{n,r_n}], p, d), $d+r_n$);
SLIDE r_n; JUMP L_e;
L:
PUSHFAIL;
L_e:

E (**let** $x = E_x$ **in** E_b **end**, p, d)
und der Wert von x wird in E_b immer benötigt
= **E** (E_x, p, d); **E** (E_b, **append**(p, [($x = d+1$)]), $d+1$);
SLIDE 1

E (**let** $x = E_x$ **in** E_b **end**, p, d) sonst
= **C** (E_x, p, d); **E** (E_b, **append**(p, [($x = d+1$)]), $d+1$);
SLIDE 1

E (**letrec** $x_1 = E_1$; ... ; $x_n = E_n$ **in** E_b **end**, p, d)
= **CLetrec**($[x_1 = E_1, \ldots, x_n = E_n]$, p', $d+n$);
E (E_b, p', $d+n$); SLIDE n
wobei $p' =$ **Xr** ($[x_1 \ \ldots \ x_n]$, p, d)

9.1.6 Das ES-Schema

Das **ES**-Schema erwartet die gleichen Argumente wie **RS** und vervollständigt die Auswertung eines Ausdrucks, wobei sich die letzten n Rippen bereits auf dem Stack befinden.

ES (f, p, d, n) f hat m Argumente
= PUSHFUN f, m; MKAP n; EVAL;

ES (x, p, d, n)
= PUSH ($d - pos$) wobei (x, pos) $\in p$;
EVAL; MKAP n; EVAL

ES ((E_1 E_2), p, d, n)
= **C** (E_2, p, d); **ES** (E_1, p, $d+1$, $n+1$)

9.1.7 Das B-Schema

B liefert den Code zur Auswertung eines Ausdrucks E, wobei das Ergebnis auf dem Basiswerte-Stack V abgelegt wird.

B (i, p, d) = PUSHBASIC i

B (**fatbar**(E_1 E_2), p, d)
= **E** (E_1, p, d); JFAIL L_1; GET; JUMP L_2;
L_1: ; **B** (E_2, p, d); L_2:

B ($E =$ (**not/neg/ord/chr** E_1), p, d)
= **B** (E_1, p, d); NOT/NEG/ORD/CHR

B ($E =$ ((**add/sub/.../or** E_1) E_2), p, d)
= **B** (E_2, p, d); **B** (E_1, p, d); ADD/...

B (**if** E_1 **then** E_2 **else** E_3 **end**, p, d)
= **B** (E_1, p, d); JFALSE L_1;
B (E_2, p, d); JUMP L_2
L_1:
B (E_3, p, d)
L_2:

B (**let** $x = E_x$ **in** E_b **end**, p, d)
und der Wert von x wird in E_b immer benötigt
= **E** (E_x, p, d); **B** (E_b, **append**(p, [($x = d+1$)]), $d+1$);
POP 1

B (**let** $x = E_x$ **in** E_b **end**, p, d) sonst
= **C** (E_x, p, d); **B** (E_b, **append**(p, [($x = d+1$)]), $d+1$);
POP 1

B (**letrec** $x_1 = E_1$; ... ; $x_n = E_n$ **in** E_b **end**, p, d)
= **CLetrec**([$x_1 = E_1$, ... , $x_n = E_n$], p', $d+n$);
B (E_b, p', $d+n$); POP n
wobei $p' =$ **Xr** ([x_1 ... x_n], p, d)

B (**case** E_c **of**
Cons$_{i_1}$ $x_{1,1}$ ... x_{1,r_1} $\Longrightarrow$ E_1 ;
...
Cons$_{i_n}$ $x_{n,1}$ ... x_{n,r_n} $\Longrightarrow$ E_n
end, p, d)
= **E** (E_c, p, d);
CASEJUMP (i_1, L_1), ... , (i_n, L_n), L
L_1:
B (E_1, **Xr** ([$x_{1,1}$ ... x_{1,r_1}], p, d), $d+r_1$);
POP r_1; JUMP L_e;
...
L_n:
B (E_n, **Xr** ([$x_{n,1}$ ... x_{n,r_n}], p, d), $d+r_n$);
POP r_n; JUMP L_e;
L:
Fehler
L_e:

B (E, p, d) sonst
= **E** (E, p, d); GET

9.2 Die eingebauten Funktionen

Nach der Übersetzung der Superkombinatoren schreibt ELCOM den Code derjenigen eingebauten Funktionen, deren Namen im Quellprogramm auftraten. Der Code für den **add**-Kombinator hat das folgende Aussehen.

```
add:
    PUSH      1      % 2. Argument
    EVAL
    GET
```

```
    EVAL            % 1. Argument
    GET             % Entfernt 1. Argument vom Stack
    ADD
    UPDBASIC  1     % Aktualisieren der Wurzel
    POP       1     % Entfernt 2. Argument
    RETURN          % Ergebnis ist in schwacher Kopf-Normalform
```

Die anderen arithmetischen und logischen Funktionen mit zwei Argumenten sehen genauso aus. Die Funktion „**neg**“ ist wie folgt codiert.

```
neg:
    EVAL
    GET
    NEG
    UPDBASIC  0
    RETURN
```

Analog sehen die Codes für „**not**“, „**ord**“ und „**chr**“ aus. Genauso einfach ist der **seq**-Kombinator realisiert.

```
seq:
    EVAL            % Wertet das 1. Argument aus
    POP       1     % Verwirft es
    EVAL            % Wertet das 2. Argument aus
    UPDATE    1     % Aktualisiert die Wurzel
    UNWIND
```

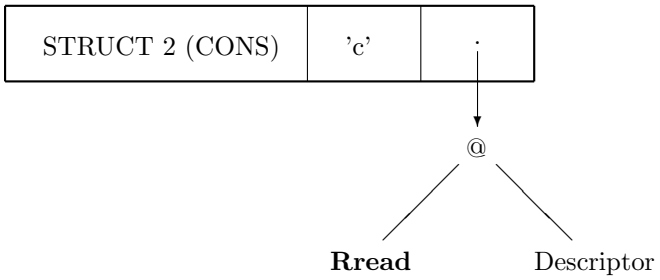
Der **read**-Kombinator erhält eine Liste aus Zeichen als Argument. Diese interpretiert er als Dateinamen und gibt den Inhalt dieser Datei als Liste von Zeichen zurück. Dazu führt „**read**“ die folgenden Schritte aus.

1. Auswertung des Arguments bis zur schwachen Kopf-Normalform.

2. Vollständige Auswertung des Arguments mit Hilfe der Funktion „**Revalarg**“. Diese Funktion wertet zuerst die Restliste und dann den Kopf aus und konstruiert daraus eine neue Liste.

3. Die Datei, deren Name die jetzt vorliegende Zeichen-Liste darstellt, wird eröffnet. Den File-Descriptor bekommt die Funktion „**Rread**“ als Argument übergeben. „**Rread**“ versucht jetzt, ein Zeichen zu lesen.

   - War das Lesen nicht erfolgreich, ist der Wert des Funktionsaufrufes die leere Liste — in der G-Maschine ein Knoten (STRUCT 1).
   - Wenn ein Zeichen gelesen wurde, ist der Wert die folgende Datenstruktur ('c' ist das gelesene Zeichen).



Der Code für „**read**“ und die beiden Hilfsfunktionen hat dann das folgende Aussehen.

```
read:
    PUSH      0      % Argument
    EVAL             % Auswerten zur schwachen Kopf-Normalform
    PUSHFUN   Revalarg, 1
    MKAP
    EVAL
% Jetzt ist das Argument vollstaendig ausgewertet.
    OPEN             % Eroeffnen der Datei
    PUSHFUN   Rread, 1
    MKAP
    EVAL
    UPDATE    2
    POP       1
    RETURN

Revalarg:
    PUSH      0
    CASEJUMP  (1,LRnil), (2,LRcons), LRnil
% Die letzte Marke ist nur Attrappe.
LRnil:
    UPDATE    1
    RETURN
LRcons:
% Stackspitze: Schwanz der Liste, darueber: Kopf
    PUSHFUN   Revalarg, 1
    MKAP
    EVAL             % Werte Schwanz aus
    PUSH      1      % Kopf
    EVAL
    CONS      2, 2
```

```
    UPDATE    3
    POP       2
    RETURN

Rread:
    PUSH      0      % Argument
    READ
    JFAIL     LRend
% Noch nicht Dateiende:
    PUSH      1      % Argument
    PUSHFUN   Rread, 1
    MKAP
    PUSH      1      % Gelesenes Zeichen
    CONS      2, 2
    UPDATE    3
    POP       2
    RETURN

LRend:                % Dateiende
    CONS      1, 0   % NIL
    UPDATE    2
    POP       1
    RETURN
```

Kapitel 10

Handhabung des Compilers ELCOM

ELCOM erwartet, daß der zu übersetzende Quelltext in einer Datei abgelegt ist, die auf "`.el`" endet. Dann können wir unseren Compiler folgendermaßen aufrufen.

elcom [–q] [–v] [–s] file[.el]

Der Suffix "`.el`" braucht nicht angegeben werden. ELCOM versteht die folgenden Optionen:

–q ELCOM gibt keinerlei Meldungen aus (außer natürlich bei Syntaxfehlern). Ist "–q" angegeben, werden die anderen beiden Optionen ignoriert.

–v ELCOM meldet jede ausgeführte Aktion.

–s ELCOM stoppt nach der Syntaxanalyse (nur bei "–vs"), nach der Variablenumbenennung, nach der Transformation der Anwendungen sowie nach dem Liften jeder Lambda-Abstraktion.

Haben wir ELCOM veranlaßt zu stoppen, begibt er sich in eine Kommandointerpreter-Schleife und gibt die folgende Mitteilung aus.

```
Stopped
Type <return> to continue, '?' for help.
> _
```

Jetzt können wir die folgenden Kommandos eingeben.

print [new] [> file]
ELCOM gibt das gesamte Programm aus. Haben wir den Parameter "new" angegeben, verwendet ELCOM nach der Variablenumbenennung

die neuen Namen und setzt die alten in eckigen Klammern dahinter. Mit "> file" lenken wir die Ausgabe in eine Datei um.

print type
ELCOM gibt den Typvereinbarungsteil aus.

print combinators [new]
ELCOM gibt nur die Superkombinator-Definitionen aus.

print expr [new]
ELCOM gibt den Ausdruck aus.

hashtable
ELCOM gibt den Inhalt der Hash-Tabelle aus.

exit
Wir brechen die Compilation ab.

Mit der ENTER-Taste verlassen wir den Kommandointerpreter, und ELCOM setzt seine Arbeit fort.

Nach Beendigung seiner Arbeit hat ELCOM die Datei "`file.g`" erzeugt, die den aus "`file.el`" hervorgegangenen G-Code enthält. Beide Dateien stehen im selben Verzeichnis.

Kapitel 11

Ausblick

Die in dieser Arbeit implementierte Sprache liegt auf dem Niveau des in [Peyton Jones 1988] beschriebenen Zwischencodes „FLIC“. Unsere Sprache unterscheidet sich von FLIC durch die für den *Anwender* leichter lesbare Syntax und dadurch, daß einige primitive Funktionen (wie z. B. **if**) durch eigene Sprachkonstrukte repräsentiert werden.

In diesem Kapitel möchten wir kurz aufzeigen, wie wir unsere Sprache einfacher handhabbar und den Compiler leistungsfähiger machen können.

11.1 Syntaktische Feinheiten

Syntaktische Feinheiten haben keine eigenständige interne Repräsentation — sie werden vom Syntaxanalysator in eine äquivalente Repräsentation umgewandelt, deren externe Darstellung für den Anwender jedoch umständlicher ist. Die syntaktischen Feinheiten geben der Sprache keine zusätzliche Mächtigkeit, sie machen nur das Programmieren leichter.

Eine Feinheit — die Notation von Zeichenketten — haben wir bereits realisiert, weil hier die Ersparnis an Schreibaufwand besonders hoch ist. Es ist schließlich `"ab"` einfacher zu schreiben als

```
construct(CONS,'a',construct(CONS,'b',construct(NIL)))
```

Weitere Feinheiten wären denkbar, wie z. B.

- die Notation von Listen und Tupeln wie in *Miranda*,
- ZF-Ausdrücke,
- die Infixnotation der binären arithmetischen und logischen sowie der Vergleichsoperatoren.

11.2 Mustererkennung

Die Aufnahme der Mustererkennung in unsere Sprache brächte sie den „höheren“ Sprachen wie *Lazy-ML* oder *Miranda* näher. Dafür müßten wir unsere Sprache und auch deren interne Repräsentation dahingehend erweitern, daß in den **case**-Ausdrücken und anstelle der zu vereinbarenden Variablen in den **let(rec)**-Ausdrücken komplexe Muster stehen dürfen.

> Ein *komplexes Muster* für Objekte des Typs T ist entweder eine Konstante, eine Variable oder ein Muster der Form $(c\ p_1\ \ldots\ p_r)$, wobei c ein Konstruktor des Typs T der Stelligkeit r ist und die p_i ihrerseits komplexe Muster sind. In einem Muster darf kein Variablenname mehrfach auftreten.

Paßt ein Datenobjekt auf ein komplexes Muster, entsteht eine Umgebung, in der alle im Muster vorkommenden Variablen an die entsprechenden Teilstrukturen gebunden sind.

Jetzt erweitern wir unsere Sprache (und entsprechend deren interne Repräsentation) so, daß sie komplexe Muster zuläßt. Dazu machen wir die folgenden Änderungen in der Grammatik.

⟨case-ausdr⟩	::=	“**case**” ⟨ausdruck⟩ “**of**” ⟨fall⟩ { “;” ⟨fall⟩ } “**end**”
⟨fall⟩	::=	⟨Muster⟩ [“&” ⟨ausdruck⟩] “=>” ⟨ausdruck⟩
⟨let-ausdr⟩	::=	“**let**” ⟨var-Vereinb⟩ “**in**” ⟨ausdruck⟩ “**end**”
⟨var-Vereinb⟩	::=	⟨Muster⟩ “=” ⟨ausdruck⟩
⟨letrec-ausdr⟩	::=	“**letrec**” ⟨var-Vereinb⟩ { “;” ⟨var-Vereinb⟩ } “**in**” ⟨ausdruck⟩ “**end**”

Mit dem Konstrukt „& ⟨ausdruck⟩“ können wir eine Bedingung angeben; und nur, wenn das Muster paßt und die Bedingung erfüllt ist, wird der Ausdruck auf der rechten Seite ausgewählt. Die Muster in einem **case**-Ausdruck dürfen sich überlappen, daher werden die Muster immer in der Reihenfolge vom ersten zum letzten abgetestet.

Zur Verdeutlichung betrachten wir als Beispiel die folgende, in *Miranda* geschriebene Funktion.

```
beispiel [1,2] = 1
beispiel [x,y] = 2, x = y
               = 3, x < y
beispiel v     = 4
```

Diese können wir jetzt so darstellen.

```
let
  beispiel = lambda n.
```

```
              case n of
                (CONS 1 (CONS 2 NIL))              => 1;
                (CONS x (CONS y NIL)) & ((eq x) y) => 2;
                (CONS x (CONS y NIL)) & ((lt x) y) => 3;
                v                                  => 4
              end
            end
in ...
```

Die erweiterten **case**- und **let(rec)**-Ausdrücke kann unser Codegenerator nicht verarbeiten, wir können sie jedoch in gewöhnliche Ausdrücke transformieren [Peyton Jones 1987, Augustsson 1987].

11.3 Typinferenz

Die Typinferenz-Komponente [Peyton Jones 1987] prüft, ob das Quellprogramm konsistent getypt werden kann. Dadurch lassen sich alle Typfehler bereits zur Compile-Zeit feststellen.

11.4 Striktheitsanalyse

Normalerweise werden die Argumente einer Funktion erst ausgewertet, wenn der Wert unbedingt gebraucht wird. Wenn aber ein Argument in jedem Falle ausgewertet wird — d. h. wenn die Funktion in diesem Argument *strikt* ist —, bringt es einen Effizienzgewinn, das Argument vor der Funktionsanwendung auszuwerten und dem Codegenerator kenntlich zu machen, daß es bereits ausgewertet ist.[1]

Die Striktheitsanalyse versucht zu bestimmen, in welchen Argumenten ein Superkombinator[2] strikt ist. Das geschieht mittels *abstrakter Interpretation*. Ausgangspunkt ist die formale Definition der Striktheit.

> Eine Funktion f mit n Argumenten ist genau dann *strikt* in ihrem i-ten Argument, wenn gilt:
>
> $$f\ x_1\ \ldots\ x_{i-1}\ \bot\ x_{i+1}\ \ldots\ x_n\ =\ \bot \qquad \forall x_j\ (1 \leq j \leq n,\ j \neq i)$$
>
> Das Symbol "$\bot$" steht für Nichtterminierung.

Wenn diese Beziehung für ein Argument gilt, können wir es vor der Funktionsanwendung auswerten, ohne das Terminierungsverhalten zu beeinflussen.

[1] Wir sparen dadurch EVAL-Befehle, wenn der Argumentwert mehrmals in der Funktion gebraucht wird.

[2] Es würde die Analyse unnötig erschweren, wenn wir in einer Lambda-Abstraktion freie Variablen zuließen.

Abstrakte Interpretation ist eine Technik zur Ableitung von Informationen über ein Programm aus seinem Text, indem eine abstrakte Version dieses Programms ausgeführt wird. Im Falle der Striktheitsanalyse benutzen wir bei der abstrakten Interpretation nur Informationen darüber, ob ein Ausdruck terminieren kann oder nicht, und lassen die wirklichen Werte außer acht.

Das Ergebnis ist der mit Anmerkungen versehene Text. In unserer Repräsentation sind im Falle strikter Argumente die Komponenten „`defstrikt`“ in den Elementen der Definitionsliste ungleich Null. Der Codegenerator kann mit diesen Anmerkungen effizienteren Code erzeugen.

Eine genaue Beschreibung der Striktheitsanalyse enthalten die Bücher [Peyton Jones 1987] und [Field 1988] sowie die dort zu diesem Thema aufgeführte Literatur.

Kapitel 12

Zusammenfassung

Diese Arbeit beschreibt den Entwurf und die Implementation eines experimentellen Compilers, der eine funktionale Programmiersprache in Instruktionen für eine abstrakte Graphenreduktionsmaschine — die G-Maschine — übersetzt. Der Compiler ist in *C* geschrieben und läuft unter dem Betriebssystem UNIX.

Die Quellsprache des Compilers ist eine Version des erweiterten Lambda-Kalküls mit verzögerter Auswertung, da dies eine Sprache mit einfacher Syntax ist, jedoch mit solcher Mächtigkeit, daß man alle funktionalen Programme in dieser Sprache ausdrücken kann. Die verzögerte Auswertung stellt einen semantischen Fortschritt gegenüber der strikten Auswertung dar, weil sie potentiell unendliche Datenstrukturen zuläßt und eine funktional saubere interaktive Ein- und Ausgabe möglich macht.

