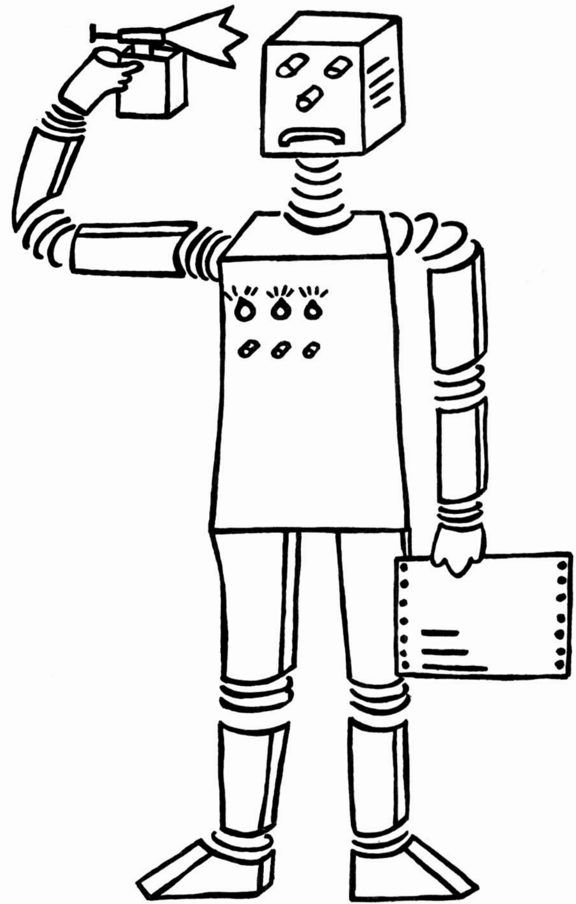
SEKI-Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



SASL: Implementierung einer rein funktionalen Sprache mit Lazy Evaluation

Klaus Nökel
Robert Rehbold

November 1986 SWP 86-07

SASL:
Implementierung einer rein funktionalen Programmiersprache mit Lazy Evaluation

Klaus Nökel
Robert Rehbold

Universität Kaiserslautern
Fachbereich Informatik
Postfach 30 49
6750 Kaiserslautern
West Germany

noekel@uklirb.UUCP
oder rehbold@uklirb.UUCP

SEKI Working Paper
SWP 86-07

Oktober 1986

SASL: Implementierung einer rein funktionalen Programmiersprache mit Lazy Evaluation

Zusammenfassung:

Dieser Report beschreibt eine Implementierung von SASL in Lisp, die auf Variablenabstraktion und Kombinatorgraph-Reduktion beruht. Nach einer Einführung in SASL schildert ein Kapitel über Variablenabstraktion ausführlich, wie aus SASL-Programmen effektive Kombinatorausdrücke erzeugt werden, einschließlich der Behandlung von lokalen Definitionen, mehrzeiligen Funktionsdefinitionen mit Fallunterscheidung durch Patternmatching sowie von ZF-Mengennotation. Zur Auswertung der so erzeugten Kombinatorgraphen wird die konkrete Implementierung einer effizienten Reduktionsmaschine vorgestellt.

Abstract:

This report presents an implementation of SASL in Lisp which uses abstraction of variables and reduction of combinator graphs. Following an introduction to SASL a section shows in detail how SASL programs can be translated into compact combinator graphs, with special emphasis on the methods used to deal with local definitions, function definitions that spread over several lines making use of pattern matching for case selection, and with ZF-set notation. The second part of the report gives a detailed account of the concrete implementation of an efficient algorithm for the reduction of combinator graphs.

Inhalt:

1. **SASL - eine Übersicht** 1
   - 1.1 Die Objekte der Sprache 2
   - 1.2 Ausdrücke 4
     - 1.2.1 Zulässige SASL-Ausdrücke 5
     - 1.2.2 Referential Transparency 7
   - 1.3 Funktionen 9
     - 1.3.1 Funktionsdefinitionen 9
     - 1.3.2 Funktionale - Curried Functions 10
     - 1.3.3 Lazy Evaluation - Nicht-strikte Funktionen 11
   - 1.4 Listen 12
     - 1.4.1 Unendliche Listenstrukturen 13
     - 1.4.2 Schwach-terminierende Listen 14
     - 1.4.3 ZF-Ausdrücke 15
   - 1.5 Der Sprachumfang unseres SASL-Interpreters 18
     - 1.5.1 Die Datentypen 18
     - 1.5.2 Das SASL-Prelude 18
     - 1.5.3 Namelist-Definitionen 19

2. **Die Abstraktion** 20
   - 2.1 Die Äquivalenz von SASL-Ausdrücken und Kombinatorausdrücken 20
   - 2.2 Grundlagen der Variablenabstraktion 21
   - 2.3 Das Entfernen von Variablen aus Ausdrücken 22
   - 2.4 Variablenabstraktion mit Kombinatoren 24
     - 2.4.1 Die Grundkombinatoren I, K und S 24
     - 2.4.2 Optimierung durch weitere Regeln und Kombinatoren 25
   - 2.5 Abstraktion von Listen 27
   - 2.6 Die Abstraktion von WHERE-Ausdrücken 30
     - 2.6.1 Alternativen bei der Behandlungsweise 30
     - 2.6.2 Die direkte Abstraktion von WHERE-Ausdrücken 31
   - 2.7 Fallunterscheidung durch Definition und Pattern-Matching 33
   - 2.8 Die Behandlung von ZF-Ausdrücken 40
   - 2.9 Zusammenfassung des Abstraktionsvorganges 45
   - 2.10 Übersicht über die verwandten Abstraktionsregeln und Kombinatoren ... 46

3. **Kombinatorgraphen und ihre Reduktion** 48
   - 3.1 Prinzipielle Vorgehensweise 48
     - 3.1.1 Kombinatorgraphen 49
     - 3.1.2 Anwendung einer Reduktionsregel im Kombinatorgraphen 51
     - 3.1.3 Auswahl des zu reduzierenden Teilgraphen 55
     - 3.1.4 Die Ausgabekomponente 58
     - 3.1.5 Ein Reduktionsalgorithmus 58
   - 3.2 Datenstrukturen und Steuerung des Reduktionsalgorithmus 62
     - 3.2.1 Abbildung der Kombinatorgraphen in Lisp-Listen . 62
     - 3.2.2 Der Stack 63
     - 3.2.3 Die Funktion REDUCE 67
     - 3.2.4 Die Funktion REDEXPO 69
     - 3.2.5 Eine generische Form für Reduktionsregeln 72
     - 3.2.6 Die Behandlung von Laufzeitfehlern 75

4. Weitere Einzelheiten der Implementation 77
4.1 Die Entscheidung für Lisp 77
4.2 Effizienzsteigernde Maßnahmen 78
4.2.1 Recycling von CONS-Zellen 78
4.2.2 Superkombinatoren 79
4.2.3 Telescoping 8Ø
4.3 Das System aus der Sicht des Benutzers 81
4.4 Die Einbettung von SASL in Lisp 82

Anhang:

Das implementierte SASL-Prelude 84

Literaturverzeichnis 95

1. SASL - eine Übersicht

Die Programmiersprache SASL (= St. Andrews Static Language) wurde im Jahre 1976 von David A. Turner an der Universität von St. Andrews entwickelt. Ursprünglich in erster Linie für Ausbildungszwecke bestimmt, sollte SASL einen möglichst reinen, funktionalen Programmierstil fördern. Man versuchte, dieser Zielsetzung gerecht zu werden, indem man sich an der verbreitetesten funktionalen Programmiersprache Lisp orientierte und daraus alle nichtfunktionalen Elemente (imperative, d.h. seiteneffektabhängige "Funktionen", iterative Kontrollstrukturen etc.) entfernte. Eine von üblichen Lisp-Implementationen verschiedene Auswertungsstrategie erlaubte, alle Objekte der Sprache (auch Funktionen) völlig gleichwertig zu verwenden. Zudem eröffnete diese Auswertungsstrategie die Möglichkeit, sinnvoll mit unendlichen Listenstrukturen zu operieren. Schließlich versah man SASL mit einer im Vergleich zu Lisp bedeutend angenehmeren Syntax, die der gewohnten mathematischen Notation für Funktionen sehr nahe kommt. Man hoffte, damit logische Fehler beim Übergang von der Problemspezifikation zum Programm weitgehend auszuschalten, indem dem Programmierer ein Wechsel der Abstraktionsebene erspart blieb.

Die Weiterentwicklung von SASL, KRC (Kent Recursive Calculator) enthielt nur wenig Neues. Außer einer leicht veränderten Syntax erlaubte es als wichtigsten Zusatz die Verwendung von Guards (d.i. syntaktischer Zucker bei der Funktionsdefinition). Wesentliche Änderungen enthält dagegen Miranda, eine ebenfalls rein funktionale Programmiersprache, in der Turner KRC um ein polymorphes Typkonzept vergleichbar ML (Meta-Language /Gordon79/) erweiterte, dessen Einhaltung streng überwacht wird. Einige elegante Programmiertechniken aus SASL und KRC sind jedoch aufgrund dieser starren Typkontrolle nicht mehr möglich.

Schon aus Platzgründen kann diese Übersicht keine vollständige Sprachbeschreibung von SASL geben. Vielmehr werden in den folgenden Abschnitten die wesentlichen Merkmale von SASL kurz vorgestellt, wobei der Schwerpunkt auf denjenigen Aspekten liegt, die über verwandte Lisp-Dialekte hinausgehen. Der Leser, der eine tiefergehende Darstellung sucht, sei auf /Turner83/ und /Richards84/ verwiesen. Die erste Quelle stellt die neueste Referenz für den von Turner selbst vorgeschlagenen Standard dar, während in der zweiten Fundstelle der Dialekt ARC-SASL beschrieben wird, der am Austin Research Center der Firma Burroughs entwickelt wurde. Da sich beide Versionen sowohl im Umfang der

vordefinierten Funktionen als auch teilweise in den Definitionen selbst unterscheiden, dient der Abschnitt 1.5 der Abgrenzung des in unserer Implementation realisierten Sprachumfangs.

1.1 Die Objekte der Sprache

Die Objekte, auf denen SASL-Programme operieren, lassen sich in sechs Klassen einteilen:

- Zahlen,
- boolesche Werte,
- Zeichen,
- Funktionen,
- Listen und
- der Wert "undefiniert".

Objekte aller Typen können in völlig gleicher Weise verwendet werden, d.h. sie können

- mit Namen versehen werden,
- Argument oder Wert von Funktionen sein,
- Listenkomponenten sein oder
- Wert eines Ausdrucks sein.

Konstanten der meisten Typen besitzen eine externe Repräsentation, die bei ihrer Ein- und Ausgabe benutzt wird. Ihre Form wird im folgenden neben weiteren Besonderheiten der einzelnen Typen vorgestellt.

Für alle Datentypen (mit Ausnahme des Wertes "undefiniert") existieren in SASL charakteristische Prädikate, die die Zugehörigkeit eines gegebenen Objekts zu dem jeweiligen Typ testen.

Die Zahlen umfassen mindestens eine Teilmenge der positiven und negativen ganzen Zahlen, je nach Implementation auch Gleitkommazahlen. Die externe Repräsentation einer Zahl ist ihre Dezimaldarstellung.

Die booleschen Werte "wahr" und "falsch" werden durch die vordefinierten Bezeichner TRUE und FALSE dargestellt.

Funktionen in SASL können ihrerseits Funktionen sowohl als Argumente erhalten als auch als Werte liefern. Eine externe Darstellung für Funktionen gibt es nicht, lediglich die vordefinierten und die durch den Benutzer hinzugefügten Funktionen können durch ihr Funktionssymbol bezeichnet werden. Es ist aber zu beachten, daß es keine Äquivalenz zwischen Funktionen und etwa Listen einer besonderen Form wie in Lisp gibt. Daher ist die dort häufig angewandte Methode, Funktionen mit Listenoperationen zu konstruieren und mit APPLY oder EVAL weiterzuverwenden, auf SASL-Verhältnisse nicht übertragbar. Aus diesem Grunde gibt es auch keine Entsprechung für die Funktionen EVAL und APPLY.

Aus Objekten beliebiger Typen können Listen gebildet werden. Für den Aufbau der Listen und den Zugriff auf die Komponenten stehen Operatoren zur Verfügung, die den Lisp-Funktionen CONS, CAR und CDR entsprechen. Es handelt sich dabei um den Konstruktor ":", der ein Element am Kopf einer Liste anfügt, sowie "hd" und "tl", die Kopf und Rest einer Liste liefern. Wie wir noch sehen werden (vgl. Abschnitt 1.4.1), können die entstehenden Listenstrukturen auch unendlich sein. Die leere Liste wird als [] dargestellt. Um endliche Listen übersichtlich notieren zu können, führt man für sie eine alternative Schreibweise ein: so kann die Liste aus den Elementen $a_1,...,a_n$ statt durch

$a_1:a_2:...:a_n:[]$

auch durch

$[a_1,a_2,...,a_n]$

angegeben werden.

Als Zeichen sind üblicherweise die ASCII-Zeichen vorhanden, sie werden zur Unterscheidung von Namen mit einem vorangesetzten Prozentzeichen dargestellt:

..., %a, %b, ..., %A, %B, ...

Zeichenketten aus mehr als einem Zeichen bilden zwar keinen der Grundtypen, können jedoch als Listen von Zeichen gebildet werden, abkürzend schreibt man dabei

für [%h,%a,%l,%l,%o] auch "hallo".

Es ist wichtig, zwischen Strings der Länge 1 und Zeichen zu differenzieren; %A ist ein Zeichen und somit etwas anderes als "A" = [%A] = %A : [], der einelementigen Liste mit dem Element %A.

Schließlich existiert noch der Wert "undefiniert", der keinem der anderen Datentypen angehört. Da dieser Wert i.d.R. nur in einer Fehlersituation auftritt, existiert keine Repräsentation für die Eingabe. ARC-SASL verfügt jedoch über eine Standardfunktion "error", die eine Fehlermeldung auslöst und "undefiniert" liefert. Als Schreibabkürzung für "undefiniert" in Aussagen verwenden wir das Symbol ⊥, es ist jedoch nicht Bestandteil von SASL. Der Wert ⊥ wird beispielsweise Ausdrücken zugeordnet, die nicht den Typrestriktionen der in ihnen vorkommenden Funktionen genügen, so ist z.B.