Der Compiler geht davon aus, daß das Zielprogramm durch verzögerte Graphenreduktion abgearbeitet wird. Er übersetzt jede Funktion in eine Codefolge, die bei der Anwendung auf Argumente den Graphen des Funktionskörpers erzeugt, in dem alle (freien) Vorkommen der Parameter durch Zeiger auf das entsprechende Argument ersetzt sind. Vorher wird jede Funktion in einen Superkombinator überführt, der keine freien Variablen mehr besitzt.

Um eine saubere Schnittstelle zu schaffen, sind alle G-Maschinenbefehle durch die entsprechenden Zustandsübergänge, die die G-Maschine bei der Abarbeitung ausführt, spezifiziert.

Im Anschluß daran sind die Hauptideen der Implementation der fünf Pässe des Compilers — Syntaxanalyse, Variablenumbenennung, Transformation der Funktionsanwendungen, Lambda-Lifting und Codegeneration — beschrieben.

Das Ergebnis dieser Arbeit ist der funktionsfähige Compiler „ELCOM“, dessen Anwendung kurz beschrieben ist. Dadurch, daß man die einzelnen Aktionen gut verfolgen kann, ist der Compiler zur Demonstration der verwendeten Methoden geeignet. Im Anhang ist ein Beispiel für einen Compilerlauf angegeben. Des weiteren finden wir dort den Quelltext des Compilers.

Anhang

Anhang A

Beispiel einer Compilation mit ELCOM

Wenn wir unser Beispiel aus dem dritten Kapitel als Datei "`prog.el`" eingeben, erhalten wir folgendes.

```
% Beispielprogramm: Berechnung von 10!

type list *a = NIL | CONS *a (list *a) end

letrec
  fac = lambda n .
           letrec
              from = lambda n .
                        construct(CONS,n,(from
                           (lambda n . ((add n) 1) end n)))
                     end;
              prod = lambda x m .
                        case x of  CONS h t  =>
                           if ((geq h) m)
                              then h
                              else ((mult h) ((prod t) m))
                           end
                        end
                     end
           in ((prod (from 1)) n) end
        end
in
  (fac 10)
end
```

Jetzt aktivieren wir unseren Compiler mit

elcom –v prog.el

ELCOM gibt während seiner Arbeit folgendes aus.

```
ELCOM  Version 1.0 -- 17th June 1991
Compiling "prog" to G-code.

PASS 1 -- Syntax analysis.

PASS 2 -- Scope analysis.
LETREC line 27, level 0 ...
... Renaming "fac" to "i1"
LAMBDA line 24, level 1 ...
... Renaming "n" to "i2"
LETREC line 23, level 2 ...
... Renaming "from" to "i3"
... Renaming "prod" to "i4"
LAMBDA line 11, level 3 ...
... Renaming "n" to "i5"
LAMBDA line 10, level 4 ...
... Renaming "n" to "i6"
LAMBDA line 20, level 3 ...
... Renaming "x" to "i7"
... Renaming "m" to "i8"
PATTERN IN CASE line 18, level 4 ...
... Renaming "h" to "i9"
... Renaming "t" to "i10"

PASS 3 -- Application transformation.
Application line 10: LAMBDA found. 1 argument.
Eliminating redundant definitions.
Computing binding levels.

PASS 4 -- Lambda lifting.
Lifting "fac" (LETREC line 24): 1 parameter, 0 free variables.
New name: "i11"
Lifting "from" (LETREC line 11): 1 parameter, 0 free variables.
New name: "i12"
Lifting "prod" (LETREC line 20): 2 parameters, 0 free variables.
New name: "i13"
Eliminating redundant let(rec)'s.

PASS 5 -- G-code generation.
```

```
Creating file "prog.g"

Generating code for main expression
Generating code for combinator "fac"
Generating code for combinator "from"
Generating code for combinator "prod"

Compilation terminated successfully!
```

Die erzeugte Datei “prog.g” hat den nachstehenden Inhalt.

```
% Program "prog"

    BEGIN     Main
    EVAL
    PRINT
    END

Main:
% Main expression to be evaluated.
    PUSHINT   10
    JUMP      i11           % fac (combinator)

i11:
% Combinator (1 arg). Original: "fac" (1 arg).
% Argument: n
    PUSH      0             % n
    PUSHINT   1
    PUSHFUN   i12, 1        % from
    MKAP      1
    SQUEEZE   2, 1
    JUMP      i13           % prod (combinator)

i12:
% Combinator (1 arg). Original: "from" (1 arg).
% Argument: n
    PUSHINT   1
    PUSH      1             % n
    PUSHFUN   add, 2
    MKAP      2
    PUSHFUN   i12, 1        % from
    MKAP      1
    PUSH      1             % n
    CONS      2, 2          % CONS
    UPDATE    2
```

```
    POP       1
    UNWIND

i13:
% Combinator (2 args). Original: "prod" (2 args).
% Arguments: x m
    PUSH      0             % x
    EVAL
    CASEJUMP  (2,L1), L2
L1:
    PUSH      3             % m
    EVAL
    GET
    PUSH      1             % h
    EVAL
    GET
    GEQ
    JFALSE    L3
    PUSH      1             % h
    EVAL
    UPDATE    5
    POP       4
    UNWIND
L3:
    PUSH      3             % m
    PUSH      1             % t
    PUSHFUN   i13, 2        % prod
    MKAP      2
    EVAL
    GET
    PUSH      1             % h
    EVAL
    GET
    MULT
    UPDBASIC  4
    POP       4
    RETURN
L2:
    PUSHFAIL
    UPDATE    3
    POP       2
    RETURN

% Built-in combinators
```

```
add:
    PUSH      1
    EVAL
    GET
    EVAL
    GET
    ADD
    UPDBASIC  1
    POP       1
    RETURN

mult:
    PUSH      1
    EVAL
    GET
    EVAL
    GET
    MULT
    UPDBASIC  1
    POP       1
    RETURN

geq:
    PUSH      1
    EVAL
    GET
    EVAL
    GET
    GEQ
    UPDBASIC  1
    POP       1
    RETURN
```

Anhang B

Quelltexte

Im folgenden ist der *C*-Quelltext des Compilers „ELCOM“, die **Lex**- und die **Yacc**-Eingabedatei angegeben. Um die Abhängigkeiten der einzelnen Dateien deutlich zu machen, folgt der Inhalt des **Make**-Files.

```
OBJ = y.tab.o elcom.o el_prim.o el_names.o el_print.o \
      el_scope.o el_app.o el_lift.o el_code.o

lex.yy.c:   el_lex.l
      lex el_lex.l

y.tab.c:    el_syn.y lex.yy.c elcom.h
      yacc el_syn.y

y.tab.o:    y.tab.c

elcom.o:    elcom.h elcom.c
el_prim.o:  el_prim.c
el_names.o: el_names.c elcom.h
el_print.o: el_print.c elcom.h
el_scope.o: el_scope.c elcom.h
el_app.o:   el_app.c elcom.h
el_lift.o:  el_lift.c elcom.h
el_code.o:  el_code.c elcom.h

elcom:      $(OBJ)
      cc -o elcom $(OBJ)
```

Nachstehend sind die Inhalte aller Dateien angegeben.

B.1 Globale Definitionen

```
/* elcom.h -- Globale Definitionen fuer ELCOM */

#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#include <string.h>

/* Typliste: */

struct typliste {
    struct typliste  *next;
    int              zeile;       /* Beginn im Quelltext   */
    char             *typop;      /* Typformender Operator */
    int              konstanz;    /* Anzahl Konstruktoren  */
    int              schemanz;    /* Anzahl schemat. Var.  */
    struct prodliste *vars;       /* Schematische Variable */
    struct sumliste  *typen;      /* Zugeh. Summentypen    */
};

struct sumliste {
    struct sumliste  *next;
    int              begzeile;    /* Beginn im Quelltext   */
    char             *kons;       /* Konstruktorname       */
    int              stell;       /* Stelligkeit           */
    struct prodliste *prodtypen;  /* Produkttypen          */
};

struct prodliste {
    struct prodliste *next;
    char             *name;       /* Name Typ/schem. Var.  */
    int              nr;          /* Nr. der schem. Var.   */
    struct typliste  *typ;        /* Typ                   */
    struct prodliste *belegung;   /* Belegung d.schem.Var. */
};

/* Ausdruecke: */

struct ausdruck {
    int              art;         /* Ausdrucksart          */
    int              beginn;      /* Beginn im Quelltext   */
    /* Typ des Ausdrucks: Hier einfuegen. */
    int              strikt;      /* Nur bei Anwendungen   */
```

```
    struct defliste  *definition;
    int              stelligkeit;
    int              standardtyp; /* Bei Konstanten        */
    char             *wert;       /* Wert bei Konstante    */
    struct ausdruck  *links, *rechts, *hinten;
};

struct defliste {
    struct defliste  *next;
    char             *varname;    /* Variablenname         */
    int              defstrikt;   /* Fuer Abstraktionen    */
    struct hashtab   *sym;        /* Hash-Tabelleneintrag  */
    struct ausdruck  *varwert;    /* Variablenwert         */
    struct defliste  *abst;       /* Kurz:Abstrahiert.Var. */
};

/* Standardtypen: */

#define INTEGER  1
#define REAL     2
#define CHAR     3
#define VARNAME  4

/* Ausdrucksarten: */

#define KONST    1
#define VAR      2
#define ANW      3
#define LAMBDA   4
#define IF       5
#define CASE     6
#define MUSTER   7
#define FATBAR   8
#define FAIL     9
#define CONS    10
#define SELECT  11
#define LET     12
#define LETREC  13

/* Kombinatoren: */

struct comliste {
    struct comliste  *next;
    char             *comname;    /* Kombinatorname        */
```

```
    char             *altcom;     /* Alter Name            */
    struct defliste  *args;       /* Argumente             */
    int              anzargs;     /* Anzahl der Argumente  */
    struct defliste  *altargs;    /* Argumente vor Liften  */
    int              altanz;      /* Anzahl                */
    struct ausdruck  *koerper;    /* Lambda-Koerper        */
};

/* Hash-Tabelle: */

struct hashtab {
    struct hashtab   *next;
    char             *neuname;    /* Name nach Umbenennung */
    char             *altname;    /* Alter Name            */
    struct ausdruck  *def;        /* Wert                  */
    int              niveau;      /* De Bruijn-Nummer      */
};

/* Hash-Tabellengroesse: Kann einfach geaendert werden!: */

#define HASHSIZE 50

/* Primitive Funktionen aus el_prim.c: */

extern char *new(unsigned);

#define TYPLISTE  sizeof(struct typliste)
#define SUMLISTE  sizeof(struct sumliste)
#define PRODLISTE sizeof(struct prodliste)
#define AUSDRUCK  sizeof(struct ausdruck)
#define DEFLISTE  sizeof(struct defliste)
#define COMLISTE  sizeof(struct comliste)
#define HASHTAB   sizeof(struct hashtab)
```

B.2 Das Hauptprogramm

```
/* elcom.c -- Hauptprogramm */

#include <signal.h>
#include "elcom.h"

extern int analyse(FILE *);               /* el_syn.y   */
extern int stop_print(char *);            /* el_print.c */
```

```
extern int scope(void);                    /* el_scope.c */
extern struct ausdruck *apptrans
(struct ausdruck *, int);                  /* el_app.c   */
extern struct ausdruck *elim
(struct ausdruck *);                       /* el_app.c   */
extern neuniv (struct ausdruck *, int);    /* el_app.c   */
extern lift (struct ausdruck *);           /* el_lift.c  */
extern codegen (char *);                   /* el_code.c  */

char *aus1 = "\nELCOM  Version 1.0 -- 17th June 1991";
char *aus2 = "Usage: elcom [-q] [-s] [-v] file[.el] \n";
char *aus3 = "Compiles extended lambda calculus to G-code";
char *aus4 = "Eliminating redundant %s. \n";

FILE *f;
char fname[85], fbasis[85];

int  msg  = 1;
int  stop = 0;

struct typliste *defs = NULL;      /* Die Typliste       */
struct ausdruck *prog = NULL;      /* Der Ausdruck       */
struct comliste *comb = NULL;      /* Kombinatoren-Liste */
struct comliste *last = NULL;      /* Letzter Kombinator */

struct hashtab  *hash[HASHSIZE];   /* Hash-Tabelle       */

interrupt (s)
int  s;
{
    char *c;

    switch (s) {
    case SIGINT:
        c = "Interrupt s";
        break;
    case SIGQUIT:
        c = "Quit s";
        break;
    case SIGTERM:
        c = "Termination s";
        break;
    default:
        c = "S";
```

```
    }
    fprintf(stderr,"\nelcom: %signal received \n",c);
    exit(1);
} /* interrupt */

main (argc, argv)
int  argc;
char **argv;
{
    struct comliste *s;
    int             first, i, j, q;
    char            *c, n[85];

    if (signal(SIGINT,interrupt) == SIG_IGN)
        signal(SIGINT,SIG_IGN);
    if (signal(SIGQUIT,interrupt) == SIG_IGN)
        signal(SIGQUIT,SIG_IGN);
    if (signal(SIGTERM,interrupt) == SIG_IGN)
        signal(SIGTERM,SIG_IGN);
    if (signal(SIGUSR1,interrupt) == SIG_IGN)
        signal(SIGUSR1,SIG_IGN);
    if (signal(SIGUSR2,interrupt) == SIG_IGN)
        signal(SIGUSR2,SIG_IGN);
    first = q = 0;
    while (--argc > 0 && (*(++argv))[0] == '-')
        for (c = argv[0]+1; *c != '\0'; c++)
            switch (*c) {
            case 'q':  /* Keine Meldungen ausgeben  */
                q = 1;
                break;
            case 's':  /* Nach jeder Aktion stoppen */
                stop = 1;
                break;
            case 'v':  /* Alle Meldungen ausgeben   */
                msg = 2;
                break;
            default:
                if (!first) {
                    puts(aus1);
                    first = 1;
                }
                fprintf(stderr,"Unknown option: -%c \n", *c);
            }
    if (first) {    /* Falsche Optionen */
```

```
    putchar('\n');
    puts(aus2);
    exit(1);
}
if (q)
    msg = stop = 0;
if (argc == 0) {   /* Kein Filename */
    puts(aus1);
    puts(aus3);
    puts(aus2);
    exit(1);
}
if (strlen(*argv) > 80) {
    puts(aus1);
    fputs("File name too long \n\n",stderr);
    exit(1);
}
strcpy(fname,*argv);
i = strlen(fname) - 1;
while (i >= 0 && fname[i] != '.' && fname[i] != '/')
    i--;
if (i < 0 || fname[i] == '/')
    strcat(fname,".el");
if ((f = fopen(fname,"r")) == NULL) {
    puts(aus1);
    fprintf(stderr,"Cannot open file \"%s\" \n\n",fname);
    exit(1);
}
if (msg)
    puts(aus1);
strcpy(n,fname);
i = strlen(n) - 1;
while (i > 0 && n[i] != '.')
    i--;
n[i] = '\0';
strcpy(fbasis,n);
while (i >= 0 && n[i] != '/')
    i--;
i++;
if (n[i] == '\0') {
    i = 0;
    strcpy(n,"noname");
}
if (msg) {
```

```
        printf("Compiling \"%s\" to G-code. \n\n", n+i);
        puts("PASS 1 -- Syntax analysis.");
    }
    if (analyse(f) != 0)
        exit(3);
    fclose(f);
    if (msg == 2 && stop)
        stop_print("");
    if (msg == 2 && !stop)
        putchar('\n');
    if (msg)
        puts("PASS 2 -- Scope analysis.");
    if (scope() != 0)
        exit(3);
    if (stop)
        stop_print("");
    /* Abhaengigkeitsanalyse: Hier einfuegen! */
    /* Typinferenz:           Hier einfuegen! */
    if (msg == 2 && !stop)
        putchar('\n');
    if (msg)
        puts("PASS 3 -- Application transformation.");
    prog = apptrans(prog,1);
    if (msg == 2)
        printf(aus4,"definitions");
    prog = elim(prog);
    if (msg == 2)
        puts("Computing binding levels.");
    neuniv(prog,0);
    if (stop)
        stop_print("");
    if (msg == 2 && !stop)
        putchar('\n');
    if (msg)
        puts("PASS 4 -- Lambda lifting.");
    lift(prog);
    if (msg == 2)
        printf(aus4,"let(rec)'s");
    s = comb;
    while (s!= NULL) {
        s->koerper = elim(s->koerper);
        s = s->next;
    }
    prog = elim(prog);
```

```
    if (stop)
        stop_print(msg == 2 ? "" :
            "after eliminating redundant let(rec)'s");
    /* Striktheitsanalyse:    Hier einfuegen! */
    if (msg == 2 && !stop)
        putchar('\n');
    if (msg)
        puts("PASS 5 -- G-code generation.");
    strcat(fbasis,".g");
    if (msg == 2) {
        if ((f = fopen(fbasis,"r")) != NULL) {
            fclose(f);
            c = "Rewriting";
        }
        else
            c = "Creating";
        printf("%s file \"%s\" \n\n",c,fbasis);
    }
    if ((f = fopen(fbasis,"w")) == NULL) {
        if (msg == 0)
            puts(aus1);
        fprintf(stderr,"Cannot open file \"%s\" \n\n",fbasis);
        exit(1);
    }
    codegen(n+i);
    fclose(f);
    if (msg)
        puts("\nCompilation terminated successfully! \n");
    exit(0);
} /* elcom */
```

B.3 Primitive Funktionen

```
/* el_prim.c -- Primitive Funktionen */

#include <stdio.h>
#include <malloc.h>

char *new (size)
unsigned size;
{
    char *p;
```

```
    if ((p = malloc(size)) == NULL) {
        fprintf(stderr,"\nelcom: Not enough space for ");
        fprintf(stderr,"internal data \n");
        exit(2);
    }
    return p;
} /* new */
```

B.4 Verwealtung der Namen

```
/* el_names.c -- Funktionen fuer Namensbehandlung */

#include "elcom.h"

extern struct hashtab *hash[HASHSIZE]; /* elcom.c */

long beznummer;

int sfunc[20] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

char *konstanten[] = {
    "not", "neg", "add", "sub", "mult", "div", "mod",
    "and", "or", "lt", "leq", "eq", "neq", "geq", "gt",
    "ord", "chr", "seq", "read", "##"
};

char *newname ()
{
    int  i;
    long n;
    char *c;

    i = 3;
    n = beznummer;
    while ((n /= 10) > 0)
        i++;
    c = new(i);
    sprintf(c,"i%ld",beznummer++);
    return c;
} /* newname */
```

```
int hashfun (c)
char *c;
{
    int i;

    i = 1;
    while (*c != '\0') {
        i *= *(c++);
        if (i > 10000)
            i -= 10000;
        i++;
    }
    return abs(i) % (HASHSIZE - 1);
} /* hashfun */

struct hashtab *lookup (c)
char *c;
{
    struct hashtab *h;

    h = hash[hashfun(c)];
    while (h != NULL && strcmp(c,h->neuname))
        h = h->next;
    return h;
} /* lookup */

struct hashtab *newentry (n, a, d, niv)
char            *n, *a;
struct ausdruck *d;
int             niv;
{
    struct hashtab *h, *p;
    int            i;

    p = (struct hashtab *) new(HASHTAB);
    p->neuname = n;
    p->altname = a;
    p->def = d;
    p->niveau = niv;
    i = hashfun(n);
    h = hash[i];
    hash[i] = p;
    p->next = h;
```

```
    return p;
} /* newentry */

int istkonst (c)
char *c;
{
    int  i;
    char **k;

    i = 0;
    k = konstanten;
    while (strcmp(*k,"##") && strcmp(*k,c)) {
        i++;
        k++;
    }
    if (!strcmp(*k,c)) {
        sfunc[i] = 1;
        return 1;
    }
    return 0;
} /* istkonst */

freehash (c)
char *c;
{
    struct hashtab *h, *hh;
    int            i;

    i = hashfun(c);
    h = hash[i];
    if (!strcmp((hash[i])->neuname,c))
        hash[i] = h->next;
    else {
        while (strcmp(h->next->neuname,c))
            h = h->next;
        hh = h->next;
        h->next = h->next->next;
        h = hh;
    }
    free(h);
} /* freehash */
```

B.5 Beschreibung des Lexikanalysators

```
/* el_lex.l -- Lexikanalysator-Beschreibung fuer ELCOM */

idrest                [A-Za-z0-9_]*
real                  [0-9]+"."[0-9]+

%%

[ \t\n]               |
"%".*\n               ;
"type"                return _TYPE;
"end"                 return _END;
"lambda"              return _LAMBDA;
"if"                  return _IF;
"then"                return _THEN;
"else"                return _ELSE;
"case"                return _CASE;
"of"                  return _OF;
"=>"                  return _ARROW;
"fail"                return _FAIL;
"fatbar"              return _FATBAR;
"construct"           return _CONSTRUCT;
"select"              return _SELECT;
"let"                 return _LET;
"in"                  return _IN;
"letrec"              return _LETREC;
"*"{idrest}           { yylval.val = new(strlen(yytext)+1);
                        strcpy(yylval.val,yytext);
                        return _SCHEMVAR; }
[A-Z]{idrest}         { yylval.val = new(strlen(yytext)+1);
                        strcpy(yylval.val,yytext);
                        return _CONSTRUCTOR; }
[a-z_]{idrest}        { yylval.val = new(strlen(yytext)+1);
                        strcpy(yylval.val,yytext);
                        return _IDENTIFIER; }
[0-9]+                { yylval.val = new(strlen(yytext)+1);
                        strcpy(yylval.val,yytext);
                        return _INTEGER; }
{real}                |
{real}[Ee][0-9]+      { yylval.val = new(strlen(yytext)+1);
                        strcpy(yylval.val,yytext);
                        return _REAL; }
\'.\'                 { yylval.val = new(strlen(yytext)+1);
```

```
                      strcpy(yylval.val,yytext);
                      return _CHAR; }
\"([^\"]|(\\\"))*\"  { yylval.val = new(strlen(yytext)+1);
                      strcpy(yylval.val,yytext);
                      return _STRING; }
.                    return *yytext;
```

B.6 Beschreibung des Syntaxanalysators

```
/* el_syn.y -- Syntaxanalysator-Beschreibung fuer ELCOM */

%{

#include "elcom.h"

extern struct typliste *defs;                /* elcom.c */
extern struct ausdruck *prog;                /* elcom.c */

extern char fname[85];                       /* elcom.c */

int syntaxerror = 0;

char *fehler1 = "\"%s\" line %d: ";
char *fehler2 = "Constructor \"%s\" not ";
char *fehler3 = "found in type declaration \n";

%}

%union {
    struct typliste  *typzeiger;
    struct sumliste  *sumzeiger;
    struct prodliste *prodzeiger;
    struct ausdruck  *auszeiger;
    struct defliste  *defzeiger;
    char             *val;
}

%token _TYPE _END _LAMBDA _IF _THEN _ELSE _CASE _OF
%token _ARROW _FAIL _FATBAR _CONSTRUCT _SELECT _LET
%token _IN _LETREC

%token <val> _SCHEMVAR
%token <val> _CONSTRUCTOR
```

```
%token <val> _IDENTIFIER
%token <val> _INTEGER
%token <val> _REAL
%token <val> _CHAR
%token <val> _STRING

%type <typzeiger>  typen
%type <typzeiger>  typdefs
%type <typzeiger>  def
%type <prodzeiger> schemvars
%type <prodzeiger> schemvar
%type <sumzeiger>  sumtypen
%type <sumzeiger>  sum
%type <prodzeiger> prodtypen
%type <prodzeiger> prod

%type <auszeiger>  ausdr
%type <defzeiger>  vars
%type <defzeiger>  var
%type <defzeiger>  varliste
%type <auszeiger>  faelle
%type <auszeiger>  fall
%type <auszeiger>  ausdruecke
%type <defzeiger>  vardefs
%type <defzeiger>  vardef

%%

programm:    typen ausdr
                 { defs = $1; prog = $2; }
           | ausdr
                 { defs = NULL; prog = $1; }
           ;

typen:       _TYPE typdefs _END
                 { $$ = defs = $2; ueberpruefe($2); }
           ;

typdefs:     def
                 { $$ = $1; }
           | def ';' typdefs
                 { $$ = app_typ($1,$3); }
           ;
```

```
def:          _IDENTIFIER schemvars '=' sumtypen
                  { $$ = b_typ($1,$2,$4); }
            | error
                  { yyerror("Error in typ declaration"); }
            ;

schemvars:        { $$ = NULL; }
            | schemvar schemvars
                  { $$ = app_prod($1,$2); }
            ;

schemvar:     _SCHEMVAR
                  { $$ = b_prod($1,NULL); }
            | error
                  { yyerror("Schemat. var. expected"); }
            ;

sumtypen:     sum
                  { $$ = $1; }
            | sum '|' sumtypen
                  { $$ = app_sum($1,$3); }
            ;

sum:          _CONSTRUCTOR prodtypen
                  { $$ = b_sum($1,$2); }
            | error
                  { yyerror("Constructor + args expected"); }
            ;

prodtypen:        { $$ = NULL; }
            | prod prodtypen
                  { $$ = app_prod($1,$2); }

prod:         _IDENTIFIER
                  { $$ = b_prod($1,NULL); }
            | schemvar
                  { $$ = $1; }
            | '(' _IDENTIFIER prodtypen ')'
                  { $$ = b_prod($2,$3); }
            ;


ausdr:        _INTEGER
                  { $$ = b_konst(KONST,INTEGER,$1); }
```

```
            | _REAL
                { $$ = b_konst(KONST,REAL,$1); }
            | _CHAR
                { $$ = b_konst(KONST,CHAR,$1); }
            | _STRING
                { $$ = b_string($1); }
            | _FAIL
                { $$ = b_konst(FAIL,0,NULL); }
            | _IDENTIFIER
                { $$ = b_konst(VAR,VARNAME,$1); }
            | '(' ausdr ausdr ')'
                { $$ = b_anw(ANW,$2,$3); }
            | _FATBAR '(' ausdr ausdr ')'
                { $$ = b_anw(FATBAR,$3,$4); }
            | _LAMBDA vars '.' ausdr _END
                { $$ = b_lambda($2,$4); }
            | _IF ausdr _THEN ausdr _ELSE ausdr _END
                { $$ = b_if($2,$4,$6); }
            | _CASE ausdr _OF faelle _END
                { $$ = b_case($2,$4); }
            | _CONSTRUCT '(' _CONSTRUCTOR ausdruecke ')'
                { $$ = b_cons($3,$4); }
            | _SELECT '(' _INTEGER ',' ausdr ')'
                { $$ = b_sel($3,$5); }
            | _LET vardef _IN ausdr _END
                { $$ = b_let(LET,$2,$4); }
            | _LETREC vardefs _IN ausdr _END
                { $$ = b_let(LETREC,$2,$4); }
            | error
                { yyerror("Error in expression"); }
            ;

vars:         var
                { $$ = $1; }
            | var vars
                { $$ = app_def($1,$2); }
            ;

var:          _IDENTIFIER
                { $$ = b_def($1,NULL); }
            | error
                { yyerror("Identifier expected"); }
            ;
```

```
faelle:      fall
                 { $$ = $1; }
           | fall ';' faelle
                 { $$ = app_ausl($1,$3); }
           ;

fall:        '(' _CONSTRUCTOR varliste ')' _ARROW ausdr
                 { $$ = b_fall($2,$3,$6); }
           | _CONSTRUCTOR varliste _ARROW ausdr
                 { $$ = b_fall($1,$2,$4); }
           | error
                 { yyerror("Pattern => expr. expected"); }
           ;

varliste:        { $$ = NULL; }
           | vars
                 { $$ = $1; }
           ;

ausdruecke:      { $$ = NULL; }
           | ',' ausdr ausdruecke
                 { $$ = app_ausl($2,$3); }
           ;

vardefs:     vardef
                 { $$ = $1; }
           | vardef ';' vardefs
                 { $$ = app_def($1,$3); }
           ;

vardef:      _IDENTIFIER '=' ausdr
                 { $$ = b_def($1,$3); }
           | error
                 { yyerror("Identifier = expr. expected"); }
           ;

%%

#include "lex.yy.c"

struct typliste *app_typ (p, q)
struct typliste *p, *q;
{
    struct typliste *r;
```

```
    r = p;
    if (r == NULL)
        return q;
    while (r->next != NULL)
        r = r->next;
    r->next = q;
    return p;
} /* app_typ */

int len_sum (p)
struct sumliste *p;
{
    int i;

    i = 0;
    while (p != NULL) {
        i++;
        p = p->next;
    }
    return i;
} /* len_sum */

int len_prod (p)
struct prodliste *p;
{
    int i;

    i = 0;
    while (p != NULL) {
        i++;
        p = p->next;
    }
    return i;
} /* len_prod */

struct typliste *b_typ (op, var, sum)
char             *op;
struct prodliste *var;
struct sumliste  *sum;
{
    struct typliste *r;

    r = (struct typliste *) new(TYPLISTE);
```

```
    r->zeile = yylineno;
    r->typop = op;
    r->konstanz = len_sum(sum);
    r->schemanz = len_prod(var);
    r->vars = var;
    r->typen = sum;
    r->next = NULL;
    return r;
} /* b_typ */

struct sumliste *app_sum (p, q)
struct sumliste *p, *q;
{
    struct sumliste *r;

    r = p;
    if (r == NULL)
        return q;
    while (r->next != NULL)
        r = r->next;
    r->next = q;
    return p;
} /* app_sum */

struct sumliste *b_sum (k, prod)
char             *k;
struct prodliste *prod;
{
    struct sumliste *p;

    p = (struct sumliste *) new(SUMLISTE);
    p->begzeile = yylineno;
    p->kons = k;
    p->stell = len_prod(prod);
    p->prodtypen = prod;
    p->next = NULL;
    return p;
} /* b_sum */

struct prodliste *app_prod (p, q)
struct prodliste *p, *q;
{
    struct prodliste *r;
```

```
    r = p;
    if (r == NULL)
        return q;
    while (r->next != NULL)
        r = r->next;
    r->next = q;
    return p;
} /* app_prod */

struct prodliste *b_prod (c, p)
char             *c;
struct prodliste *p;
{
    struct prodliste *r;

    r = (struct prodliste *) new(PRODLISTE);
    r->name = c;
    r->nr = 0;
    r->typ = NULL;
    r->belegung = p;
    r->next = NULL;
    return r;
} /* b_prod */


struct ausdruck *b_konst(ausart,typ,konstwert)
int  ausart,typ;
char *konstwert;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = ausart;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    p->standardtyp = typ;
    p->wert = konstwert;
    p->links = p->rechts = p->hinten = NULL;
    return p;
} /* b_konst */

struct ausdruck *b_anw (ausart, aus1, aus2)
```

```
int             ausart;
struct ausdruck *aus1, *aus2;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = ausart;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    p->standardtyp = 0;
    p->wert = NULL;
    p->links = aus1;
    p->rechts = aus2;
    p->hinten = NULL;
    return p;
} /* b_anw */

int len_def (p)
struct defliste *p;
{
    int i;

    i = 0;
    while (p != NULL) {
        i++;
        p = p->next;
    }
    return i;
} /* len_def */

struct ausdruck *b_lambda(vars, aus)
struct defliste *vars;
struct ausdruck *aus;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = LAMBDA;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = vars;
    p->stelligkeit = len_def(vars);
```

```
    p->standardtyp = 0;
    p->wert = NULL;
    p->links = aus;
    p->rechts = p->hinten = NULL;
    return p;
} /* b_lambda */

struct ausdruck *b_if(aus1, aus2, aus3)
struct ausdruck *aus1, *aus2, *aus3;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = IF;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    p->standardtyp = 0;
    p->wert = NULL;
    p->links = aus1;
    p->rechts = aus2;
    aus2->hinten = aus3;
    p->hinten = NULL;
    return p;
} /* b_if */

int findekonst (t, c)
struct typliste *t;
char            *c;
{
    struct sumliste *p;
    int             i;

    p = t->typen;
    i = 1;
    while (p != NULL & strcmp(p->kons,c)) {
        i++;
        p = p->next;
    }
    if (p != NULL)
        return i;
    return 0;
} /* findekonst */
```

```
struct typliste *findetyp (c)
char *c;
{
    struct typliste *t;

    t = defs;
    while (t != NULL) {
        if (findekonst(t,c))
            break;
        t = t->next;
    }
    return t;
} /* findetyp */

struct ausdruck *b_case (aus1, aus2)
struct ausdruck *aus1, *aus2;
{
    struct typliste *t;
    struct ausdruck *a, *p, *pp;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = CASE;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    p->standardtyp = 0;
    p->wert = NULL;
    p->links = aus1;
    p->rechts = aus2;
    p->hinten = NULL;
    a = p->rechts;
    if ((t = findetyp(a->wert)) == NULL) {
        fprintf(stderr,fehler1,fname,a->beginn);
        fprintf(stderr,fehler2,a->wert);
        fputs(fehler3,stderr);
        syntaxerror = 1;
        return p;
    }
    while (a != NULL) {
        if ((a->standardtyp = findekonst(t,a->wert))
            == 0) {
            fprintf(stderr,fehler1,fname,a->beginn);
```

```
            fprintf(stderr,fehler2,a->wert);
            fputs("defined in type \"",stderr);
            fputs(t->typop,stderr);
            fputs("\" \n",stderr);
            syntaxerror = 1;
        }
        pp = p->rechts;
        while (pp != NULL &&
               pp->standardtyp != a->standardtyp)
            pp = pp->hinten;
        if (pp != NULL && pp != a) {
            fprintf(stderr,fehler1,fname,a->beginn);
            fputs("Constructor \"",stderr);
            fputs(a->wert,stderr);
            fputs("\" occurs twice in CASE \n",stderr);
            syntaxerror = 1;
        }
        a = a->hinten;
    }
    return p;
} /* b_case */

struct ausdruck *b_cons (name, aus)
char            *name;
struct ausdruck *aus;
{
    struct typliste *t;
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = CONS;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    if ((t = findetyp(name)) == NULL) {
        fprintf(stderr,fehler1,fname,yylineno);
        fprintf(stderr,fehler2,name);
        fputs(fehler3,stderr);
        syntaxerror = 1;
    }
    else
        p->standardtyp = findekonst(t,name);
    p->wert = name;
```

```
    p->links = aus;
    p->rechts = p->hinten = NULL;
    return p;
} /* b_cons */

struct ausdruck *b_sel (index, aus)
char            *index;
struct ausdruck *aus;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = SELECT;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = NULL;
    p->stelligkeit = 0;
    p->standardtyp = INTEGER;
    p->wert = index;
    p->links = aus;
    p->rechts = p->hinten = NULL;
    return p;
} /* b_sel */

struct ausdruck *b_let (art, defs, aus)
int             art;
struct defliste *defs;
struct ausdruck *aus;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = art;
    p->beginn = yylineno;
    p->strikt = 0;
    p->definition = defs;
    p->stelligkeit = len_def(defs);
    p->standardtyp = 0;
    p->wert = NULL;
    p->links = aus;
    p->rechts = p->hinten = NULL;
    return p;
} /* b_let */
```

```
struct ausdruck *b_fall (kons, vars, aus)
char            *kons;
struct defliste *vars;
struct ausdruck *aus;
{
    struct ausdruck *p;

    p = (struct ausdruck *) new(AUSDRUCK);
    p->art = MUSTER;
    p->beginn = yylineno;
    p->definition = vars;
    p->stelligkeit = len_def(vars);
    p->standardtyp = -1;
    p->wert = kons;
    p->links = aus;
    p->rechts = p->hinten = NULL;
    return p;
} /* b_fall */

struct defliste *app_def (p, q)
struct defliste *p, *q;
{
    struct defliste *r;

    r = p;
    if (r == NULL)
        return q;
    while (r->next != NULL)
        r = r->next;
    r->next = q;
    return p;
} /* app_def */

struct defliste *b_def (var, wert)
char            *var;
struct ausdruck *wert;
{
    struct defliste *p;

    p = (struct defliste *) new(DEFLISTE);
    p->varname = var;
    p->defstrikt = 0;
    p->sym = NULL;
    p->varwert = wert;
```

```
    p->abst = NULL;
    p->next = NULL;
    return p;
} /* b_def */

struct ausdruck *app_ausl (p, q)
struct ausdruck *p, *q;
{
    struct ausdruck *r;

    r = p;
    if (r == NULL)
        return q;
    while (r->hinten != NULL)
        r = r->hinten;
    r->hinten = q;
    return p;
} /* app_ausl */

struct ausdruck *b_str (s)
char *s;
{
    struct ausdruck *p, *pp;
    char            *c;

    if (*s == '\0')
        return b_cons("NIL",NULL);
    c = new(4);
    if (*s == '\\')
        s++;
    c[0] = '\'';
    c[1] = *s;
    c[2] = '\'';
    c[3] = '\0';
    p = b_konst(KONST,CHAR,c);
    pp = b_str(s+1);
    return b_cons("CONS",app_ausl(p,pp));
} /* b_str */

struct ausdruck *b_string (s)
char *s;
{
    struct ausdruck *p;
```

```
    s[strlen(s)-1] = '\0';
    p = b_str(s+1);
    free(s);
    return p;
} /* b_string */

fehler (zeile, name)
int  zeile;
char *name;
{
    fprintf(stderr,fehler1,fname,zeile);
    fprintf(stderr,"Type/constructor \"%s\" ",name);
    fputs("declared twice \n",stderr);
    syntaxerror = 1;
} /* fehler */

struct defliste *nachsehen (zeile, name, p)
int             zeile;
char            *name;
struct defliste *p;
{
    struct defliste *r;

    r = p;
    while (r != NULL && strcmp(r->varname,name))
        r = r->next;
    if (r != NULL) {
        fehler(zeile,name);
        return p;
    }
    r = (struct defliste *) new(DEFLISTE);
    r->varname = name;
    r->next = p;
    return r;
} /* nachsehen */

int ueberpruefe (typen)
struct typliste *typen;
{
    struct defliste *p, *q;
    struct typliste *t;
    struct sumliste *s;

    p = NULL;
```

```
    t = typen;
    while (t != NULL) {
        p = nachsehen(t->zeile,t->typop,p);
        s = t->typen;
        while (s != NULL) {
            p = nachsehen(s->begzeile,s->kons,p);
            s = s->next;
        }
        t = t->next;
    }
    while (p != NULL) {
        q = p->next;
        free(p);
        p = q;
    }
} /* ueberpruefe */

int analyse (f)
FILE *f;
{
    yyin = f;
    if (yyparse() != 0)
        return 1;
    return syntaxerror;
} /* analyse */

yywrap ()
{
    return 1;
} /* yywrap */

yyerror (s)
char *s;
{
    fprintf(stderr,fehler1,fname,yylineno);
    fputs(s,stderr);
    putc('\n',stderr);
    syntaxerror = 1;
} /* yyerror */
```