3 + TRUE = ⊥

Nicht immer läßt sich in endlicher Zeit entscheiden, ob der Wert eines Ausdrucks "undefiniert" ist. So kann dieser Wert auch in der Form einer nichtterminierenden Berechnung auftreten. Entsprechend unterschiedlich behandelt das SASL-System diesen Wert. Während in Situationen wie der oben dargestellten davon ausgegangen wird, daß ein Fehler vorliegt, und eine entsprechende Fehlermeldung ausgegeben wird, kann eine solche Diagnose im Fall einer Endlosschleife prinzipiell nicht gegeben werden, es erfolgt gar keine Ausgabe. Da ansonsten ⊥ unabhängig von seiner Herkunft völlig uniform behandelt werden soll, ist es mit keinen Operationen typverträglich und besitzt auch kein Typprädikat.

1.2 Ausdrücke

Der Dialog des Benutzers mit dem SASL-Interpreter besteht genau wie bei einem Lisp-Interpreter darin, dem System Ausdrücke einzugeben, die anschließend ausgewertet werden. Sofern für ihre Werte externe Repräsentationen existieren, werden diese vom System ausgegeben, bevor die nächste Eingabe erwartet wird. Da abgesehen von Funktionsdefinitionen, die in Abschnitt 1.3.1 beschrieben werden, die gesamte Kommunikation über SASL-Ausdrücke stattfindet, muß geklärt werden, welche Eingaben gültig sind. Die folgende Beschreibung zeigt lediglich auf, welche Formen syntaktisch zulässig sind. Damit ein Ausdruck einen von ⊥ verschiedenen Wert besitzt, müssen in einigen Fällen zusätzliche Bedingungen erfüllt sein (Beachtung der Stelligkeit der Funktionen usw.).

1.2.1 Zulässige SASL-Ausdrücke

Eine Beschreibung der legalen SASL-Ausdrücke in Backus-Naur-Form findet man in /Richards84, S. 85ff/. Für unsere Zwecke reicht die folgende vereinfachte Charakterisierung der Menge EXP der Ausdrücke:

1.) Konstanten $\in$ EXP

2.) Namen $\in$ EXP

3.) $E_1, E_2 \in$ EXP $\Longrightarrow E_1\ E_2 \in$ EXP (**Applikation**)

4.) $E \in$ EXP $\Longrightarrow (E) \in$ EXP

5.) $E \in$ EXP, $Fktdef_1, \ldots, Fktdef_n$ Funktionsdefinitionen (s. 1.3.1)

$$\Longrightarrow \text{E WHERE } Fktdef_1 \\ Fktdef_2 \\ \ldots \\ Fktdef_n$$

$\in$ EXP (**WHERE-Ausdruck**)

6.) $E \in$ EXP, $G_1, \ldots, G_n$ Generatoren, wobei
$G_i = E_i \in$ EXP oder
$G_i = V_i \leftarrow E_i$, $E_i \in$ EXP, V_i Variable

$\Longrightarrow [E;\ G_1;\ \ldots;\ G_n] \in$ EXP (**ZF-Ausdruck**)

(für die Variablen V_i gelten weitere Nebenbedingungen, die in dem Abschnitt 1.4.3 über ZF-Ausdrücke erläutert werden.)

Applikationen decken zwei auf den ersten Blick völlig verschiedene Bedeutungen ab. Ist E_1 eine Funktion, so bezeichnet man mit $E_1\ E_2$ die Anwendung der Funktion auf das Argument E_2. Alternativ kann es sich aber bei E_1 auch um eine Liste handeln. Ist dann der Wert von E_2 eine Zahl, so bezeichnet $E_1\ E_2$ die E_2-te Komponente von E_1.

Zweideutigkeiten in der Interpretation ungeklammerter Ausdrücke werden dadurch beseitigt, daß implizit Linksklammerung angenommen wird; für die vordefinierten Operatoren ist zudem noch eine zwölfstufige Bindungshierarchie festgelegt. Soll von diesem

Grundsatz abgewichen werden, muß geklammert werden.

Die implizite Linksklammerung erlaubt, auch Applikationen von Funktionen auf mehrere Argumente in Präfixnotation und ohne Klammern zu schreiben. Von dieser Regel weichen allerdings einige der vordefinierten Standard-Operatoren ab, um die Übersichtlichkeit zu erhöhen. In diese Gruppe fallen unter anderem die arithmetischen Operationen und der Listenkonstruktor ":".

In den Punkten 5.) und 6.) schließlich werden zwei Arten von Ausdrücken eingeführt, die SASL-spezifisch sind.

WHERE-Ausdrücke dienen im Gegensatz zu globalen Definitionen dazu, Namen einzuführen, deren Gültigkeit sich nur auf einen einzelnen Ausdruck erstreckt.

Der Ausdruck, in dem die lokalen Definitionen gelten sollen wird von diesen durch das nachgestellte Schlüsselwort WHERE getrennt. Auf das WHERE können beliebig viele Funktionsdefinitionen folgen, deren Form im Abschnitt 1.3.1 beschrieben ist. Ein Name darf jedoch im Rahmen eines WHERE-Ausdrucks nur einmal definiert werden. Von dieser Einschränkung bleiben Definitionen in über- oder untergeordneten WHERE-Ausdrücken unberührt. Insbesondere können die Funktionen nullstellig sein, so daß mit Hilfe von WHERE auch lokale Konstanten definiert werden können. Der Geltungsbereich der lokalen Bezeichner umfaßt genau den WHERE-Ausdruck, d.h. alle Funktionsdefinitionen rechts des WHERE sowie den links des WHERE stehenden Ausdruck. Zum Beispiel hat der Ausdruck

```
f 4 WHERE f x = x + (x WHERE x = 6)
```

den Wert 10, da nur das x unmittelbar vor dem zweiten WHERE im Gültigkeitsbereich der lokalen Definition liegt.

Wenn in einem Ausdruck E ein Name x verwendet wird, so bestimmt sich sein Wert nach folgender Festlegung (da die Fälle nicht disjunkt sind, gilt der erste zutreffende Fall in der angegebenen Reihenfolge):

(i) Falls E ein WHERE-Ausdruck ist und x darin definiert wird, so gilt diese Definition.

(ii) Falls E der Körper einer Funktionsdefinition ist und x als formaler Parameter auftritt, so ist der Wert der jeweilige aktuelle Parameter.

(iii) Wenn E Subterm von anderen Ausdrücken ist, so ist der Wert von x in E gleich dem Wert von x in dem minimalen Ausdruck E', dessen Subterm E ist.

(iv) Wenn x global definiert ist, so gilt diese Definition.

(v) x hat den Wert ⊥.

Man beachte, daß die Bestimmung der gültigen Definition rein statisch erfolgt, so daß in SASL im Gegensatz zu vielen Lisp-Implementationen eine korrekte Realisierung der λ-Bindung vorliegt und die CLOSURE-Problematik nicht existiert.

ZF-Ausdrücke werden im Zusammenhang mit Listenstrukturen im Abschnitt 1.4.3 erklärt.