B.7 Kommando-Interpreter

```
/* el_print.c -- Stoppen und Syntaxbaum ausgeben */
```

```
#include "elcom.h"

extern int msg;                             /* elcom.c */

extern struct typliste *defs;               /* elcom.c */
extern struct ausdruck *prog;               /* elcom.c */
extern struct comliste *comb;               /* elcom.c */
extern struct hashtab  *hash[HASHSIZE];     /* elcom.c */

extern struct hashtab *lookup (char *); /* el_names.c */

char *vor (zeile)
char *zeile;
{
    while (*zeile == ' ' || *zeile == '\t')
        zeile++;
    return zeile;
} /* vor */

char *lese (zeile, w)
char *zeile, *w;
{
    int i;

    zeile = vor(zeile);
    i = 1;
    while (!isspace(*zeile) && i++ < 20)
        *(w++) = *(zeile++);
    *w = '\0';
    return zeile;
} /* lese */

stop_print (m)
char *m;
{
    struct hashtab *h;
    FILE           *dest;
    int            i, neu, zaehler;
    char           zeile[150], w[25], *c, *cc;

    printf("\nStopped %s \n",m);
    puts("Type <return> to continue, '?' for help.");
    for (;;) {
```

```
printf("> ");
zaehler = 0;
while (fgets(zeile,150,stdin) == NULL) {
    fputs("Use \"exit\" to leave ELCOM!",stderr);
    printf("\n> ");
    zaehler++;
    if (zaehler >= 6) {
        puts("\n");
        exit(3);
    }
}
c = lese(zeile,w);
if (*w == '\0') {
    putchar('\n');
    return;
}
else if (!strcmp(w,"exit")) {
    puts("You have buried ELCOM! \n");
    exit(3);
}
else if (!strcmp(w,"?")) {
    puts("     print      [ new ] [ > file ]");
    puts("     print      type");
    puts("     print      combinators [ new ]");
    puts("     print      expr [ new ]");
    puts("     hashtable");
}
else if (!strcmp(w,"print") || !strcmp(w,"p")) {
    dest = stdout;
    c = lese(c,w);
    if (*w == '>') {
        c -= strlen(w);
        *w = '\0';
    }
    if (*w == '\0' || !strcmp(w,"new")) {
        neu = !strcmp(w,"new");
        c = lese(c,w);
        if (*w == '>') {
            cc = w;
            do {
                *cc = *(cc+1);
                cc++;
            } while (*(cc-1) != '\0');
            if (!isalnum(*w))
```

```
                    c = lese(c,w);
                if ((dest = fopen(w,"r")) != NULL) {
                    /* Vorsicht ist die Mutter... */
                    fclose(dest);
                    fputs("File \"",stderr);
                    fputs(w,stderr);
                    fputs("\" already exists \n",stderr);
                    continue;
                }
                if ((dest = fopen(w,"w")) == NULL) {
                    fputs("Cannot open file \"",stderr);
                    fputs(w,stderr);
                    fputs("\" \n",stderr);
                    continue;
                }
            }
            aus_typ(defs,dest);
            aus_com(comb,dest,neu);
            aus_aus(prog,dest,1,neu);
        }
        else if (!strcmp(w,"type"))
            aus_typ(defs,stdout);
        else if (!strcmp(w,"combinators") ||
            !strcmp(w,"comb")) {
            c = lese(c,w);
            neu = !strcmp(w,"new");
            aus_com(comb,stdout,neu);
        }
        else if (!strcmp(w,"expr")) {
            c = lese(c,w);
            neu = !strcmp(w,"new");
            aus_aus(prog,stdout,1,neu);
        }
        else
            printf("Unknown argument: %s \n",w);
        if (dest != stdout)
            fclose(dest);
    }
    else if (!strcmp(w,"hashtable") ||
        !strcmp(w,"h")) {
        i = HASHSIZE;
        zaehler = 0;
        printf("Hash table has %d indexes.",i);
        for (i = 0; i < HASHSIZE; i++)
```

```
                if (hash[i] != NULL) {
                    printf("\nIndex %2d:",i);
                    h = hash[i];
                    while (h != NULL) {
                        printf(" %s[",h->neuname);
                        if (!strcmp(h->altname,h->neuname))
                            printf("--");
                        else
                            printf(h->altname);
                        printf(",%d",h->niveau);
                        putchar(']');
                        h = h->next;
                    }
                    zaehler = 1;
                }
            if (zaehler)
                putchar('\n');
            else
                puts(" Table is empty.");
        }
        else
            printf("Unknown command: %s \n",w);
    }
} /* stop_print */

int ez;

writeln (f)
FILE *f;
{
    int i;

    putc('\n',f);
    for (i = 1; i++ <= ez; )
        putc(' ',f);
} /* writeln */

aus_typ(p, f)
struct typliste *p;
FILE            *f;
{
    if (p == NULL)
        return;
    ez = 0;
```

```
    fputs("type ",f);
    ez = 5;
    while (p != NULL) {
        fputs(p->typop,f);
        aus_schemvars(p->vars,f);
        ez += 4;
        writeln(f);
        fputs("= ",f);
        aus_summen(p->typen,f);
        ez -= 4;
        if (p->next != NULL)
            putc(';',f);
        else
            ez = 0;
        writeln(f);
        p = p->next;
    }
    fputs("end \n\n",f);
} /* aus_typ */

aus_schemvars (p, f)
struct prodliste *p;
FILE             *f;
{
    int i;

    i = 1;
    while (p != NULL) {
        if (i++ % 10 == 0)
            writeln(f);
        else
            putc(' ',f);
        fputs(p->name,f);
        p = p->next;
    }
} /* aus_schemvars */

aus_summen (p, f)
struct sumliste *p;
FILE            *f;
{
    while (p != NULL) {
        fputs(p->kons,f);
        ez += 2;
```

```
        aus_prod(p->prodtypen,f);
        ez -= 2;
        p = p->next;
        if (p != NULL) {
            writeln(f);
            fputs("| ",f);
        }
    }
} /* aus_summen */

aus_prod (p, f)
struct prodliste *p;
FILE             *f;
{
    while (p != NULL) {
        putc(' ',f);
        if (p->belegung != NULL)
            putc('(',f);
        fputs(p->name,f);
        aus_schemvars(p->belegung,f);
        if (p->belegung != NULL)
            putc(')',f);
        p = p->next;
    }
} /* aus_prod */

aus_com (p, f, neu)
struct comliste *p;
FILE            *f;
int             neu;
{
    if (p == NULL)
        return;
    ez = 0;
    fputs("combinators",f);
    ez = 4;
    while (p != NULL) {
        writeln(f);
        if (neu) {
            fprintf(f,"%s[",p->comname);
            if (!strcmp(p->comname,p->altcom))
                fputs("--",f);
            else
                fputs(p->altcom,f);
```

```
            putc(']',f);
        }
        else
            fputs(p->altcom,f);
        aus_defs(p->args,f,neu);
        fputs(" = ",f);
        ez += 4;
        aus_aus(p->koerper,f,0,neu);
        ez -= 4;
        p = p->next;
        if (p != NULL)
            putc(';',f);
    }
    ez = 0;
    writeln(f);
    fputs("end \n\n",f);
} /* aus_com */

aus_aus (p, f, links, neu)
struct ausdruck *p;
FILE            *f;
int             links, neu;
{
    struct hashtab *h;
    int            ezret;

    if (links)
        ez = 0;
    switch (p->art) {
    case KONST:
        fputs(p->wert,f);
        break;
    case VAR:
        h = lookup(p->wert);
        if (h == NULL) {
            fputs(p->wert,f);
            break;
        }
        if (neu)
            fprintf(f,"%s[",p->wert);
        if (neu && !strcmp(h->altname,h->neuname))
            fputs("--",f);
        else
            fputs(h->altname,f);
```

```
        if (neu)
            putc(']',f);
        break;
    case FAIL:
        fputs("fail",f);
        break;
    case FATBAR:
        fputs("fatbar",f);
    case ANW:
        putc('(',f);
        ez++;
        aus_aus(p->links,f,0,neu);
        putc(' ',f);
        ez += 5;
        aus_aus(p->rechts,f,0,neu);
        ez -= 6;
        putc(')',f);
        break;
    case LAMBDA:
        fputs("lambda",f);
        ez += 7;
        aus_defs(p->definition,f,neu);
        fputs(". ",f);
        ez += 5;
        aus_aus(p->links,f,0,neu);
        ez -= 12;
        fputs(" end",f);
        break;
    case IF:
        fputs("if ",f);
        aus_aus(p->links,f,0,neu);
        ez += 3;
        writeln(f);
        fputs("then ",f);
        ez += 5;
        aus_aus(p->rechts,f,0,neu);
        ez -= 5;
        writeln(f);
        fputs("else ",f);
        ez += 5;
        aus_aus(p->rechts->hinten,f,0,neu);
        ez -= 8;
        writeln(f);
        fputs("end",f);
```

```
        break;
    case CASE:
        fputs("case ",f);
        ez += 5;
        aus_aus(p->links,f,0,neu);
        ez -= 2;
        fputs(" of",f);
        writeln(f);
        aus_al(p->rechts,f,neu);
        ez -= 3;
        writeln(f);
        fputs("end",f);
        break;
    case MUSTER:
        fputs(p->wert,f);
        ez += 3;
        aus_defs(p->definition,f,neu);
        fputs(" => ",f);
        ez += 3;
        aus_aus(p->links,f,0,neu);
        ez -= 6;
        if (p->hinten != NULL) {
            putc(';',f);
            writeln(f);
        }
        break;
    case CONS:
        fputs("construct(",f);
        fputs(p->wert,f);
        if (p->links != NULL)
            fputs(", ",f);
        ezret = ez;
        ez += strlen(p->wert) + 12;
        aus_al(p->links,f,neu);
        ez = ezret;
        putc(')',f);
        break;
    case SELECT:
        fprintf(f,"select(%s, ",p->wert);
        ezret = ez;
        ez += strlen(p->wert) + 9;
        aus_aus(p->links,f,0,neu);
        ez = ezret;
        putc(')',f);
```

```
        break;
    case LET:
    case LETREC:
        fputs("let",f);
        if (p->art == LETREC)
            fputs("rec",f);
        ez += 4;
        writeln(f);
        aus_defs(p->definition,f,neu);
        ez -= 4;
        writeln(f);
        fputs("in ",f);
        ez += 4;
        aus_aus(p->links,f,0,neu);
        ez -= 4;
        writeln(f);
        fputs("end",f);
    }
    if (links)
        fputs("\n\n",f);
} /* aus_aus */

aus_defs (p, f, neu)
struct defliste *p;
FILE            *f;
int             neu;
{
    while (p != NULL) {
        putc(' ',f);
        if (p->sym == NULL)
            fputs(p->varname,f);
        else {
            if (neu)
                fprintf(f,"%s[",p->varname);
            if (neu && !strcmp(p->sym->altname,
                p->sym->neuname))
                fputs("--",f);
            else
                fputs(p->sym->altname,f);
            if (neu)
                putc(']',f);
        }
        if (p->varwert != NULL) {
            fputs(" = ",f);
```

```
            ez += 4;
            aus_aus(p->varwert,f,0,neu);
            ez -= 4;
            if (p->next != NULL) {
                putc(';',f);
                writeln(f);
            }
        }
        p = p->next;
    }
} /* aus_defs */

aus_al (p, f, neu)
struct ausdruck *p;
FILE            *f;
int             neu;
{
    while (p != NULL) {
        aus_aus(p,f,0,neu);
        if (p->art != MUSTER && p->hinten != NULL)
            fputs(", ",f);
        p = p->hinten;
    }
} /* aus_al */
```

B.8 Variablenumbenennung

```
/* el_scope.c -- Pass:  Gueltigkeitsbereich-Analyse */

#include "elcom.h"

extern int  msg;                        /* elcom.c */
extern char fname[85];                  /* elcom.c */
extern long beznummer;                /* el_names.c */

extern struct typliste *defs;           /* elcom.c */
extern struct ausdruck *prog;           /* elcom.c */
extern struct hashtab  *hash[HASHSIZE]; /* elcom.c */

extern char *newname(void);           /* el_names.c */
extern struct hashtab *lookup(char *);
extern struct hashtab *newentry(char *, char *,
struct ausdruck *, int);
```

```
extern int istkonst(char *);

struct bindliste {
    struct bindliste *next;
    struct bindliste *prev;
    struct defliste  *def;
} *erster, *letzter;

neubind (d, niv, art, zeilennr)
struct defliste *d;
int             niv, art, zeilennr;
{
    struct bindliste *b;
    char             *neu;

    /* Erzeuge neues Bindungsniveau-Listenelement. */
    b = (struct bindliste *) new(sizeof(struct bindliste));
    b->def = d;
    b->next = NULL;
    if (erster == NULL) {
        erster = letzter = b;
        b->prev = NULL;
    }
    else {
        letzter->next = b;
        b->prev = letzter;
        letzter = b;
    }
    if (msg == 2) {
        switch (art) {
        case LAMBDA:
            printf("LAMBDA");
            break;
        case MUSTER:
            printf("PATTERN IN CASE");
            break;
        case LET:
            printf("LET");
            break;
        case LETREC:
            printf("LETREC");
        }
        printf(" line %d, level %d ... \n",zeilennr,niv);
    }
```

```
    while (d != NULL) {
        neu = newname();
        if (msg == 2)
            printf("... Renaming \"%s\" to \"%s\" \n",
                d->varname,neu);
        d->sym = newentry(neu,d->varname,d->varwert,niv);
        d->varname = new(strlen(neu)+1);
        strcpy(d->varname,neu);
        d = d->next;
    }
} /* neubind */

freibind ()
{
    struct bindliste *b;

    b = letzter;
    if (erster == letzter)
        erster = letzter = NULL;
    else {
        letzter = letzter->prev;
        letzter->next = NULL;
    }
    free(b);
} /* freibind */

int umbenennen (a, niv)
struct ausdruck *a;
int             niv;
{
    struct bindliste *b;
    struct ausdruck  *p;
    struct defliste  *d;
    int              i;

    switch (a->art) {
    case KONST:
    case FAIL:
        return 0;
    case VAR:
        b = letzter;
        d = NULL;
        while (b != NULL) {
            d = b->def;
```

```
            while (d != NULL && strcmp(a->wert,d->sym->altname))
                d = d->next;
            if (d != NULL)
                break;
            b = b->prev;
        }
        if (d != NULL) {
            free(a->wert);
            a->wert = new(strlen(d->varname)+1);
            strcpy(a->wert,d->varname);
            return 0;
        }
        if (istkonst(a->wert))
            return 0;
        fprintf(stderr,"\"%s\" line %d: ",fname,a->beginn);
        fprintf(stderr,"Unbound variable: \"%s\" \n",
            a->wert);
        return 1;
    case ANW:
    case FATBAR:
        return umbenennen(a->links,niv) +
            umbenennen(a->rechts,niv);
    case IF:
    case CASE:
        i = umbenennen(a->links,niv);
        p = a->rechts;
        while (p != NULL) {
            i += umbenennen(p,niv);
            p = p->hinten;
        }
        return i ? 1 : 0;
    case CONS:
        i = 0;
        p = a->links;
        while (p != NULL) {
            i += umbenennen(p,niv);
            p = p->hinten;
        }
        return i ? 1 : 0;
    case SELECT:
        return umbenennen(a->links,niv);
    case MUSTER:
    case LAMBDA:
        neubind(a->definition,niv,a->art,a->beginn);
```

```
        i = umbenennen(a->links,niv+1);
        freibind();
        return i;
    case LET:
        i = umbenennen(a->definition->varwert,niv);
        neubind(a->definition,niv,a->art,a->beginn);
        i += umbenennen(a->links,niv+1);
        freibind();
        return i ? 1 : 0;
    case LETREC:
        neubind(a->definition,niv,a->art,a->beginn);
        i = 0;
        d = a->definition;
        while (d != NULL) {
            i += umbenennen(d->varwert,niv+1);
            if (d->varwert->art == VAR &&
                !strcmp(d->varname,d->varwert->wert)) {
                fprintf(stderr,"\"%s\" line %d: ",
                    fname,d->varwert->beginn);
                fprintf(stderr,"\"Letrec %s = %s;\" ",
                    d->sym->altname,d->sym->altname);
                fputs("not allowed (sensless)\n",stderr);
                i = 1;
            }
            d = d->next;
        }
        i += umbenennen(a->links,niv+1);
        freibind();
        return i ? 1 : 0;
    }
} /* umbenennen */

scope ()
{
    int i;

    erster = letzter = NULL;
    beznummer = 1;
    for (i = 0; i < HASHSIZE; )
        hash[i++] = NULL;
    return umbenennen(prog,0);
} /* scope */
```

B.9 Transformation der Anwendungen

```
/* el_app.c -- Pass:  Transformation der Anwendungen.  */

#include "elcom.h"

extern int  msg;                            /* elcom.c */
extern int  stop;                           /* elcom.c */
extern char fname[85];                      /* elcom.c */

extern aus_aus(struct ausdruck *, FILE *, int, int);
extern char *newname(void);              /* el_names.c */
extern struct hashtab *newentry(char *, char *,
struct ausdruck *, int);
extern freehash (char *);
/* el_syn.y: */
extern struct ausdruck *b_konst(int, int, char *);
extern struct ausdruck *b_anw(int, struct ausdruck *,
struct ausdruck *);
extern struct ausdruck *b_lambda (struct defliste *,
struct ausdruck *);
extern struct ausdruck *b_let(int, struct defliste *,
struct ausdruck *);
extern struct defliste *b_def(char *, struct ausdruck *);

struct argstack {
    struct argstack *next;
    struct ausdruck *arg;
};

translambda (a)
struct ausdruck *a;
{
    struct ausdruck *p;
    struct defliste *d;
    int             stell;

    /* lambda x.lambda y... --> lambda x y. ... */
    /* Geht, da nach Umbenennung immer: x != y. */
    while (a->links->art == LAMBDA) {
        d = a->definition;
        while (d->next != NULL)
            d = d->next;
        d->next = a->links->definition;
```

```
        p = a->links;
        a->links = a->links->links;
        free(p);
        stell = 0;
        d = a->definition;
        while (d != NULL) {
            stell++;
            d = d->next;
        }
        a->stelligkeit = stell;
    }
} /* translambda */

struct ausdruck *konstr (a, c)
struct ausdruck *a;
char            *c;
{
    char *cc;

    cc = new(strlen(c)+1);
    strcpy(cc,c);
    return b_anw(ANW,a,b_konst(VAR,VARNAME,cc));
} /* konstr */

struct ausdruck *apptrans (a, anwrechts)
struct ausdruck *a;
int             anwrechts;
{
    struct argstack *stack, *s;
    struct ausdruck *aret, *p, *pp, *pvor, *u;
    struct ausdruck *t, *temp, *templ;
    struct defliste *d;
    int             i, n, anzahl, ueberarg;
    char            *neu, *neu1, *c;

    switch (a->art) {
    case KONST:
    case VAR:
    case FAIL:
        return a;
    case FATBAR:
        a->links = apptrans(a->links,1);
        a->rechts = apptrans(a->rechts,1);
        return a;
```

```
case LAMBDA:
    translambda(a);
case MUSTER:
case SELECT:
    a->links = apptrans(a->links,1);
    return a;
case IF:
    a->links = apptrans(a->links,1);
    a->rechts = apptrans(a->rechts,1);
    a->rechts->hinten = apptrans(a->rechts->hinten,1);
    return a;
case CASE:
    a->links = apptrans(a->links,1);
    p = a->rechts;
    while (p != NULL) {
        p = apptrans(p,1);
        p = p->hinten;
    }
    return a;
case CONS:
    p = a->links;
    while (p != NULL) {
        p = apptrans(p,1);
        p = p->hinten;
    }
    return a;
case LET:
case LETREC:
    d = a->definition;
    while (d != NULL) {
        d->varwert = apptrans(d->varwert,1);
        d = d->next;
    }
    a->links = apptrans(a->links,1);
    return a;
}
/* Jetzt sind nur noch Anwendungen uebrig. */
p = pvor = a;
anzahl = 0;
while (p->art == ANW) {
    /* Transformation der Argumente */
    if (anwrechts)
        a->rechts = apptrans(a->rechts,1);
    anzahl++;
```

```
    pvor = p;
    p = p->links;
}
switch (p->art) {
case LAMBDA:
    c = "LAMBDA";
    break;
case LET:
    c = "LET";
    break;
case LETREC:
    c = "LETREC";
    break;
case CASE:
    c = "CASE";
    break;
case IF:
    c = "IF";
    break;
case FATBAR:
    c = "FATBAR";
    break;
case SELECT:
    c = "SELECT";
    break;
case VAR:
    return a;
default:  /* Typfehler */
    fprintf(stderr,"\"%s\" line %d: ",fname,p->beginn);
    fputs("Expression not allowed in application \n",
        stderr);
    if (msg)
        aus_aus(p,stderr,1,0);
    exit(3);
}
if (msg == 2) {
    printf("Application line %d: ",a->beginn);
    printf("%s found. %d argument",c,anzahl);
    if (anzahl != 1)
        putchar('s');
    puts(".");
}
switch (p->art) {  /* Transformation */
case LAMBDA:
```

```
translambda(p);
if (ueberarg = anzahl > p->stelligkeit) {
    aret = a;
    for (i = 1; i++ <= anzahl - p->stelligkeit; ) {
        u = a;
        a = a->links;
        anzahl--;
    }
}
p = a;
stack = NULL;
while (p->art == ANW) {  /* Argumentstack bauen */
    s = (struct argstack *)
        new(sizeof(struct argstack));
    s->arg = p->rechts;
    s->next = stack;
    stack = s;
    t = p;
    p = p->links;
    free(t);
}
temp = NULL;
while (stack != NULL) { /* Definitionen --> LET's */
    d = p->definition;
    p->definition = d->next;
    d->next = NULL;
    d->varwert = stack->arg;
    t = b_let(LET,d,NULL);
    t->beginn = a->beginn;
    if (temp == NULL)
        temp = templ = t;
    else {
        templ->links = t;
        templ = t;
    }
    s = stack;
    stack = stack->next;
    free(s);
}
if (p->definition == NULL) {
    /* Genuegend Argumente fuer das Lambda */
    templ->links = p->links;
    free(p);
}
```

```
        else
            templ->links = p;
        a = temp;
        if (ueberarg) {
            u->links = a;
            a = apptrans(aret,0);
        }
        else
            templ->links = apptrans(templ->links,1);
        break;
    case LET:
    case LETREC:
        pvor->links = p->links;
        p->links = a;
        a = p;
        a->links = apptrans(a->links,0);
        break;
    case IF:
    case CASE:
    case FATBAR:
    case SELECT:
        neu = newname();
        pvor->links = b_konst(VAR,VARNAME,neu);
        neu1 = new(strlen(neu)+1);
        strcpy(neu1,neu);
        p = apptrans(p);
        pp = b_lambda(NULL,p);
        pp->beginn = p->beginn;
        d = b_def(neu1,pp);
        neu1 = new(strlen(neu)+1);
        strcpy(neu1,neu);
        d->sym = newentry(neu,neu1,pp,0);
        pp = b_let(LET,d,a);
        pp->beginn = a->beginn;
        a = pp;
    }
    return a;
} /* apptrans */

subst (alt, neu, a)
char            *alt, *neu;
struct ausdruck *a;
{
    struct ausdruck *p;
```

```
struct defliste *d;

/* Ersetze im Ausdruck a alt gegen neu. */
switch (a->art) {
case KONST:
case FAIL:
    return;
case VAR:
    if (!strcmp(a->wert,alt)) {
        free(a->wert);
        a->wert = new(strlen(neu)+1);
        strcpy(a->wert,neu);
    }
    return;
case IF:
case CASE:
    subst(alt,neu,a->links);
    p = a->rechts;
    while (p != NULL) {
        subst(alt,neu,p);
        p = p->hinten;
    }
    return;
case LAMBDA:
case MUSTER:
case SELECT:
    subst(alt,neu,a->links);
    return;
case ANW:
case FATBAR:
    subst(alt,neu,a->links);
    subst(alt,neu,a->rechts);
    return;
case CONS:
    p = a->links;
    while (p != NULL) {
        subst(alt,neu,p);
        p = p->hinten;
    }
    return;
case LET:
case LETREC:
    d = a->definition;
    while (d != NULL) {
```

```
            subst(alt,neu,d->varwert);
            d = d->next;
        }
        subst(alt,neu,a->links);
    }
} /* subst */

struct ausdruck *elim (a)
struct ausdruck *a;
{
    struct ausdruck *p, *pp;
    struct defliste *d, *d1;
    int             first;
    char            *alt, *neu;

    /* Eliminieren unnuetzer Vereinbarungen */
    /* der Art let(rec) x = y ...           */
    switch (a->art) {
    case KONST:
    case VAR:
    case FAIL:
        return a;
    case ANW:
    case FATBAR:
        a->links = elim(a->links);
        a->rechts = elim(a->rechts);
        return a;
    case LAMBDA:
    case MUSTER:
    case SELECT:
        a->links = elim(a->links);
        return a;
    case IF:
    case CASE:
        a->links = elim(a->links);
        p = a->rechts;
        first = 1;
        while (p != NULL) {
            pp = p->hinten;
            p = elim(p);
            p->hinten = pp;
            if (first) {
                a->rechts = p;
                first = 0;
```

```
        }
        p = p->hinten;
    }
    return a;
case CONS:
    p = a->links;
    first = 1;
    while (p != NULL) {
        pp = p->hinten;
        p = elim(p);
        p->hinten = pp;
        if (first) {
            a->links = p;
            first = 0;
        }
        p = p->hinten;
    }
    return a;
case LET:
case LETREC:
    d = a->definition;
    while (d != NULL)
        if (d->varwert->art == VAR) {
            /* Definition der Art x = y */
            alt = d->varname;
            neu = d->varwert->wert;
            /* Ausketten: */
            if (d == a->definition)
                a->definition = d1 = d->next;
            else {
                d1 = a->definition;
                while (d1->next != d)
                    d1 = d1->next;
                d1->next = d->next;
                d1 = d1->next;
            }
            free(d);
            d = d1;
            /* Substitution: */
            if (a->art == LETREC) {
                d1 = a->definition;
                while (d1 != NULL) {
                    subst(alt,neu,d1->varwert);
                    d1 = d1->next;
```

```
                    }
                }
                subst(alt,neu,a->links);
                freehash(alt);
            }
            else
                d = d->next;
        if (a->definition == NULL) {
            p = a->links;
            free(a);
            return elim(p);
        }
        d = a->definition;
        while (d != NULL) {
            d->varwert = elim(d->varwert);
            d = d->next;
        }
        a->links = elim(a->links);
        return a;
    }
} /* elim */

neuesniv (d, niv)
struct defliste *d;
int             niv;
{
    while (d != NULL) {
        d->sym->niveau = niv;
        d = d->next;
    }
} /* neuesniv */

neuniv (a, niv)
struct ausdruck *a;
int             niv;
{
    struct ausdruck *p;
    struct defliste *d;

    /* Bindungsniveaus neu berechnen. */
    switch (a->art) {
    case KONST:
    case VAR:
    case FAIL:
```

```
    return;
case ANW:
case FATBAR:
    neuniv(a->links,niv);
    neuniv(a->rechts,niv);
    return;
case IF:
    neuniv(a->links,niv);
    neuniv(a->rechts,niv);
    neuniv(a->rechts->hinten,niv);
    return;
case CONS:
    p = a->links;
    while (p != NULL) {
        neuniv(p,niv);
        p = p->hinten;
    }
    return;
case SELECT:
    neuniv(a->links,niv);
    return;
case CASE:
    neuniv(a->links,niv);
    p = a->rechts;
    while (p != NULL) {
        neuniv(p,niv);
        p = p->hinten;
    }
    return;
case MUSTER:
case LAMBDA:
    neuesniv(a->definition,niv);
    neuniv(a->links,niv+1);
    return;
case LET:
    neuesniv(a->definition,niv);
    neuniv(a->definition->varwert,niv);
    neuniv(a->links,niv+1);
    return;
case LETREC:
    d = a->definition;
    neuesniv(d,niv);
    while (d != NULL) {
        neuniv(d->varwert,niv+1);
```

```
            d = d->next;
        }
        neuniv(a->links,niv+1);
    }
} /* neuniv */
```