1.2.2 Referential Transparency

Allen zusammengesetzten Ausdrücken ist eine Eigenschaft gemeinsam, die direkt aus der Seiteneffektfreiheit der Sprache und der konsequenten Realisierung statischer Bindung folgt.

> Ein zusammengesetzter SASL-Ausdruck E(T) enthalte einen Subausdruck T. Sei ferner T' ein SASL-Ausdruck mit Wert(T) = Wert(T'). Dann gilt auch:
> Wert(E(T)) = Wert(E(T')).

Teilausdrücke mit gleichem Wert dürfen also beliebig ausgetauscht werden, ohne den Wert des umgebenden Ausdrucks zu ändern. Da also Ausdrücke ohne Einschränkung dieselben Substitutionseigenschaften haben wie Terme in der Logik, können Beweisverfahren aus der allgemeinen Mathematik ohne aufwendige Hilfskonstruktionen übernommen werden.

Weitere wichtige Eigenschaften ergeben sich als Folgerungen. So hängt der Wert eines Ausdrucks nur von den Werten der Variablen ab, in deren Gültigkeitsbereich er liegt, nicht jedoch von einer bestimmten Auswertungsreihenfolge oder gar von etwaigen Namenskonflikten (daher auch die Bezeichnung referential transparency). Aus diesem Grunde ist die Gleichheit zweier SASL-Objekte eine reine Wertgleichheit, das entgegengesetzte Konzept der Gleichheit aufgrund physikalischer Identität im Speicher (mit unterschiedlichen Gleichheitsfunktionen wie EQUAL und EQ in Lisp) hat in SASL keine Bedeutung. Desgleichen hat ein Ausdruck bei aufeinanderfolgenden Auswertungen stets den gleichen Wert. Wie wir später sehen werden, gestattet uns dies, durch einen entsprechenden Entwurf der Reduktionsmaschine mehrfache Auswertungen eines Ausdrucks zu vermeiden.

Eine interessante Konsequenz für den Begriff der Gleichheit soll nicht unerwähnt bleiben. Da Funktionen in SASL gleichberechtigte Objekte sind, muß auch für sie die Funktion eq erklärt werden. Zwar gilt

```
f = g ⇔ f x = g x für alle x
```

doch ist diese Bedingung im allgemeinen nicht entscheidbar. Natürlich wissen wir, daß

```
f = f WHERE f a b c = a * (b + c)
```

den Wert TRUE haben sollte. Doch entsprechend der referential transparency dürfen wir einen Teilausdruck durch einen wertgleichen ersetzen, ohne daß sich der Wert des Gesamtausdrucks ändert. Im Falle von

```
f = g WHERE f a b c = a * (b + c)
            g a b c = a * b + a * c
```

ist jedoch die Gleichheit der Funktionen f und g nicht mehr so offenkundig und zumindest durch den SASL-Interpreter nicht mehr feststellbar. Die einzige konsistente Lösung für dieses Problem besteht darin, grundsätzlich festzulegen:

$$\text{function } f \;\&\; \text{function } g \Longrightarrow \text{eq } f\ g = \bot$$

1.3 Funktionen

1.3.1 Funktionsdefinitionen

In SASL gibt es zwei Methoden, Werten einen Namen zuzuordnen; beide stehen im Zusammenhang mit Funktionen. Zum einen kann der Benutzer neue Funktionen definieren und mit einem Funktionssymbol versehen, zum anderen besitzen (nicht-nullstellige) Funktionen formale Parameter, auf die innerhalb der Funktionsdefinition Bezug genommen werden kann. Funktionen können sowohl global als auch lokal definiert werden; die Form der Definition stimmt dabei überein, nur wird im Fall der globalen Definition das Schlüsselwort **def** vorangestellt. Lokale Definitionen werden dagegen - wie in 1.2.1 beschrieben - in WHERE-Ausdrücke eingebettet.

Im einfachsten Fall hat eine Funktionsdefinition die Form

<Name> <Parameterliste> = <Ausdruck>,

wobei <Name> das Funktionssymbol darstellt und die Parameterliste leer sein kann (Konstantendefinition). Der <Ausdruck> wird gelegentlich definierender Ausdruck oder Körper der Definition genannt. Ein Beispiel verdeutlicht dies:

```
def fak n = n = 0 → 1 ; n * fak (n-1)        (1)
```

Der dreistellige Infixoperator ".→.;." ist dabei als IF-THEN-ELSE zu lesen. Da in rekursiven Definitionen häufig Fallunterscheidungen vorkommen, kann man diese in SASL übersichtlicher anordnen, indem einzelne der formalen Parameter durch Konstanten ersetzt werden. So kann man die Fakultätsfunktion völlig gleichwertig zu (1) auch durch

```
def fak 0 = 1
    fak n = n * fak (n-1)                    (2)
```

Sind die Fälle nicht disjunkt (wie auch in (2)), wird der erste Fall in der Reihenfolge der Aufschreibung ausgewertet, dessen Parameterliste mit den aktuellen Parametern verträglich ist.

Außer Konstanten können auch strukturierte Parameter verwendet werden, die nur mit einem gleichartig strukturierten Listenargument gematcht werden können. Auf diese Weise läßt sich etwa die vordefinierte Listenoperation hd sehr knapp definieren:

```
def hd (a:x) = a                                   (3)
```

Wie man sieht, bringt die Verwendung strukturierter Parameter zwei Vorteile mit sich: sie gestattet die übersichtliche Darstellung von komplexen Bedingungen an die Parameter ohne expliziten Einsatz des IF-THEN-ELSE-Operators (im Beispiel (3): hd ist nur auf nicht-leeren Listen definiert), zudem können die Komponenten des strukturierten Parameters mit Namen versehen werden (in (3) a und x), auf die im Funktionskörper Bezug genommen werden kann.

Eine Fortführung dieses Konzepts stellen Namelist-Definitionen dar, in denen mehrere in einer Listenstruktur enthaltene Parameter simultan definiert werden. So ist z.B.

```
def (a : x) = [1,2,3,4]
```

gleichbedeutend mit

```
def a = 1
def x = [2,3,4]
```

Ist die zu definierende Listenstruktur nicht mit dem definierenden Ausdruck verträglich, so erhalten ***alle*** darin vorkommenden Namen den Wert ⊥.

Während die Definition (1) den Namen fak global mit der Fakultätsfunktion verbindet, ist der Gültigkeitsbereich von in WHERE-Ausdrücken eingeführten Funktionssymbolen auf den WHERE-Ausdruck beschränkt (vgl. 1.2.1).

1.3.2 Funktionale - Curried Functions

Wenn die Form der Funktionsdefinitionen auch suggeriert, daß Funktionen eine beliebige Stelligkeit besitzen können, sind tatsächlich alle Funktionen einstellig. Die zweistellige Funktion plus ist z.B. aufzufassen als ein Funktional, das einer Zahl n die einstellige Funktion zuordnet, die zu ihrem Argument n addiert. Mit Hilfe der oben eingeführten WHERE-Ausdrücke können wir

dies verdeutlichen, indem wir eine Definition von plus angeben, die diese Interpretation hervorhebt:

```
def plus n = f WHERE f m = n+m
```

Die Interpretation mehrstelliger Funktionen als einstellige Funktionale geht auf eine von H. B. Curry in /Curry, Feys58/ weiterverfolgte Idee von Schönfinkel /Schönfinkel24/ zurück; man spricht daher in diesen Fällen auch von curried functions. Umgekehrt ist es sinnvoll, als n-stellig definierte Funktionen auf m < n Argumente anzuwenden. Das Ergebnis ist in diesem Fall eine (n-m)-stellige Funktion, die man als Spezialisierung der ursprünglichen Funktion auffassen kann. So läßt sich etwa die Nachfolgerfunktion durch

```
def suc = plus 1 WHERE plus m n = m+n
```

als Spezialfall der Addition auszeichnen.