B.10 Lambda-Lifting

```
/* el_lift.c -- Pass:  Lambda-Lifting. */

#include "elcom.h"

extern int msg;                         /* elcom.c    */
extern int stop;                        /* elcom.c    */

extern struct comliste *comb;           /* elcom.c    */
extern struct comliste *last;           /* elcom.c    */

extern struct ausdruck *b_konst
(int, int, char *);                     /* el_syn.y   */
extern struct ausdruck *b_anw (int,
struct ausdruck *, struct ausdruck *);
extern struct defliste *b_def (char *,
struct ausdruck *);
extern struct defliste *app_def (struct defliste *,
struct defliste *);
extern char *newname (void);            /* el_names.c */
extern struct hashtab *lookup (char *);
extern struct hashtab *newentry (char *, char *,
struct ausdruck *, int);
extern int istkonst (char *);
extern stop_print (char *);             /* el_print.c */
extern subst (char *, char *,
struct ausdruck *);                     /* el_app.c   */

struct bindliste {
    struct bindliste *next;
    struct defliste  *def;
} *erster, *letzter;

neuniveau (d)
struct defliste *d;
{
```

```
    struct bindliste *b;

    /* Erzeuge Bindungsniveau fuer 'freivar' */
    b = (struct bindliste *)
        new(sizeof (struct bindliste));
    b->def = d;
    b->next = NULL;
    if (erster == NULL)
        erster = letzter = b;
    else {
        letzter->next = b;
        letzter = b;
    }
} /* neuniveau */

freiniveau ()
{
    struct bindliste *b, *bb;

    b = letzter;
    if (erster == letzter)
        erster = letzter = NULL;
    else {
        bb = erster;
        while (bb->next != letzter)
            bb = bb->next;
        bb->next = NULL;
        letzter = bb;
    }
    free(b);
} /* freiniveau */

struct defliste *fvar (frei, a)
struct defliste *frei;  /* Schon erkannte freie Var's */
struct ausdruck *a;
{
    struct bindliste *b;
    struct comliste  *s;
    struct defliste  *d, *t, *tt;
    struct ausdruck  *p;
    int              kom;
    char             *w;

    switch (a->art) {
```

```
case KONST:
case FAIL:
    return frei;
case VAR:
    b = erster;
    d = NULL;
    while (b != NULL) {
        d = b->def;
        while (d != NULL && strcmp(d->varname,a->wert))
            d = d->next;
        if (d != NULL)
            break;
        b = b->next;
    }
    /* Kombinatoren werden nicht als freie */
    /* Variablen gemeldet.                 */
    kom = 0;
    if (d == NULL) {
        s = comb;
        while (s != NULL)
            if (!strcmp(s->comname,a->wert)) {
                kom = 1;
                break;
            }
            else
                s = s->next;
    }
    if (d == NULL && !istkonst(a->wert) && !kom) {
        /* Freie Variable */
        w = new(strlen(a->wert)+1);
        strcpy(w,a->wert);
        t = b_def(w,NULL);
        t->defstrikt = (lookup(w))->niveau
            /* defstrikt: Niveau! */;
        if (frei == NULL)
            frei = t;
        else
            /* Einsortieren: Bind.niveaus aufsteigend */
            if (t->defstrikt < frei->defstrikt) {
                t->next = frei;
                frei = t;
            }
            else {
                tt = frei;
```

```
                while (tt->next != NULL &&
                    tt->next->defstrikt > t->defstrikt)
                    tt = tt->next;
                t->next = tt->next;
                tt->next = t;
            }
        }
        return frei;
    case ANW:
    case FATBAR:
        frei = fvar(frei,a->links);
        frei = fvar(frei,a->rechts);
        return frei;
    case IF:
    case CASE:
        frei = fvar(frei,a->links);
        p = a->rechts;
        while (p != NULL) {
            frei = fvar(frei,p);
            p = p->hinten;
        }
        return frei;
    case CONS:
        p = a->links;
        while (p != NULL) {
            frei = fvar(frei,p);
            p = p->hinten;
        }
        return frei;
    case SELECT:
        return fvar(frei,a->links);
    case MUSTER:
    case LAMBDA:
        neuniveau(a->definition);
        frei = fvar(frei,a->links);
        freiniveau();
        return frei;
    case LET:
        frei = fvar(frei,a->definition->varwert);
        neuniveau(a->definition);
        frei = fvar(frei,a->links);
        freiniveau();
        return frei;
    case LETREC:
```

```
        neuniveau(a->definition);
        d = a->definition;
        while (d != NULL) {
            frei = fvar(frei,d->varwert);
            d = d->next;
        }
        frei = fvar(frei,a->links);
        freiniveau();
        return frei;
    }
} /* fvar */

struct defliste *freivar (def, letrec, a)
struct defliste *def;         /* Definitionsliste */
struct defliste *letrec;
/* Im selben LETREC gebundene LAMBDA's.           */
struct ausdruck *a;
{
    struct defliste *frei;

    /* Bestimme die freien Variablen von "a". Ist */
    /* "a"  der  Ausdruck eines  LETREC's, werden */
    /* alle die im selben LETREC gebundenen Vari- */
    /* ablen nicht  als  frei  gemeldet,  die ein */
    /* LAMBDA zum Wert haben.                     */
    erster = letzter = NULL;
    neuniveau(def);
    neuniveau(letrec);
    /* Macht die eigentliche Arbeit: */
    frei = fvar(NULL,a);
    freiniveau();
    freiniveau();
    return frei;
} /* freivar */

struct defliste *umfrei (frei, a)
struct defliste *frei;
struct ausdruck *a;
{
    struct defliste *ab, *t, *tt;
    char            *neu, *neu1;

    /* Umbenennen der freien Variablen, damit sie */
    /* abstrahiert werden koennen.                */
```

```
    ab = NULL;
    while (frei != NULL) {
        neu = newname();
        neu1 = new(strlen(neu)+1);
        strcpy(neu1,neu);
        t = b_def(neu,NULL);
        t->sym = newentry(neu1,neu1,NULL,frei->defstrikt);
        subst(frei->varname,neu,a);
        if (ab == NULL)
            ab = t;
        else {
            tt = ab;
            while (tt->next != NULL)
                tt = tt->next;
            tt->next = t;
        }
        frei = frei->next;
    }
    return ab;
} /* umfrei */

struct ausdruck *anwendung (neu, frei)
char            *neu;
struct defliste *frei;
{
    struct ausdruck *l, *p, *var;
    char            *n, *n1;

    /* Konstruktion der Anwendung von  "neu"  auf */
    /* die freien Variablen (original) "frei".    */
    n = new(strlen(neu)+1);
    strcpy(n,neu);
    l = b_konst(VAR,VARNAME,n);
    while (frei != NULL) {
        n1 = new(strlen(frei->varname)+1);
        strcpy(n1,frei->varname);
        var = b_konst(VAR,VARNAME,n1);
        p = b_anw(ANW,l,var);
        l = p;
        frei = frei->next;
    }
    return l;
} /* anwendung */
```

```
freiabst (frei)
struct defliste *frei;
{
    struct defliste *d;

    while (frei != NULL) {
        d = frei;
        frei = frei->next;
        free(d);
    }
} /* freiabst */

ueberschreibe (a, p)
struct ausdruck *a, *p;
{
    a->art = p->art;
    a->strikt = 0;
    a->definition = NULL;
    a->stelligkeit = 0;
    a->standardtyp = p->standardtyp;
    a->wert = p->wert;
    a->links = p->links;
    a->rechts = p->rechts;
} /* ueberschreibe */

liftsubst (alt, neu, frei, a)
char            *alt, *neu;
struct defliste *frei;
struct ausdruck *a;
{
    struct ausdruck *p;
    struct defliste *d;

    /* Substituiert in "a" alle Vorkommen der Var */
    /* "alt" gegen (neu ...frei...).              */
    switch (a->art) {
    case KONST:
    case FAIL:
        return;
    case VAR:
        if (!strcmp(a->wert,alt)) {
            p = anwendung(neu,frei);
            ueberschreibe(a,p);
            free(p);
```

```
        }
        return;
    case ANW:
    case FATBAR:
        liftsubst(alt,neu,frei,a->links);
        liftsubst(alt,neu,frei,a->rechts);
        return;
    case LAMBDA:
    case MUSTER:
    case SELECT:
        liftsubst(alt,neu,frei,a->links);
        return;
    case IF:
    case CASE:
        liftsubst(alt,neu,frei,a->links);
        p = a->rechts;
        while (p != NULL) {
            liftsubst(alt,neu,frei,p);
            p = p->hinten;
        }
        return;
    case CONS:
        p = a->links;
        while (p != NULL) {
            liftsubst(alt,neu,frei,p);
            p = p->hinten;
        }
        return;
    case LET:
    case LETREC:
        d = a->definition;
        while (d != NULL) {
            liftsubst(alt,neu,frei,d->varwert);
            d = d->next;
        }
        liftsubst(alt,neu,frei,a->links);
        return;
    }
} /* liftsubst */

struct defliste *vorkommen (vor, ausser, lambdas, a)
struct defliste *vor;
char            *ausser;
struct defliste *lambdas;
```

```
struct ausdruck *a;
{
    struct ausdruck *p;
    struct defliste *d, *t, *tt;
    char            *neu;

    /* Gibt die Liste aller Var. aus  "a"  ausser */
    /* "ausser" zurueck, die in "lambdas" vorkom- */
    /* men.                                       */
    switch (a->art) {
    case KONST:
    case FAIL:
        return vor;
    case VAR:
        d = lambdas;
        while (d != NULL)
            if (!strcmp(d->varname,a->wert))
                if (strcmp(a->wert,ausser))
                    break;
                else
                    d = d->next;
            else
                d = d->next;
        if (d != NULL) {
            neu = new(strlen(a->wert)+1);
            strcpy(neu,a->wert);
            tt = b_def(neu,NULL);
            tt->abst = d->abst;
            if (vor == NULL)
                vor = tt;
            else {
                t = vor;
                while (t->next != NULL)
                    t = t->next;
                t->next = tt;
            }
        }
        return vor;
    case ANW:
    case FATBAR:
        vor = vorkommen(vor,ausser,lambdas,a->links);
        vor = vorkommen(vor,ausser,lambdas,a->rechts);
        return vor;
    case LAMBDA:
```

```
    case MUSTER:
    case SELECT:
        vor = vorkommen(vor,ausser,lambdas,a->links);
        return vor;
    case IF:
    case CASE:
        vor = vorkommen(vor,ausser,lambdas,a->links);
        p = a->rechts;
        while (p != NULL) {
            vor = vorkommen(vor,ausser,lambdas,p);
            p = p->hinten;
        }
        return vor;
    case CONS:
        p = a->links;
        while (p != NULL) {
            vor = vorkommen(vor,ausser,lambdas,p);
            p = p->hinten;
        }
        return vor;
    case LET:
    case LETREC:
        d = a->definition;
        while (d != NULL) {
            vor = vorkommen(vor,ausser,lambdas,d->varwert);
            d = d->next;
        }
        vor = vorkommen(vor,ausser,lambdas,a->links);
        return vor;
    }
} /* vorkommen */

struct defliste *vereinigung (a, b)
struct defliste *a, *b;
{
    struct defliste *t, *tt;
    char            *neu;

    /* a := a U b */
    if (a == NULL) {
        if (b == NULL)
            return NULL;
        neu = new(strlen(b->varname)+1);
        strcpy(neu,b->varname);
```

```
        a = b_def(neu,NULL);
        b = b->next;
    }
    while (b != NULL) {
        t = a;
        while (t != NULL && strcmp(t->varname,b->varname))
            t = t->next;
        if (t == NULL) {
            neu = new(strlen(b->varname)+1);
            strcpy(neu,b->varname);
            t = b_def(neu,NULL);
            t->defstrikt = b->defstrikt;
            if (t->defstrikt < a->defstrikt) {
                t->next = a;
                a = t;
            }
            else {
                tt = a;
                while (tt->next != NULL &&
                    tt->next->defstrikt > t->defstrikt)
                    tt = tt->next;
                t->next = tt->next;
                tt->next = t;
            }
        }
        b = b->next;
    }
    return a;
} /* vereinigung */

lift (a)
struct ausdruck *a;
{
    struct comliste *s, *neucomb, *neulast;
    struct ausdruck *p, *pp;
    struct defliste *d, *frei, *abstrahiert;
    struct defliste *vars, *lambdas, *t;
    int             i, iteration;
    char            *neu, *neu1, *str;

    switch (a->art) {
    case KONST:
    case VAR:
    case FAIL:
```

```
        return;
    case ANW:
    case FATBAR:
        lift(a->links);
        lift(a->rechts);
        return;
    case MUSTER:
    case SELECT:
        lift(a->links);
        return;
    case IF:
    case CASE:
        lift(a->links);
        p = a->rechts;
        while (p != NULL) {
            lift(p);
            p = p->hinten;
        }
        return;
    case CONS:
        p = a->links;
        while (p != NULL) {
            lift(p);
            p = p->hinten;
        }
        return;
    case LAMBDA:
        if (msg == 2) {
            printf("Lifting LAMBDA line %d: %d parameter",
                a->beginn,a->stelligkeit);
            if (a->stelligkeit != 1)
                putchar('s');
        }
        frei = freivar(a->definition,NULL,a->links);
        d = frei;
        i = 0;
        while (d != NULL) {
            i++;
            d = d->next;
        }
        if (msg == 2) {
            printf(", %d free variable",i);
            if (i != 1)
                putchar('s');
```

```
        }
        abstrahiert = umfrei(frei,a->links);
        /* Abstrahieren: */
        neu = newname();
        neu1 = new(strlen(neu)+1);
        strcpy(neu1,neu);
        newentry(neu1,neu1,NULL,0);
        s = (struct comliste *) new(COMLISTE);
        s->comname = s->altcom = neu;
        if (msg == 2)
            printf(". New name: \"%s\" \n",neu);
        if (abstrahiert == NULL)
            s->args = a->definition;
        else {
            d = abstrahiert;
            while (d->next != NULL)
                d = d->next;
            d->next = a->definition;
            s->args = abstrahiert;
        }
        s->anzargs = a->stelligkeit + i;
        s->altargs = a->definition;
        s->altanz = a->stelligkeit;
        s->koerper = a->links;
        s->next = NULL;
        if (comb == NULL)
            comb = last = s;
        else {
            last->next = s;
            last = s;
        }
        /* LAMBDA "a" ueberschreiben: */
        pp = anwendung(neu,frei);
        freiabst(frei);
        ueberschreibe(a,pp);
        free(pp);
        if (stop)
            stop_print(msg == 2 ? "" :
                "after lifting anonymous LAMBDA");
        /* Kombinator-Koerper liften: */
        lift(s->koerper);
        return;
    case LET:
        d = a->definition;
```

```
p = d->varwert;
if (p->art != LAMBDA) {
    lift(d->varwert);
    lift(a->links);
    return;
}
if (msg == 2) {
    printf("Lifting \"%s\" (LET line %d): ",
        d->sym->altname,p->beginn);
    printf("%d parameter",p->stelligkeit);
    if (p->stelligkeit != 1)
        putchar('s');
}
frei = freivar(p->definition,NULL,p->links);
d = frei;
i = 0;
while (d != NULL) {
    i++;
    d = d->next;
}
if (msg == 2) {
    printf(", %d free variable",i);
    if (i != 1)
        putchar('s');
}
abstrahiert = umfrei(frei,p->links);
/* Abstrahieren: */
d = a->definition;
neu = newname();
neu1 = new(strlen(neu)+1);
strcpy(neu1,neu);
str = new(strlen(d->sym->altname)+1);
strcpy(str,d->sym->altname);
newentry(neu1,str,NULL,0);
neu1 = new(strlen(d->sym->altname)+1);
strcpy(neu1,d->sym->altname);
s = (struct comliste *) new(COMLISTE);
s->comname = neu;
s->altcom = neu1;
if (msg == 2)
    printf(". New name: \"%s\" \n",neu);
if (abstrahiert == NULL)
    s->args = p->definition;
else {
```

```
            d = abstrahiert;
            while (d->next != NULL)
                d = d->next;
            d->next = p->definition;
            s->args = abstrahiert;
        }
        s->anzargs = p->stelligkeit + i;
        s->altargs = p->definition;
        s->altanz = p->stelligkeit;
        s->koerper = p->links;
        s->next = NULL;
        if (comb == NULL)
            comb = last = s;
        else {
            last->next = s;
            last = s;
        }
        /* Vorkommen der gelifteten Variablen */
        /* ueberschreiben.                    */
        liftsubst(a->definition->varname,neu,frei,
            a->links);
        freiabst(frei);
        free(a->definition);
        a->definition = NULL;
        if (stop) {
            str = new(strlen(s->altcom)+25);
            strcpy(str,"after lifting \"");
            strcat(str,s->altcom);
            strcat(str,"\"");
            stop_print(msg == 2 ? "" : str);
            free(str);
        }
        /* Kombinator-Koerper liften: */
        lift(s->koerper);
        lift(a->links);
        return;
    case LETREC:
        /* Aufteilen der Definitionsliste in nor- */
        /* male  Var's und solche,  die  LAMBDA's */
        /* zum Wert haben.                        */
        vars = lambdas = NULL;
        d = a->definition;
        while (d != NULL) {
            frei = d;
```

```
        d = d->next;
        frei->next = NULL;
        if (frei->varwert->art == LAMBDA) {
            if (lambdas == NULL)
                lambdas = frei;
            else {
                t = lambdas;
                while (t->next != NULL)
                    t = t->next;
                t->next = frei;
            }
        }
        else {
            if (vars == NULL)
                vars = frei;
            else {
                t = vars;
                while (t->next != NULL)
                    t = t->next;
                t->next = frei;
            }
        }
    }
    a->definition = vars;
    if (lambdas == NULL) {    /* Nichts zum Liften! */
        d = a->definition;
        while (d != NULL) {
            lift(d->varwert);
            d = d->next;
        }
        lift(a->links);
        return;
    }
    d = lambdas;
    while (d != NULL) {    /* Freie Var's bestimmen */
        p = d->varwert;
        d->abst = freivar(p->definition,lambdas,
            p->links);
        d = d->next;
    }
    /* Kommen im Wert einer Variablen aus "lambdas" */
    /* andere Variablen  aus "lambdas" vor, muessen */
    /* deren abstrahierte Variablen zur  jeweiligen */
    /* Liste der  abstrahierten  Variablen im Sinne */
```

```
/* der Mengenvereinigung hinzugefuegt werden.   */
for (iteration = 1; iteration++ <= 2; ) {
    d = lambdas;
    while (d != NULL) {
        frei = vorkommen(NULL,d->varname,lambdas,
            d->varwert->links);
        t = frei;
        while (t != NULL) {
            d->abst = vereinigung(d->abst,t->abst);
            t = t->next;
        }
        freiabst(frei);
        d = d->next;
    }
}
/* In allen LAMBDA's:  Substitution aller Var's */
/* mit LAMBDA's zum Wert gegen (var ...abst...) */
/* ACHTUNG! var & abst noch alt!                */
d = lambdas;
while (d != NULL) {
    t = lambdas;
    while (t != NULL) {
        liftsubst(t->varname,t->varname,t->abst,
            d->varwert->links);
        t = t->next;
    }
    d = d->next;
}
/* Kombinatoren basteln: */
neucomb = neulast = NULL;
d = lambdas;
while (d != NULL) {
    neu = newname();
    neu1 = new(strlen(neu)+1);
    strcpy(neu1,neu);
    str = new(strlen(d->sym->altname)+1);
    strcpy(str,d->sym->altname);
    newentry(neu1,str,NULL,0);
    s = (struct comliste *) new(COMLISTE);
    s->comname = neu;
    neu1 = new(strlen(d->sym->altname)+1);
    strcpy(neu1,d->sym->altname);
    s->altcom = neu1;
    i = 0;
```

```
        t = d->abst;
        while (t != NULL) {
            i++;
            t = t->next;
        }
        p = d->varwert;
        if (msg == 2) {
            printf("Lifting \"%s\" ",neu1);
            printf("(LETREC line %d): ",p->beginn);
            printf("%d parameter",p->stelligkeit);
            if (p->stelligkeit != 1)
                putchar('s');
            printf(", %d free variable",i);
            if (i != 1)
                putchar('s');
            printf(". New name: \"%s\" \n",neu);
        }
        s->args = umfrei(d->abst,p->links);
        if (s->args == NULL)
            s->args = p->definition;
        else {
            t = s->args;
            while (t->next != NULL)
                t = t->next;
            t->next = p->definition;
        }
        s->anzargs = p->stelligkeit + i;
        s->altargs = p->definition;
        s->altanz = p->stelligkeit;
        s->koerper = p->links;
        s->next = NULL;
        if (neucomb == NULL)
            neucomb = neulast = s;
        else {
            neulast->next = s;
            neulast = s;
        }
        d = d->next;
    }
    /* Alle  gelifteten  Variablen  in  den */
    /* Kombinator-Koerpern substituieren:   */
    d = lambdas;
    while (d != NULL) {
        s = neucomb;
```

```
    t = lambdas;
    while (s != NULL) {
        subst(t->varname,s->comname,
            d->varwert->links);
        s = s->next;
        t = t->next;
    }
    d = d->next;
}
/* Kombinatoren  an  "comb"  anhaengen: */
if (comb == NULL)
    comb = neucomb;
else
    last->next = neucomb;
last = neulast;
/* Substituieren in der  LETREC-Defini- */
/* tionsliste und im Ausdruck:          */
d = a->definition;
while (d != NULL) {
    t = lambdas;
    s = neucomb;
    while (t != NULL) {
        liftsubst(t->varname,s->comname,
            t->abst,d->varwert);
        t = t->next;
        s = s->next;
    }
    d = d->next;
}
t = lambdas;
s = neucomb;
while (t != NULL) {
    liftsubst(t->varname,s->comname,t->abst,
        a->links);
    freiabst(t->abst);
    free(t->varwert);
    t = t->next;
    s = s->next;
}
freiabst(lambdas);
if (stop)
    stop_print(msg == 2 ? "" :
        "after lifting LETREC");
/* Liften des restlichen LETREC's: */
```

```
        s = neucomb;
        while (s != NULL) {
            lift(s->koerper);
            s = s->next;
        }
        d = a->definition;
        while (d != NULL) {
            lift(d->varwert);
            d = d->next;
        }
        lift(a->links);
        return;
    }
} /* lift */
```

B.11 Codegeneration

```
/* el_code.c -- Pass:  Codegenerator */

#include "elcom.h"

extern FILE *f;                          /* elcom.c    */
extern int  msg;                         /* elcom.c    */
extern int  sfunc[20];                   /* el_names.c */

extern struct typliste *defs;            /* elcom.c    */
extern struct ausdruck *prog;            /* elcom.c    */
extern struct comliste *comb;            /* elcom.c    */

extern struct hashtab *lookup (char *);  /* el_names.c */
extern int istkonst (char *);            /* el_names.c */
extern aus_defs (struct defliste *,
FILE *, int);                            /* el_print.c */

long labelnummer;

struct stack {
    struct stack *next;
    char         *var;
    char         *altvar;
    int          stackpos;
};
```

```
char *newlabel ()
{
    int  i;
    long l;
    char *c;

    i = 3;
    l = labelnummer;
    while ((l /= 10) > 0)
        i++;
    c = new(i);
    sprintf(c,"L%ld",labelnummer++);
    return c;
} /* newlabel */

struct stack *copystack (p)
struct stack *p;
{
    struct stack *s, *ss;

    s = NULL;
    while (p != NULL) {
        ss = (struct stack *) new(sizeof(struct stack));
        ss->var = p->var;
        ss->altvar = p->altvar;
        ss->stackpos = p->stackpos;
        ss->next = s;
        s = ss;
        p = p->next;
    }
    return s;
} /* copystack */

struct stack *append (p, name, altname, pos)
struct stack *p;
char         *name, *altname;
int          pos;
{
    struct stack *s;

    s = (struct stack *) new(sizeof(struct stack));
    s->var = name;
    s->altvar = altname;
    s->stackpos = pos;
```

```
    s->next = p;
    return s;
} /* append */

freestack (p)
struct stack *p;
{
    struct stack *s;

    while (p != NULL) {
        s = p;
        p = p->next;
        free(s);
    }
} /* freestack */

struct comliste *istkomb (c)
char *c;
{
    struct comliste *k;

    k = comb;
    while (k != NULL && strcmp(c,k->comname))
        k = k->next;
    return k;
} /* istkombinator */

int nchar (n)
int n;
{
    int i;

    i = 1;
    while ((n /= 10) > 0)
        i++;
    return i;
} /* nchar */

struct stack *pos (c, p)
char         *c;
struct stack *p;
{
    struct stack *s;
```

```
    s = p;
    while (s != NULL && strcmp(c,s->var))
        s = s->next;
    return s;
} /* pos */

int notevaled (ps, e)
int          ps;
struct stack *e;
{
    while (e != NULL && e->stackpos != ps)
        e = e->next;
    return e == NULL;
} /* notevaled */

struct stack *Xr (struct ausdruck *, struct stack *, int);
CLetrec (struct ausdruck *, struct stack *, int);
struct stack *E (struct ausdruck *, struct stack *, int,
struct stack *);
struct stack *RS (struct ausdruck *, struct stack *, int,
int, struct stack *);
struct stack *CS (struct ausdruck *, struct stack *, int,
int, struct stack *);
struct stack *ES (struct ausdruck *, struct stack *, int,
int, struct stack *);
B (struct ausdruck *, struct stack *, int, struct stack *);

struct stack *C (a, p, d, e)
struct ausdruck *a;
struct stack    *p;
int             d;
struct stack    *e;
{
    struct ausdruck *aa, *al;
    struct comliste *k;
    struct stack    *lab, *pp, *s, *ss;
    int             dd, i, j, st;
    char            *c, *cc;

    /* Generiert Code zur Konstruktion  des  Gra- */
    /* phen von "a".                              */
    switch(a->art) {
    case KONST:
        switch (a->standardtyp) {
```

```
        case INTEGER:
            c = "INT ";
            break;
        case REAL:
            c = "REAL";
            break;
        case CHAR:
            c = "CHAR";
        }
        fprintf(f,"    PUSH%s  %s \n",c,a->wert);
        return e;
    case VAR:
        if ((k = istkomb(a->wert)) != NULL) {
            i = strlen(k->comname) + nchar(k->anzargs);
            fprintf(f,"    PUSHFUN   %s, %d ",
                k->comname,k->anzargs);
            for (j = 17 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",k->altcom);
            return e;
        }
        if (!strcmp(a->wert,"neg") ||
            !strcmp(a->wert,"not") ||
            !strcmp(a->wert,"ord") ||
            !strcmp(a->wert,"chr") ||
            !strcmp(a->wert,"seq") ||
            !strcmp(a->wert,"read")) {
            fprintf(f,"    PUSHFUN   %s, 1 \n",a->wert);
            return e;
        }
        if (istkonst(a->wert)) {
            fprintf(f,"    PUSHFUN   %s, 2 \n",a->wert);
            return e;
        }
        s = pos(a->wert,p);
        if (s == NULL) {
            puts("! Internal error 1");
            exit(4);
        }
        i = nchar(d - s->stackpos);
        fprintf(f,"    PUSH      %d ",d - s->stackpos);
        for (j = 15 + i; j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s \n",s->altvar);
```

```
    return e;
case FAIL:
    fputs("    PUSHFAIL \n",f);
    return e;
case FATBAR:
    e = E(a->links,p,d,e);
    c = newlabel();
    cc = newlabel();
    fprintf(f,"    JFAIL     %s \n",c);
    fprintf(f,"    JUMP      %s \n",cc);
    fprintf(f,"%s: \n",c);
    e = C(a->rechts,p,d,e);
    fprintf(f,"%s: \n",cc);
    free(c);
    free(cc);
    return e;
case ANW:
    e = CS(a,p,d,0,e);
    return e;
case IF:
    B(a->links,p,d,e);
    c = newlabel();
    cc = newlabel();
    fprintf(f,"    JFALSE    %s \n",c);
    e = C(a->rechts,p,d,e);
    fprintf(f,"    JUMP      %s \n",cc);
    fprintf(f,"%s: \n",c);
    e = C(a->rechts->hinten,p,d,e);
    fprintf(f,"%s: \n",cc);
    free(c);
    free(cc);
    return e;
case CASE:
    e = E(a->links,p,d,e);
    fputs("    CASEJUMP  ",f);
    aa = a->rechts;
    lab = NULL;
    while (aa != NULL) {
        /* Markenliste fuer CASEJUMP erzeugen */
        s = (struct stack *) new(sizeof(struct stack));
        s->var = newlabel();
        s->next = NULL;
        fprintf(f,"(%d,%s), ",aa->standardtyp,s->var);
        if (lab == NULL)
```

```
            lab = s;
        else {
            ss = lab;
            while (ss->next != NULL)
                ss = ss->next;
            ss->next = s;
        }
        aa = aa->hinten;
    }
    c = newlabel();
    cc = newlabel();
    fprintf(f,"%s \n",c);
    aa = a->rechts;
    s = lab;
    while (aa != NULL) {
        /* Uebersetzen der CASE-Faelle */
        fprintf(f,"%s: \n",s->var);
        pp = Xr(aa,p,d);
        dd = d + aa->stelligkeit;
        e = C(aa->links,pp,dd,e);
        if (aa->stelligkeit != 0)
            fprintf(f,"    SLIDE     %d \n",
                aa->stelligkeit);
        fprintf(f,"    JUMP      %s \n",cc);
        freestack(pp);
        s = s->next;
        aa = aa->hinten;
    }
    /* Default: */
    fprintf(f,"%s: \n",c);
    fprintf(f,"    PUSHFAIL \n");
    /* Marke nach CASE: */
    fprintf(f,"%s: \n",cc);
    free(c);
    free(cc);
    freestack(lab);
    return e;
case CONS:
    if ((al = a->links) != NULL) {
        st = 1;
        while (al->hinten != NULL) {
            st++;
            al = al->hinten;
        }
```

```
            for (;;) {
                e = C(al,p,d++,e);
                if (al == a->links)
                    break;
                aa = a->links;
                while (aa->hinten != al)
                    aa = aa->hinten;
                al = aa;
            }
        }
        i = nchar(a->standardtyp) + nchar(st);
        fprintf(f,"    CONS      %d, %d ",
            a->standardtyp,st);
        for (j = 17 + i; j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s \n",a->wert);
        return e;
    case SELECT:
        e = E(a->links,p,d,e);
        fprintf(f,"    SELECT    %s \n",a->wert);
        return e;
    case LET:
        if (a->definition->defstrikt)
            e = E(a->definition->varwert,p,d,e);
        else
            e = C(a->definition->varwert,p,d,e);
        pp = copystack(p);
        pp = append(p,a->definition->varname,
            a->definition->sym->altname,++d);
        e = C(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    SLIDE     1 \n");
        return e;
    case LETREC:
        pp = Xr(a,p,d);
        dd = d + a->stelligkeit;
        CLetrec(a,pp,dd);
        e = C(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    SLIDE     %d \n",dd-d);
        return e;
    }
} /* C */
```

```
struct stack *Xr (a, p, d)
struct ausdruck *a;
struct stack    *p;
int             d;
{
    struct defliste *df;
    struct stack    *s;

    df = a->definition;
    s = NULL;
    s = copystack(p);
    while (df != NULL) {
        s = append(s,df->varname,df->sym->altname,++d);
        df = df->next;
    }
    return s;
} /* Xr */

CLetrec (a, p, d)
struct ausdruck *a;
struct stack    *p;
int             d;
{
    struct defliste *df;
    struct stack    *e;
    int             n;

    n = a->stelligkeit;
    df = a->definition;
    fprintf(f,"    ALLOC     %d \n",n);
    while (df != NULL) {
        e = C(df->varwert,p,d,NULL);
        freestack(e);
        fprintf(f,"    UPDATE    %d \n",n--);
        df = df->next;
    }
} /* CLetrec */

struct stack *E (a, p, d, e)
struct ausdruck *a;
struct stack    *p;
int             d;
struct stack    *e;
{
```

```
struct ausdruck *aa, *al;
struct comliste *k;
struct stack    *lab, *pp, *s, *ss;
int             dd, i, j, st;
char            *c, *cc;

/* Generiert Code zur Auswertung von "a". Das */
/* Ergebnis ist an der Stack-Spitze.          */
switch(a->art) {
case KONST:
    switch (a->standardtyp) {
    case INTEGER:
        c = "INT ";
        break;
    case REAL:
        c = "REAL";
        break;
    case CHAR:
        c = "CHAR";
    }
    fprintf(f,"    PUSH%s  %s \n",c,a->wert);
    return e;
case VAR:
    if ((k = istkomb(a->wert)) != NULL) {
        i = strlen(k->comname) + nchar(k->anzargs);
        fprintf(f,"    PUSHFUN   %s, %d ",
            k->comname,k->anzargs);
        for (j = 17 + i; j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s \n",k->altcom);
        return e;
    }
    if (!strcmp(a->wert,"neg") ||
        !strcmp(a->wert,"not") ||
        !strcmp(a->wert,"ord") ||
        !strcmp(a->wert,"chr") ||
        !strcmp(a->wert,"seq") ||
        !strcmp(a->wert,"read")) {
        fprintf(f,"    PUSHFUN   %s, 1 \n",a->wert);
        return e;
    }
    if (istkonst(a->wert)) {
        fprintf(f,"    PUSHFUN   %s, 2 \n",a->wert);
        return e;
```

```
        }
        s = pos(a->wert,p);
        if (s == NULL) {
            puts("! Internal error 2");
            exit(4);
        }
        i = nchar(d - s->stackpos);
        fprintf(f,"    PUSH      %d ",d - s->stackpos);
        for (j = 15 + i; j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s \n",s->altvar);
        if (notevaled(s->stackpos,e))
            fprintf(f,"    EVAL       \n");
        return e;
    case FAIL:
        fputs("    PUSHFAIL \n",f);
        return e;
    case FATBAR:
        e = E(a->links,p,d,e);
        c = newlabel();
        cc = newlabel();
        fprintf(f,"    JFAIL     %s \n",c);
        fprintf(f,"    JUMP      %s \n",cc);
        fprintf(f,"%s: \n",c);
        e = E(a->rechts,p,d,e);
        fprintf(f,"%s: \n",cc);
        free(c);
        free(cc);
        return e;
    case ANW:
        aa = a;
        i = 0;
        while (aa->art == ANW) {
            i++;
            aa = aa->links;
        }
        if (istkonst(aa->wert)) {
            if ((!strcmp(aa->wert,"neg") ||
                !strcmp(aa->wert,"not") ||
                !strcmp(aa->wert,"ord") ||
                !strcmp(aa->wert,"chr")) && i == 1)
                j = 1;
            else if (i == 2 && strcmp(aa->wert,"seq") &&
                strcmp(aa->wert,"ord") &&
```

```
            strcmp(aa->wert,"chr") &&
            strcmp(aa->wert,"neg") &&
            strcmp(aa->wert,"not") &&
            strcmp(aa->wert,"read"))
            j = 1;
        else
            j = 0;
        if (j) {
            B(a,p,d,e);
            fputs("    MKBASIC \n",f);
            return e;
        }
    }
    e = ES(a,p,d,0,e);
    return e;
case IF:
    B(a->links,p,d,e);
    c = newlabel();
    cc = newlabel();
    fprintf(f,"    JFALSE    %s \n",c);
    e = E(a->rechts,p,d,e);
    fprintf(f,"    JUMP      %s \n",cc);
    fprintf(f,"%s: \n",c);
    e = E(a->rechts->hinten,p,d,e);
    fprintf(f,"%s: \n",cc);
    free(c);
    free(cc);
    return e;
case CASE:
    e = E(a->links,p,d,e);
    fputs("    CASEJUMP  ",f);
    aa = a->rechts;
    lab = NULL;
    while (aa != NULL) {
        /* Markenliste fuer CASEJUMP erzeugen */
        s = (struct stack *) new(sizeof(struct stack));
        s->var = newlabel();
        s->next = NULL;
        fprintf(f,"(%d,%s), ",aa->standardtyp,s->var);
        if (lab == NULL)
            lab = s;
        else {
            ss = lab;
            while (ss->next != NULL)
```

```
                ss = ss->next;
            ss->next = s;
        }
        aa = aa->hinten;
    }
    c = newlabel();
    cc = newlabel();
    fprintf(f,"%s \n",c);
    aa = a->rechts;
    s = lab;
    while (aa != NULL) {
        /* Uebersetzen der CASE-Faelle */
        fprintf(f,"%s: \n",s->var);
        pp = Xr(aa,p,d);
        dd = d + aa->stelligkeit;
        e = E(aa->links,pp,dd,e);
        if (aa->stelligkeit != 0)
            fprintf(f,"    SLIDE     %d \n",
                aa->stelligkeit);
        fprintf(f,"    JUMP      %s \n",cc);
        freestack(pp);
        s = s->next;
        aa = aa->hinten;
    }
    /* Default: */
    fprintf(f,"%s: \n",c);
    fprintf(f,"    PUSHFAIL \n");
    /* Marke nach CASE: */
    fprintf(f,"%s: \n",cc);
    free(c);
    free(cc);
    freestack(lab);
    return e;
case CONS:
    if ((al = a->links) != NULL) {
        st = 1;
        while (al->hinten != NULL) {
            st++;
            al = al->hinten;
        }
        for (;;) {
            e = C(al,p,d++,e);
            if (al == a->links)
                break;
```

```
                aa = a->links;
                while (aa->hinten != al)
                    aa = aa->hinten;
                al = aa;
            }
        }
        i = nchar(a->standardtyp) + nchar(st);
        fprintf(f,"    CONS      %d, %d ",
            a->standardtyp,st);
        for (j = 17 + i; j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s \n",a->wert);
        return e;
    case SELECT:
        e = E(a->links,p,d,e);
        fprintf(f,"    SELECT    %s \n",a->wert);
        fprintf(f,"    EVAL         \n");
        return e;
    case LET:
        if (a->definition->defstrikt)
            e = E(a->definition->varwert,p,d,e);
        else
            e = C(a->definition->varwert,p,d,e);
        pp = copystack(p);
        pp = append(p,a->definition->varname,
            a->definition->sym->altname,++d);
        e = E(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    SLIDE     1 \n");
        return e;
    case LETREC:
        pp = Xr(a,p,d);
        dd = d + a->stelligkeit;
        CLetrec(a,pp,dd);
        e = E(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    SLIDE     %d \n",dd-d);
        return e;
    }
} /* E */

struct stack *RS (a, p, d, n, e)
struct ausdruck *a;
struct stack    *p;
```

```
int             d, n;
struct stack    *e;
{
    struct comliste *k;
    struct stack    *s;
    int             i, j;

    switch (a->art) {
    case VAR:
        if ((k = istkomb(a->wert)) != NULL) {
            i = strlen(k->comname) + nchar(k->anzargs);
            fprintf(f,"    PUSHFUN   %s, %d ",
                k->comname,k->anzargs);
            for (j = 17 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",k->altcom);
        }
        else if (!strcmp(a->wert,"neg") ||
            !strcmp(a->wert,"not") ||
            !strcmp(a->wert,"ord") ||
            !strcmp(a->wert,"chr") ||
            !strcmp(a->wert,"seq") ||
            !strcmp(a->wert,"read")) {
            fprintf(f,"    PUSHFUN   %s, 1 \n",a->wert);
        }
        else if (istkonst(a->wert)) {
            fprintf(f,"    PUSHFUN   %s, 2 \n",a->wert);
        }
        else {
            s = pos(a->wert,p);
            if (s == NULL) {
                puts("! Internal error 3");
                exit(4);
            }
            i = nchar(d - s->stackpos);
            fprintf(f,"    PUSH      %d ",d - s->stackpos);
            for (j = 15 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",s->altvar);
            if (notevaled(s->stackpos,e))
                fputs("    EVAL \n",f);
        }
        fprintf(f,"    MKAP      %d \n",n);
        fprintf(f,"    UPDATE    %d \n",d-n+1);
```

```
        if (d-n != 0)
            fprintf(f,"    POP       %d \n",d-n);
        fprintf(f,"    UNWIND       \n");
        return e;
    case ANW:
        if (a->strikt)
            e = E(a->rechts,p,d,e);
        else
            e = C(a->rechts,p,d,e);
        e = RS(a->links,p,d+1,n+1,e);
        return e;
    }
} /* RS */

struct stack *CS (a, p, d, n, e)
struct ausdruck *a;
struct stack    *p;
int             d, n;
struct stack    *e;
{
    struct comliste *k;
    struct stack    *s;
    int             i, j;

    switch (a->art) {
    case VAR:
        if ((k = istkomb(a->wert)) != NULL) {
            i = strlen(k->comname) + nchar(k->anzargs);
            fprintf(f,"    PUSHFUN   %s, %d ",
                k->comname,k->anzargs);
            for (j = 17 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",k->altcom);
        }
        else if (!strcmp(a->wert,"neg") ||
            !strcmp(a->wert,"not") ||
            !strcmp(a->wert,"ord") ||
            !strcmp(a->wert,"chr") ||
            !strcmp(a->wert,"seq") ||
            !strcmp(a->wert,"read")) {
            fprintf(f,"    PUSHFUN   %s, 1 \n",a->wert);
        }
        else if (istkonst(a->wert)) {
            fprintf(f,"    PUSHFUN   %s, 2 \n",a->wert);
```

```
        }
        else {
            s = pos(a->wert,p);
            if (s == NULL) {
                puts("! Internal error 4");
                exit(4);
            }
            i = nchar(d - s->stackpos);
            fprintf(f,"    PUSH      %d ",d - s->stackpos);
            for (j = 15 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",s->altvar);
            if (notevaled(s->stackpos,e))
                fputs("    EVAL \n",f);
        }
        fprintf(f,"    MKAP      %d \n",n);
        return e;
    case ANW:
        if (a->strikt)
            e = E(a->rechts,p,d,e);
        else
            e = C(a->rechts,p,d,e);
        e = CS(a->links,p,d+1,n+1,e);
        return e;
    }
} /* CS */

struct stack *ES (a, p, d, n, e)
struct ausdruck *a;
struct stack    *p;
int             d, n;
struct stack    *e;
{
    struct comliste *k;
    struct stack    *s;
    int             i, j;

    switch (a->art) {
    case VAR:
        if ((k = istkomb(a->wert)) != NULL) {
            i = strlen(k->comname) + nchar(k->anzargs);
            fprintf(f,"    PUSHFUN   %s, %d ",
                k->comname,k->anzargs);
            for (j = 17 + i; j++ < 28; )
```

```
                putc(' ',f);
            fprintf(f,"%% %s \n",k->altcom);
        }
        else if (!strcmp(a->wert,"neg") ||
            !strcmp(a->wert,"not") ||
            !strcmp(a->wert,"ord") ||
            !strcmp(a->wert,"chr") ||
            !strcmp(a->wert,"seq") ||
            !strcmp(a->wert,"read")) {
            fprintf(f,"    PUSHFUN   %s, 1 \n",a->wert);
        }
        else if (istkonst(a->wert)) {
            fprintf(f,"    PUSHFUN   %s, 2 \n",a->wert);
        }
        else {
            s = pos(a->wert,p);
            if (s == NULL) {
                puts("! Internal error 5");
                exit(4);
            }
            i = nchar(d - s->stackpos);
            fprintf(f,"    PUSH      %d ",d - s->stackpos);
            for (j = 15 + i; j++ < 28; )
                putc(' ',f);
            fprintf(f,"%% %s \n",s->altvar);
            if (notevaled(s->stackpos,e))
                fputs("    EVAL \n",f);
        }
        fprintf(f,"    MKAP      %d \n",n);
        fprintf(f,"    EVAL         \n");
        return e;
    case ANW:
        if (a->strikt)
            e = E(a->rechts,p,d,e);
        else
            e = C(a->rechts,p,d,e);
        e = ES(a->links,p,d+1,n+1,e);
        return e;
    }
} /* ES */

char *upper (c)
char *c;
{
```

```
    int  i;
    char *cc;

    i = 0;
    cc = new(strlen(c)+1);
    while (c[i] != '\0') {
        cc[i] = (char)((int)(c[i]) - 32);
        i++;
    }
    cc[i] = '\0';
    return cc;
} /* upper */

B (a, p, d, e)
struct ausdruck *a;
struct stack    *p;
int             d;
struct stack    *e;
{
    struct ausdruck *aa;
    struct stack    *lab, *pp, *s, *ss;
    int             dd, i;
    char            *c, *cc;

    /* Generiert Code zur Ausfuehrung des  Basis- */
    /* wert-Rechnungen.                          */
    switch (a->art) {
    case KONST:
        fprintf(f,"    PUSHBASIC %s \n",a->wert);
        return;
    case ANW:
        aa = a;
        i = 0;
        while (aa->art == ANW) {
            i++;
            aa = aa->links;
        }
        if ((!strcmp(aa->wert,"neg") ||
            !strcmp(aa->wert,"not") ||
            !strcmp(aa->wert,"ord") ||
            !strcmp(aa->wert,"chr")) && i == 1) {
            B(a->rechts,p,d,e);
            c = upper(aa->wert);
            fprintf(f,"    %s \n",c);
```

```
            free(c);
            return;
        }
        else if (istkonst(aa->wert) && i == 2) {
            B(a->rechts,p,d,e);
            B(a->links->rechts,p,d,e);
            c = upper(aa->wert);
            fprintf(f,"    %s \n",c);
            free(c);
            return;
        }
        e = E(a,p,d,e);
        fputs("    GET \n",f);
        return;
    case FATBAR:
        e = E(a->links,p,d,e);
        c = newlabel();
        cc = newlabel();
        fprintf(f,"    JFAIL     %s \n",c);
        fprintf(f,"    GET          \n");
        fprintf(f,"    JUMP      %s \n",cc);
        fprintf(f,"%s: \n",c);
        B(a->rechts,p,d,e);
        fprintf(f,"%s: \n",cc);
        free(c);
        free(cc);
        return;
    case IF:
        B(a->links,p,d,e);
        c = newlabel();
        cc = newlabel();
        fprintf(f,"    JFALSE    %s \n",c);
        B(a->rechts,p,d,e);
        fprintf(f,"    JUMP      %s \n",cc);
        fprintf(f,"%s: \n",c);
        B(a->rechts->hinten,p,d,e);
        fprintf(f,"%s: \n",cc);
        free(c);
        free(cc);
        return;
    case CASE:
        e = E(a->links,p,d,e);
        fputs("    CASEJUMP  ",f);
        aa = a->rechts;
```

```
lab = NULL;
while (aa != NULL) {
    /* Markenliste fuer CASEJUMP erzeugen */
    s = (struct stack *) new(sizeof(struct stack));
    s->var = newlabel();
    s->next = NULL;
    fprintf(f,"(%d,%s), ",aa->standardtyp,s->var);
    if (lab == NULL)
        lab = s;
    else {
        ss = lab;
        while (ss->next != NULL)
            ss = ss->next;
        ss->next = s;
    }
    aa = aa->hinten;
}
c = newlabel();
cc = newlabel();
fprintf(f,"%s \n",c);
aa = a->rechts;
s = lab;
while (aa != NULL) {
    /* Uebersetzen der CASE-Faelle */
    fprintf(f,"%s: \n",s->var);
    pp = Xr(aa,p,d);
    dd = d + aa->stelligkeit;
    B(aa->links,pp,dd,e);
    if (aa->stelligkeit != 0)
        fprintf(f,"    POP       %d \n",
            aa->stelligkeit);
    fprintf(f,"    JUMP      %s \n",cc);
    freestack(pp);
    s = s->next;
    aa = aa->hinten;
}
/* Default -- Fehler: */
fprintf(f,"%s: \n    END \n",c);
/* Marke nach CASE: */
fprintf(f,"%s: \n",cc);
free(c);
free(cc);
freestack(lab);
return;
```

```
    case LET:
        if (a->definition->defstrikt)
            e = E(a->definition->varwert,p,d,e);
        else
            e = C(a->definition->varwert,p,d,e);
        pp = copystack(p);
        pp = append(p,a->definition->varname,
            a->definition->sym->altname,++d);
        B(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    POP       1 \n");
        return;
    case LETREC:
        pp = Xr(a,p,d);
        dd = d + a->stelligkeit;
        CLetrec(a,pp,dd);
        B(a->links,p,d,e);
        freestack(pp);
        fprintf(f,"    POP       %d \n",dd-d);
        return;
    default:
        e = E(a,p,d,e);
        fputs("    GET \n",f);
    }
} /* B */

struct stack *R (a, p, d, e)
struct ausdruck *a;
struct stack    *p;
int             d;
struct stack    *e;
{
    struct ausdruck *aa;
    struct comliste *k;
    struct defliste *df;
    struct stack    *lab, *pp, *s, *ss;
    int             dd, i, j;
    char            *c;

    /* Generiert Code zur Anwendung eines  Super- */
    /* kombinators auf seine d Argumente.         */
    switch (a->art) {
    case KONST:
        B(a,p,d,e);
```

```
        fprintf(f,"    UPDBASIC  %d \n",d);
        if (d != 0)
            fprintf(f,"    POP       %d \n",d);
        fprintf(f,"    RETURN       \n");
        return e;
    case VAR:
    case CONS:
    case SELECT:
        e = E(a,p,d,e);
        fprintf(f,"    UPDATE    %d \n",d+1);
        if (d != 0)
            fprintf(f,"    POP       %d \n",d);
        fprintf(f,"    UNWIND       \n");
        return e;
    case FAIL:
        fprintf(f,"    PUSHFAIL     \n");
        fprintf(f,"    UPDATE    %d \n",d+1);
        if (d != 0)
            fprintf(f,"    POP       %d \n",d);
        fprintf(f,"    RETURN       \n");
        return e;
    case FATBAR:
        e = E(a->links,p,d,e);
        c = newlabel();
        fprintf(f,"    JFAIL     %s \n",c);
        fprintf(f,"    UPDATE    %d \n",d+1);
        if (d != 0)
            fprintf(f,"    POP       %d \n",d);
        fprintf(f,"    UNWIND       \n");
        fprintf(f,"%s: \n",c);
        free(c);
        e = R(a->rechts,p,d,e);
        return e;
    case ANW:
        aa = a;
        i = 0;
        while (aa->art == ANW) {
            i++;
            aa = aa->links;
        }
        if (istkonst(aa->wert)) {
            if ((!strcmp(aa->wert,"neg") ||
                !strcmp(aa->wert,"not") ||
                !strcmp(aa->wert,"ord") ||
```

```
            !strcmp(aa->wert,"chr")) && i == 1)
            j = 1;
        else if (i == 2 && strcmp(aa->wert,"seq") &&
            strcmp(aa->wert,"neg") &&
            strcmp(aa->wert,"not") &&
            strcmp(aa->wert,"ord") &&
            strcmp(aa->wert,"chr") &&
            strcmp(aa->wert,"read"))
            j = 1;
        else
            j = 0;
        if (j) {
            B(a,p,d,e);
            fprintf(f,"    UPDBASIC  %d \n",d);
            if (d != 0)
                fprintf(f,"    POP       %d \n",d);
            fprintf(f,"    RETURN       \n");
            return e;
        }
    }
    if ((k = istkomb(aa->wert)) != NULL &&
        k->anzargs == i) {
        /* Tail Call */
        dd = d;
        aa = a;
        while (aa->art == ANW) {
            e = C(aa->rechts,p,dd++,e);
            aa = aa->links;
        }
        if (d != 0)
            fprintf(f,"    SQUEEZE   %d, %d \n",
                i,d);
        fprintf(f,"    JUMP      %s ",aa->wert);
        for (j = 15 + strlen(aa->wert); j++ < 28; )
            putc(' ',f);
        fprintf(f,"%% %s (combinator) \n",k->altcom);
        return e;
    }
    e = RS(a,p,d,0,e);
    return e;
case IF:
    B(a->links,p,d,e);
    c = newlabel();
    fprintf(f,"    JFALSE    %s \n",c);
```

```
        e = R(a->rechts,p,d,e);
        fprintf(f,"%s: \n",c);
        free(c);
        e = R(a->rechts->hinten,p,d,e);
        return e;
    case CASE:
        e = E(a->links,p,d,e);
        fputs("    CASEJUMP  ",f);
        aa = a->rechts;
        lab = NULL;
        while (aa != NULL) {
            /* Markenliste fuer CASEJUMP erzeugen */
            s = (struct stack *) new(sizeof(struct stack));
            s->var = newlabel();
            s->next = NULL;
            fprintf(f,"(%d,%s), ",aa->standardtyp,s->var);
            if (lab == NULL)
                lab = s;
            else {
                ss = lab;
                while (ss->next != NULL)
                    ss = ss->next;
                ss->next = s;
            }
            aa = aa->hinten;
        }
        c = newlabel();
        fprintf(f,"%s \n",c);
        aa = a->rechts;
        s = lab;
        while (aa != NULL) {
            /* Uebersetzen der CASE-Faelle */
            fprintf(f,"%s: \n",s->var);
            pp = Xr(aa,p,d);
            dd = d + aa->stelligkeit;
            e = R(aa->links,pp,dd,e);
            freestack(pp);
            s = s->next;
            aa = aa->hinten;
        }
        /* Default: */
        fprintf(f,"%s: \n",c);
        fprintf(f,"    PUSHFAIL \n");
        fprintf(f,"    UPDATE    %d \n",d+1);
```

```
        if (d != 0)
            fprintf(f,"    POP       %d \n",d);
        fprintf(f,"    RETURN       \n");
        freestack(lab);
        free(c);
        return e;
    case LET:
        df = a->definition;
        if (df->defstrikt)
            e = E(df->varwert,p,d,e);
        else
            e = C(df->varwert,p,d,e);
        pp = copystack(p);
        pp = append(pp,df->varname,df->sym->altname,++d);
        e = R(a->links,pp,d,e);
        freestack(pp);
        return e;
    case LETREC:
        pp = Xr(a,p,d);
        d += a->stelligkeit;
        CLetrec(a,pp,d);
        e = R(a->links,pp,d,e);
        freestack(pp);
        return e;
    }
} /* R */

standard1 (fun)
char *fun;
{
    char *c;

    fprintf(f,"%s: \n",fun);
    fputs("    EVAL \n",f);
    fputs("    GET  \n",f);
    c = upper(fun);
    fprintf(f,"    %s \n",c);
    fputs("    UPDBASIC  0 \n",f);
    fputs("    RETURN    \n\n",f);
    free(c);
} /* standard1 */

standard2 (fun)
char *fun;
```

```
{
    char *c;

    fprintf(f,"%s: \n",fun);
    fputs("    PUSH      1 \n",f);
    fputs("    EVAL \n",f);
    fputs("    GET  \n",f);
    fputs("    EVAL \n",f);
    fputs("    GET  \n",f);
    c = upper(fun);
    fprintf(f,"    %s \n",c);
    fputs("    UPDBASIC  1 \n",f);
    fputs("    POP       1 \n",f);
    fputs("    RETURN    \n\n",f);
    free(c);
} /* standard2 */

codegen (name)
char *name;
{
    struct comliste *k;
    struct defliste *df;
    struct stack    *e, *sb, *s;
    int             i;

    labelnummer = 1;
    /* Vorspann: */
    fprintf(f,"%% Program \"%s\" \n\n",name);
    fprintf(f,"    BEGIN     Main \n");
    fprintf(f,"    EVAL  \n");
    fprintf(f,"    PRINT \n");
    fprintf(f,"    END   \n\n");
    fprintf(f,"Main: \n");
    fprintf(f,"%% Main expression to be evaluated. \n");
    if (msg == 2)
        puts("Generating code for main expression");
    e = R(prog,NULL,0,NULL);
    freestack(e);
    k = comb;
    while (k != NULL) {
        if (msg == 2)
            printf("Generating code for combinator \"%s\"\n",
                k->altcom);
        fprintf(f,"\n%s: \n",k->comname),
```

```
            fprintf(f,"%% Combinator (%d arg",k->anzargs);
        if (k->anzargs != 1)
            putc('s',f);
        fprintf(f,"). Original: \"%s\" (%d arg",
            k->altcom,k->altanz);
        if (k->altanz != 1)
            putc('s',f);
        fputs("). \n% Argument",f);
        if (k->anzargs != 1)
            putc('s',f);
        putc(':',f);
        aus_defs(k->args,f,0);
        putc('\n',f);
        i = k->anzargs;
        e = NULL;
        sb = NULL;
        df = k->args;
        while (df != NULL) {
            s = (struct stack *) new(sizeof(struct stack));
            s->var = df->varname;
            s->altvar = df->sym->altname;
            s->stackpos = i;
            s->next = sb;
            sb = s;
            if (df->defstrikt) {
                if (i == k->anzargs)
                    fprintf(f,"    EVAL         \n");
                else {
                    fprintf(f,"    PUSH      %d \n",
                        k->anzargs - i);
                    fprintf(f,"    EVAL         \n");
                    fprintf(f,"    POP       1  \n");
                }
                e = append(e,NULL,NULL,i);
            }
            i--;
            df = df->next;
        }
        e = R(k->koerper,sb,k->anzargs,e);
        freestack(e);
        freestack(sb);
        k = k->next;
    }
    fputs("\n% Built-in combinators \n\n",f);
```

```
if (sfunc[0])
    standard1("not");
if (sfunc[1])
    standard1("neg");
if (sfunc[2])
    standard2("add");
if (sfunc[3])
    standard2("sub");
if (sfunc[4])
    standard2("mult");
if (sfunc[5])
    standard2("div");
if (sfunc[6])
    standard2("mod");
if (sfunc[7])
    standard2("and");
if (sfunc[8])
    standard2("or");
if (sfunc[9])
    standard2("lt");
if (sfunc[10])
    standard2("leq");
if (sfunc[11])
    standard2("eq");
if (sfunc[12])
    standard2("neq");
if (sfunc[13])
    standard2("geq");
if (sfunc[14])
    standard2("gt");
if (sfunc[15])
    standard1("ord");
if (sfunc[16])
    standard1("chr");
if (sfunc[17]) {
    fputs("seq: \n",f);
    fputs("    EVAL        \n",f);
    fputs("    POP       1 \n",f);
    fputs("    EVAL        \n",f);
    fputs("    UPDATE    1 \n",f);
    fputs("    UNWIND    \n\n",f);
}
if (sfunc[18]) {
    fputs("read: \n",f);
```

```
fputs("    PUSH      0 \n",f);
fputs("    EVAL        \n",f);
fputs("    PUSHFUN   Revalarg, 1 \n",f);
fputs("    MKAP        \n",f);
fputs("    EVAL        \n",f);
fputs("    OPEN        \n",f);
fputs("    PUSHFUN   Rread, 1 \n",f);
fputs("    MKAP        \n",f);
fputs("    EVAL        \n",f);
fputs("    UPDATE    2 \n",f);
fputs("    POP       1 \n",f);
fputs("    RETURN    \n\n",f);
fputs("Revalarg: \n",f);
fputs("    PUSH      0 \n",f);
fputs("    CASEJUMP  (1,LRnil), ",f);
fputs("(2,LRcons), LRnil \n",f);
fputs("LRnil: \n",f);
fputs("    UPDATE    1 \n",f);
fputs("    RETURN      \n",f);
fputs("LRcons: \n",f);
fputs("    PUSHFUN   Revalarg, 1 \n",f);
fputs("    MKAP        \n",f);
fputs("    EVAL        \n",f);
fputs("    PUSH      1 \n",f);
fputs("    EVAL        \n",f);
fputs("    CONS      2, 2 \n",f);
fputs("    UPDATE    3 \n",f);
fputs("    POP       2 \n",f);
fputs("    RETURN    \n\n",f);
fputs("Rread: \n",f);
fputs("    PUSH      0 \n",f);
fputs("    READ        \n",f);
fputs("    JFAIL     LRend \n",f);
fputs("    PUSH      1 \n",f);
fputs("    PUSHFUN   Rread, 1 \n",f);
fputs("    MKAP        \n",f);
fputs("    PUSH      1 \n",f);
fputs("    CONS      2, 2 \n",f);
fputs("    UPDATE    3 \n",f);
fputs("    POP       2 \n",f);
fputs("    RETURN      \n",f);
fputs("LRend: \n",f);
fputs("    CONS      1, 0 \n",f);
fputs("    UPDATE    2 \n",f);
```

```
        fputs("    POP       1 \n",f);
        fputs("    RETURN    \n\n",f);
    }
} /* codegen */
```