Funktionen können jedoch nicht nur Werte, sondern auch Argumente von Funktionen sein. So befindet sich unter den vordefinierten SASL-Funktionen ein Funktional map, das eine Funktion auf die Komponenten einer Liste anwendet und die Liste der Funktionswerte zurückliefert. Die Definition von map kann folgendermaßen aussehen:

```
def map f []      = []
    map f (a:x) = f a : map f x
```

1.3.3 Lazy Evaluation - Nicht-strikte Funktionen

Die Auswertungsstrategien, die bei herkömmlichen Programmiersprachen zugrundegelegt werden, führen in der Regel dazu, daß die Berechnung eines Funktionswerts nicht terminiert, wenn die Auswertung eines der Argumente nicht terminiert. Üblicherweise werden nämlich in einer ersten Phase alle Argumente ausgewertet und dann erst die ermittelten Werte in den Funktionskörper eingesetzt und dieser ausgewertet.

Im Falle von SASL werden Ausdrücke jedoch erst evaluiert, wenn ihr Wert tatsächlich benötigt wird (Lazy Evaluation oder Demand-driven Evaluation). Für die Behandlung von Funktionsanwendungen bedeutet dies, daß nur genau die Argumente ausgewertet werden,

die zur Berechnung des Funktionswertes auch benötigt werden. Dies entspricht der Auswertungsstrategie "Call by name", von der bekannt ist, daß sie (im Gegensatz zu "Call by value") immer eine Lösung liefert, wenn eine Lösung überhaupt möglich ist. Dies folgt aus einem Satz des λ-Kalküls; SASL aber besteht aus dem λ-Kalkül plus weiterer Definitionen. Beispielsweise hat der Ausdruck

```
f 1 (fak -1) WHERE f x y = x
                   fak 0 = 1
                   fak n = n * fak (n-1)
```

den Wert 1, obwohl die Berechnung von (fak -1) nicht abbricht. Da der Wert von f aber nur vom ersten Argument abhängt, wird die Auswertung von (fak -1) auch gar nicht angestoßen.

Eine entscheidende Rolle spielt diese Auswertungsstrategie bei der Behandlung von Funktionen, die unendliche Listenstrukturen erzeugen. Man kann solche Funktionen als nicht-terminierende Erzeugungsprozesse interpretieren. Im nächsten Abschnitt werden wir sehen, wie die SASL-Auswertungsstrategie eine elegante Verwendung solcher Strukturen in terminierenden Berechnungen zuläßt.

Da viele Programmiertechniken auf den vorteilhaften Gebrauch nicht-terminierender Berechnungen (meist in Gestalt unendlicher Listen) beruhen, wird es bei jeder Implementierung auf eine effiziente Realisierung dieser Auswertungsstrategie ankommen.

1.4 Listen

Wie auch in Lisp stellen in SASL Listenstrukturen beliebiger Schachtelungstiefe die einzige Datenstruktur dar. Bereits im Abschnitt 1.1 wurden die primitiven Listenoperationen ":", hd und tl vorgestellt. Durch Verwendung von rekursiven Listendefinitionen ist es jedoch nicht nur möglich, endliche Listen zu erzeugen, sondern auch endliche Beschreibungen unendlicher Strukturen anzugeben.

1.4.1 Unendliche Listenstrukturen

Betrachten wir als Motivation ein Standardbeispiel für eine Definition einer unendlichen Liste:

```
def nats = from 1 WHERE from n = n : from (n+1)
```

Man übersieht schnell, daß nats die Liste der natürlichen Zahlen ist. Es ist jedoch zunächst einmal fraglich, ob diese Definition sinnvoll ist, da die Berechnung von from 1 offensichtlich nicht abbricht. Auch hier hilft jedoch die Strategie der Demand-driven Evaluation weiter. Nehmen wir einmal an, wir wollten das erste Element von nats isolieren. Wäre nats endlich, so würde hd nats den Kopf der Liste liefern. Tatsächlich ist dies aber auch dann der Fall, wenn nats eine unendliche Liste ist. hd wertet sein Argument im Einklang mit der Philosophie der Lazy Evaluation nur so weit aus, wie es zur Bestimmung des Ergebnisses erforderlich ist. Da hd durch

```
def hd (a:x) = a
```

definiert werden kann (vgl. 1.2.1), reicht es aus, wenn das Argument auf die Form a:x reduziert werden kann. Nach Definition ist aber

```
hd nats = hd (from 1) = hd (1 : from 2) = 1
```

Der entscheidende Schritt ist der letzte, in dem die Auswertung von from 2 wiederum unterbleibt, weil sie für die Bestimmung des Wertes des Ausdrucks irrelevant ist.

Dieses Beispiel zeigt deutlich, unter welchen Voraussetzungen die Auswertung eines Gesamtausdrucks terminieren kann, obwohl er als Teilausdrücke unendliche Listenstrukturen enthält. Identifiziert man einen Ausdruck, dessen Auswertung nicht abbricht, mit ⊥, so erhält man ausgehend von from 1 eine Folge von endlichen Approximationen

```
⊥
1:⊥
1:2:⊥
1:2:3:⊥
...
```

dessen Limes from 1 selbst ist. Greift eine Funktion nun auf endlich viele Komponenten (in unserem Beispiel nur auf die erste) von from 1 zurück, so führt das Prinzip der Demand-driven Evaluation dazu, daß das kleinste Anfangsstück, das diese Komponenten enthält, explizit erzeugt wird. Der Rest der Liste bleibt als unausgewerteter Ausdruck (d.h. als endliche Beschreibung) stehen.

In der Praxis hat die Verwendung unendlicher Listenstrukturen den Vorteil, daß der Programmierer sich keine Gedanken darüber machen muß, welches Anfangsstück tatsächlich benötigt werden wird. So lassen sich in SASL-Programmen die Bereiche der Listenkonstruktion und des -zugriffs funktional trennen, wodurch die Modularität im allgemeinen sehr gefördert wird.

1.4.2 Schwach-terminierende Listen

Die Interpretation von ":" als listenerzeugender Operator impliziert, daß das zweite Argument eine Liste sein muß. Wenn man diese zusätzliche Bedingung jedoch fallen läßt, gelangt man zu einem allgemeinen Paarbildungsoperator, dessen ***beide*** Argumente beliebigen Typs sein können. Es ist dann beispielsweise möglich, ein Zahlentripel (1,2,3) statt als dreielementige Liste [1,2,3] auch als 1:2:3 darzustellen. Solche Listenstrukturen unterscheiden sich von den bisher behandelten dadurch, daß die Aussage

$$x = a{:}y \implies y = [\,] \text{ oder } y = b{:}z \qquad (5)$$

auf sie nicht mehr zutrifft. Weil diese Listen insbesondere nicht auf [ ] enden und somit in gewissem Sinne nicht wohlgeformt sind, nennt man sie schwach-terminierende Listen.

Während die verallgemeinerte Verwendung von ":" generell zulässig ist, differiert die Definition der Listenzugriffsfunktionen von einer SASL-Version zur anderen. So ist etwa in der ursprünglichen Sprachdefinition /Turner83, S.31/ tl auch auf schwach-terminierenden Listen stets definiert, denn

```
def tl (a:x) = x
```

Der ARC-SASL-Standard /Richards84/ erfordert demgegenüber, daß der Listenrest selbst wieder eine Liste ist:

```
def tl (a:x) = list x → x ; ⊥
```

Diese Definition führt zu einer Normierung der schwach-terminierenden Listen, so daß statt (5) jetzt stets gilt:

$$x = a{:}y \implies y = \bot \text{ oder } y = [\,] \text{ oder } y = b{:}z \qquad (6)$$

Eine solche Normierung reduziert die Zahl der zu betrachtenden Fälle beim Beweis von Aussagen über Listenausdrücke; so liegt auch die Vermutung nahe, daß die Einführung der verschärften Definition von tl im Zusammenhang mit dem Projekt zur Verifikation von SASL-Programmen steht, da frühere Versionen von ARC-SASL /Scheevel84, S.14/ noch die ursprüngliche Definition enthielten. Da uns die erstere Version die natürlichere erschien, haben wir uns in unserer Implementierung für sie entschieden.

1.4.3 ZF-Ausdrücke

Listen können nicht nur - wie oben dargestellt - mittels der primitiven Listenoperationen beschrieben werden. Um der gewohnten mathematischen Notation möglichst nahe zu kommen, wurde in SASL eine weitere spezielle Klasse von Ausdrücken eingeführt, deren Werte Listen sind. Es handelt sich dabei um die sogenannten ZF-Ausdrücke, die aus der Zermelo-Frankel-Mengennotation abgeleitet sind.

Ihre allgemeine Form ist:

$$[E;\ G_1;\ldots;G_n] \qquad (7)$$

wobei E ein beliebiger Ausdruck ist und
$G_i = V_i \leftarrow E_i$, V_i Variable,
E_i listenwertiger Ausdruck (Generator)
oder
$G_i = E_i$, E_i boolescher Ausdruck (Filter)

Dabei dürfen in E alle V_i vorkommen, in den E_i jedoch nur V_j mit $j<i$.

Während es sich aus Gründen der Effizienz empfiehlt, Filter so weit links wie möglich in den ZF-Ausdruck einzubauen, bleibt die Semantik des Ausdrucks von der Reihenfolge der Generatoren und Filter unberührt (vorausgesetzt, die Einschränkung für die V_i bleibt erfüllt). Wir können daher zur Vereinfachung der Darstellung o.B.d.A. voraussetzen, daß alle Generatoren zu Beginn des

Ausdrucks aufeinanderfolgen und die Filter am Ende des Ausdrucks zusammengefaßt sind.

Die Semantik eines ZF-Ausdrucks

$$ZFE = [E; G_1; ...; G_k; ...; G_n] \qquad (8)$$

ist wie folgt erklärt:

o.B.d.A. seien

$G_1, ..., G_k$ Generatoren und
$G_{k+1}, ..., G_n$ Filter.

Sei weiter für 1<=m<=k

$$\underline{m} = [V_1, ..., V_m]$$

das Tupel der ersten m in den Generatoren vorkommenden Variablen. Das geordnete kartesische Produkt einer listenerzeugenden Funktion mit einer Liste von Listen sei definiert als:

```
def
ocp f []      = []
ocp f (a:x)   = append (join (f a) a) (ocp f x)
                WHERE join [] L     = []
                      join (b:y) L = (b:L) : join y L
```

Dann ist

```
ZFE = map f (filter p_k+1 (...(filter p_n L)...)
      WHERE f k       = E
            p_1 []    = E_1
            p_2 1     = E_2
            ...
            p_k k-1   = E_k
            p_k+1 k   = E_k+1
            ...
            p_n k     = E_n
            L = foldr ocp [[]] [p_k,...,p_1]
```

Man erkennt, daß der Wert des ZF-Ausdrucks auch mit den "normalen" Listenoperationen beschrieben werden kann (map, filter und foldr sind vordefinierte SASL-Funktionen). Ebenso deutlich wird

aber, daß die ZF-Ausdrücke an Übersichtlichkeit der ZF-freien Form überlegen sind. In unserer SASL-Implementierung nutzen wir die Umwandelbarkeit jedoch aus, indem wir ZF-Ausdrücke bereits vor der Auswertung auf ZF-freie Ausdrücke zurückführen (vgl. 2.8).

Das kartesische Produkt zweier Listen kann z.B. als ZF-Ausdruck elegant und klar so definiert werden:

```
def cp x y = [[a,b]; a ← x; b ← y]
```

Interessiert man sich nur für diejenigen Paare, deren erstes Element kleiner ist als das zweite, so kann man dies durch einen zusätzlichen Filter zum Ausdruck bringen:

```
def cp' x y = [[a,b]; a ← x; b ← y; a<b]
```

Mit dieser Definition erhält man z.B.

```
cp' [1,2,2] [3,2,1] = [[1,3],[1,2],[2,3],[2,3]]
```

Trotz der engen syntaktischen Verwandtschaft zur üblichen Mengennotation ist zu beachten, daß ZF-Ausdrücke ***keine*** Mengen beschreiben, sondern Listen. Wie das Beispiel bereits zeigt, kann der Wert eines ZF-Ausdrucks durchaus Doubletten enthalten, außerdem sind die Komponenten durch ihre Reihenfolge in der Liste geordnet.

Wenn auch mengentheoretische Aussagen über ZF-Ausdrücke daher nicht getroffen werden können, so sind diese doch überaus nützlich, wo Listen durch iterative Bearbeitung anderer Listen entstehen, da sie dem Benutzer die explizite Programmierung der Rekursion ersparen. Zudem gestatten ZF-Ausdrücke häufig, die Funktionen auf der gleichen Abstraktionsebene zu definieren wie die formale Spezifikation. Dies erleichtert die Verifikation solcher Programme ganz wesentlich.

1.5 Der Sprachumfang unseres SASL-Interpreters

1.5.1 Die Datentypen

Von den in der Sprachdefinition vorgesehenen Datentypen sind in unserem System vorhanden:

- ganze Zahlen,
- boolesche Werte,
- Character,
- Funktionen,
- Listen und
- der Wert ⊥

Auf Gleitkommazahlen wurde bei der ersten Implementierung auf einem Apple II aus praktischen Erwägungen (sie fehlten im benutzten Lisp-Dialekt völlig) verzichtet, zumal ihr Fehlen in einer funktionalen Programmiersprache wohl leicht zu verschmerzen ist. Es ist aber selbstverständlich ohne großen Aufwand möglich, auch Gleitkommazahlen entweder als eigenen Datentyp oder in Mischung mit Ganzzahlen zu implementieren, sofern die Implementierungssprache dies zuläßt. So ist bei unserer Reimplementation in Common Lisp auf der Symbolics-Lispmaschine die Mischung ohne zusätzlichen Aufwand möglich.