Literaturverzeichnis

[Aho 1988] Aho, A.V., R. Sethi, J.D. Ullman: Compilerbau. Teil 1. Addison-Wesley, Bonn 1988.

[Augustsson 1987] Augustsson, L.: Compiling Lazy Functional Languages, Part II. Ph.D. Thesis. Chalmers University of Technology, Department of Computer Sciences, Göteborg 1987.

[Augustsson 1989] Augustsson, L., T. Johnsson: The Chalmers Lazy-ML Compiler. Computer Journal Vol. 32 (1989) No. 2, pp. 127 – 141.

[Burstall 1980] Burstall, R.M., D.B. MacQueen, D.T. Sanella: HOPE: An Experimental Applicative Language. Proceedings of the 1980 LISP Conference. Stanford, California 1980, pp. 136 – 143.

[Field 1988] Field, A.J., P.G. Harrison: Functional Programming. Addison-Wesley, Wokingham 1988.

[Fradet 1991] Fradet, P., D. le Métayer: Compiling of Functional Languages by Program Transformation. ACM Transactions on Programming Languages and Systems Vol. 13 (1991) No. 1, pp. 21 – 51.

[Grossmann 1990] Großmann, D.: Übersetzungsprinzipien funktionaler Programmiersprachen. Hochschul-Abschlußarbeit. Technische Universität Chemnitz, 1990.

[Henderson 1980] Henderson, P.: Functional Programming: Application and Implementation. Prentice-Hall, London 1980.

[Johnsson 1987] Johnsson, T.: Compiling Lazy Functional Languages. Ph.D. Thesis. Chalmers University of Technology, Department of Computer Sciences, Göteborg 1987.

[Joy 1988] Joy, M., T. Axford: A Standard for a Graph Representation for Functional Programs. SIGPLAN Notices Vol. 23 (1988) No. 1, pp. 75 – 82.

[Loogen 1990] Loogen, R.: Parallele Implementierung funktionaler Programmiersprachen. Informatik-Fachberichte 232, Springer-Verlag, Berlin 1990.

[Peyton Jones 1987] Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, New Jersey 1987.

[Peyton Jones 1988] Peyton Jones, S.L.: FLIC — A Functional Language Intermediate Code. SIGPLAN Notices Vol. 23 (1988) No. 8, pp. 30 – 48.

[Turner 1985] Turner, D.A.: Miranda: A Non-strict Functional Language with Polymorphic Types. Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy. Lecture Notes in Computer Science Vol. 201, Springer-Verlag, Berlin 1985, pp. 1 – 16.

[Turner 1987] Turner, D.A.: An Introduction to Miranda. Appendix to Peyton Jones: The Implementation of Functional Programming Languages [Peyton Jones 1987].

[Wikström 1987] Wikström, Å.: Functional Programming Using Standard ML. Prentice-Hall, London 1987.

[Wray 1989] Wray, S.C., J. Fairbrain: Non-strict Languages — Programming and Implementation. Computer Journal Vol. 32 (1989) No. 2, pp. 142 – 151.