1.5.2 Das SASL-Prelude

Für die Gesamtheit der vordefinierten Funktionen hat sich der Begriff Prelude eingebürgert. Dabei ist es unerheblich, ob die im Prelude enthaltenen Funktionen im Maschinencode des benutzten Rechners oder in SASL selbst geschrieben sind; die Abgrenzung zwischen beiden Gruppen ist letztlich auch nur eine Frage des Aufwands. Vom Konzept her stellt das Prelude eine Folge von globalen Definitionen dar, die bei Beginn einer Sitzung automatisch geladen wird.

Das Prelude unseres Systems ist im Anhang wiedergegeben, es enthält außer den SASL-Definitionen auch kurze Beschreibungen der Funktionen.

Gegenüber dem ARC-SASL-Prelude fehlen einige selten gebrauchte Funktionen, auf die wir aus Platzgründen verzichtet haben, da sie bei Bedarf ebensogut in SASL definiert werden können.

1.5.3 Namelist-Definitionen

Unser System erlaubt keine globalen Namelist-Definitionen. Im Gegensatz zu der lokalen Verwendung solcher Definitionen in WHERE-Ausdrücken (die unser System unterstützt) erscheint die globale Benutzung in aller Regel überflüssig und eher unhandlich, so daß der Aufwand für ihre Realisierung nicht gerechtfertigt erscheint.

2. Die Abstraktion

Wie wir bereits im vorigen Kapitel gesehen haben, stellen einige Aspekte in der Definition von SASL (unendliche Listen, nicht-strikte Funktionen, mehrzeilige Funktionsdefinitionen etc.) ungewöhnliche Anforderungen an den Interpreter. Es ist daher nicht verwunderlich, daß praktisch alle Implementierungen in ihrer Auswertungsstrategie dem Vorschlag folgen, den Turner in seinem klassischen Aufsatz /Turner79a/ veröffentlicht hat (zunächst Umwandlung eines SASL-Programmes mittels Abstraktion in einen Kombinatorausdruck, anschließend Reduktion dieses Kombinatorausdrucks zum gewünschten Ergebnis). Auch wir bauen bei der Implementierung unseres Systems auf dieser Basis auf, daher wird in diesem Kapitel auf die implementationsunabhängigen Regeln der Abstraktion eingegangen, bevor in Kapitel 3 dargestellt wird, wie der erzeugte Kombinatorgraph dann von der Reduktionsmaschine zum gewünschten Ergebnis reduziert wird.

2.1 Die Äquivalenz von SASL-Ausdrücken und Kombinatorausdrücken

Drei Zielsetzungen für die Auswertungsstrategie lassen sich aus Kapitel 1 ableiten:

1. Korrekte Realisierung des λ-Kalküls, d.h. statische Bindung
2. Endliche interne Repräsentationen für unendliche Listen
3. Unterstützung nicht-strikter Funktionen durch Demand-driven Evaluation

Gerade Punkt 1 schließt die Verwendung eines herkömmlichen Lisp-Interpreters mit geänderter Syntax aus, da diese - mit Ausnahme der neuesten Dialekte wie Common Lisp /Steele84/, SCHEME /Steele, Sussman78/ oder NIL /White79/ - bevorzugt mit dynamischer Bindung ausgestattet werden. Damit erzielt man in der Regel eine effizientere Behandlung von Funktionsaufrufen, weil der Overhead für das Bilden und Auflösen der lokalen Variablenbindungen geringer ist, handelt sich jedoch das sogenannte CLOSURE- oder FUNARG-Problem ein. Freie Variablen im Funktionskörper werden u.U. mit dem falschen Wert versorgt, wenn ihr Name in der dynamischen Aufrufstruktur des Programms mehrfach vorkommt. Dies hat zur Folge, daß die Semantik von Lisp-Programmen im Gegensatz zum reinen λ-Kalkül nicht invariant gegenüber Variablenumbenennungen

ist. Als Ausweg aus diesem Dilemma sah man ursprünglich nur die direkte Substitution von aktuellen Parametern in den Funktionskörper (wie auch von McCarthy in der ersten Spezifikation vorgesehen). Der damit verbundene erhebliche Mehraufwand ließ diese Alternative jedoch unattraktiv erscheinen.

Ein völlig anderer Ansatz wird in /Turner79a/ verfolgt, wo die vollständige Eliminierung von formalen Parametern (Abstraktion genannt) propagiert wird.

2.2 Grundlagen der Variablenabstraktion

Fast jeder SASL-Ausdruck enthält gebundene Variable. Dazu gehören formale Parameter wie n in

```
fak n  =  n=0 ⟶ 1 ; n*fak(n-1)
```

ebenso wie lokale Variable, z.B. x in:

```
x*(x+1)  where  x = 3
```

Viele Implementationsprobleme, nicht nur bei funktionalen, sondern auch bei imperativen Sprachen, rühren vom Problem der Festlegung des Geltungsbereiches einer gebundenen Variable her, insbesondere wenn nur schwer erkennbar ist, ob die Geltungsbereiche zweier gleichnamiger Variablen sich überlappen. Die in /Turner79a/ vorgeschlagene Methode verfolgt hierbei ein radikal anderes Konzept als das üblicherweise benutzte Modell, das auf der Verwendung von Umgebungen basiert. Turner benutzt ein Resultat der Logik aus /Schönfinkel24/, in dem gezeigt wird, daß die in der Logik und üblichen Mathematik benutzten Variablen nicht unbedingt nötig sind. Es ist nämlich möglich, mit Hilfe einer begrenzten Anzahl neuer Konstanten (Kombinatoren genannt) jede Formel systematisch in eine Notation zu übersetzen, in der keine gebundenen Variablen mehr vorkommen.

Da in SASL die in Mathematik und Logik übliche Notation weitestgehend übernommen wurde, verwenden wir dieses Resultat für eine völlig neuartige Implementierungsmethode für SASL. Für Sprachen, die Zuweisungen und Nebeneffekte zulassen (dazu gehören auch die meisten Lisp-Dialekte!) ist eine Verwendung der Abstraktion jedoch ausgeschlossen, da die dort benutzten dynamischen Variablen völlig andere Eigenschaften haben als die in der Mathematik und

auch in SASL üblichen statischen Variablen.

Eine systematische Übersetzung eines SASL-Ausdrucks in eine variablenfreie Notation kann als eine Art Compilierung gesehen werden, die als Ergebnis-Code einen Kombinatorausdruck liefert. Obwohl dieser Kombinatorausdruck für einen menschlichen Betrachter kaum noch verständlich ist, kann er recht effektiv von einer geeigneten Maschine "ausgeführt" werden. Dazu wird der Kombinatorausdruck gemäß einem überschaubaren Satz von Regeln zu einem Ergebnis reduziert. Eine derartige Reduktionsmaschine wird in Kapitel 3 ausführlich beschrieben. In diesem Kapitel interessieren wir uns stattdessen für den Algorithmus zur Beseitigung gebundener Variablen aus SASL-Ausdrücken und für die dazu verwendeten Kombinatoren.

2.3 Das Entfernen von Variablen aus Ausdrücken

Beginnen wir mit einer einfachen SASL-Definition, der Nachfolgerfunktion auf Zahlen (vgl. 1.3.2):

def suc x = plus 1 x (9)

Unser Ziel ist es, die gebundene Variable x aus dieser Definition zu entfernen, so daß wir etwas der Form

def suc = ...

erhalten. In (9) können wir einfach x auf beiden Seiten entfernen und erhalten

def suc = plus 1 (10)

Dies ist eine brauchbare Lösung, da das <u>Prinzip der Extensionalität</u> (f = g $\Leftrightarrow$ f x = g x für alle x) erfüllt ist. In unserem Beispiel bedeutet dieses Prinzip, daß man an beliebigen Stellen "suc" und "plus 1" austauschen kann, ohne dadurch den Sinn des betroffenen Ausdrucks zu ändern.

Der Schritt von (9) nach (10) war nun allerdings besonders einfach, da x auf beiden Seiten nur jeweils einmal als rechtestes Symbol vorkam. Wir suchen aber ein allgemein anwendbares Verfahren für eine beliebige Definition

def f x = T (11)

wo T ein aus Konstanten und der Variablen x mittels diverser Operatoren aufgebauter SASL-Ausdruck ist.

Zunächst wandeln wir T in einen gleichbedeutenden Ausdruck E um, indem wir alle (Infix- und Postfix-) Operatoren unter Beachtung ihrer Bindungsstärken und Assoziativitätsregeln durch die entsprechenden <u>curried functions</u> (vgl. 1.3.2) ersetzen.

```
Bsp.:        T  =  x=3 → 0 ; x*f(x+1)-1
       ⟹     E  =  if (eq x 3) 0
                      (minus (times x (f (plus x 1))) 1)
```

In E ist dann die Funktionsanwendung die einzige Operation. Ein so geformter Ausdruck heißt <u>Kombination</u>. Wird aus

def f x = E (12)

x entfernt, so schreiben wir

def f = [x] E (13)

wo "[x] E" (gesprochen: "x abstrahiert aus E") das Ergebnis des noch zu definierenden Abstraktionsverfahrens ist.

Eine wichtige Forderung an [x] E ist, daß alle Informationen über die Vorkommen von x in E noch enthalten sind, so daß

([x] E) x = E (14)

erfüllt ist. (14) heißt das <u>Gesetz der Abstraktion</u> und fordert, daß die Abstraktion die Umkehrung der Applikation (also der Funktionsanwendung) ist.

2.4 Variablenabstraktion mit Kombinatoren

2.4.1 Die Grundkombinatoren I, K und S

Wir führen zunächst drei Kombinatoren I, K und S ein, die durch folgende Gleichungen definiert sind:

$$
\begin{aligned}
I\ x &= x \\
K\ y\ x &= y \\
S\ f\ g\ x &= f\ x\ (g\ x)
\end{aligned}
$$

Nun können wir die Abstrahierung so beschreiben:

$$
\begin{aligned}
[x]\ x &= I \\
[x]\ y &= K\ y \qquad\qquad (x \neq y) \\
[x]\ (E_1\ E_2) &= S\ ([x]\ E_1)\ ([x]\ E_2)
\end{aligned}
$$

Dies entspricht in Notation des λ-Kalküls

$$
\begin{aligned}
\lambda x.x &= I \\
\lambda x.y &= K\ y \qquad\qquad (x \neq y) \\
\lambda x.(E_1\ E_2) &= S\ (\lambda x.E_1)\ (\lambda x.E_2)
\end{aligned}
$$

Man sieht leicht, daß die so definierte Abstrahierung das Gesetz der Abstraktion erfüllt:

$$
\begin{aligned}
([x]\ x)\ x &= I\ x = x \\
([x]\ y)\ x &= K\ y\ x = y \\
([x]\ (E_1\ E_2))\ x &= S\ ([x]\ E_1)\ ([x]\ E_2)\ x = \\
&= ([x]\ E_1)\ x\ (([x]\ E_2)\ x) = E_1\ E_2
\end{aligned}
$$

Die drei bisher definierten Abstraktionsregeln sind notwendig und hinreichend, um eine Variable aus jedem beliebigen Term zu entfernen. Daher nennen wir die drei Kombinatoren I, K und S Grundkombinatoren. Leider werden aber die bei ausschließlicher Verwendung von I, K und S erzeugten Kombinatorausdrücke schnell sehr lang. So erhalten wir z.B. für die Nachfolgerfunktion

$$
\begin{aligned}
\textbf{def}\ suc &= [x]\ (plus\ 1\ x) \\
&\Rightarrow S\ ([x]\ (plus\ 1))\ ([x]\ x) \\
&\Rightarrow S\ (S\ ([x]\ plus)\ ([x]\ 1))\ I \\
&\Rightarrow S\ (S\ (K\ plus)\ (K\ 1))\ I \qquad\qquad (15)
\end{aligned}
$$

was zwar korrekt ist, aber nicht so handlich wie "plus 1".

2.4.2 Optimierung durch weitere Regeln und Kombinatoren

Als erste Verbesserung bieten sich zwei verkürzende Regeln an (E_1, E_2 beliebige Kombinationen):

$$S\ (K\ E_1)\ (K\ E_2) \Rightarrow K\ (E_1\ E_2) \quad (16)$$
$$S\ (K\ E_1)\ I \Rightarrow E_1 \quad (17)$$

Auch hier gilt das Prinzip der Extensionalität:

$$S\ (K\ E_1)\ (K\ E_2)\ x = (K\ E_1)\ x\ ((K\ E_2)\ x) =$$
$$= E_1\ E_2 = K\ (E_1\ E_2)\ x$$
$$S\ (K\ E_1)\ I\ x = (K\ E_1)\ x\ (I\ x) = E_1\ x$$

Mit Hilfe von (16) und (17) können wir nun auch die Abstraktion der Nachfolgerfunktion aus (15) auf das erwartete Maß kürzen:

```
def suc  =  S (S (K plus) (K 1)) I
         ⇒  S (K (plus 1)) I
         ⇒  plus 1
```

Eine weitere Verbesserung erfolgt durch die Einführung von zwei weiteren Kombinatoren B und C mit

```
B f g x  = f (g x)       (entspricht Komposition)
C f x y  = f y x         (Argumenttausch)
```

Die zugehörigen Verkürzungsregeln sind

$$S\ (K\ E_1)\ E_2 \Rightarrow B\ E_1\ E_2 \quad (18)$$
$$S\ E_1\ (K\ E_2) \Rightarrow C\ E_1\ E_2 \quad (19)$$

Diese Regeln werden nur dann angewandt, wenn (16) und (17) *nicht* anwendbar sind.

Die durch (16)-(19) beschriebenen Optimierungen lassen sich als integraler Bestandteil in den Abstraktionsalgorithmus einbauen, so daß die umständliche Form wie in (14) gar nicht erst entsteht.

Die Abstraktion von Funktionen mit mehr als einem Argument wie in

```
def f x y = E
```

ist durch zweimaliges (bzw. bei n Argumenten durch n-maliges) sukzessives Anwenden der o.a. Abstraktionsregeln lösbar:

def f = [x]([y] E)

"[y] E" entspricht hier der einstelligen Funktion "(f x)", wird daraus x abstrahiert, so bleibt f übrig.

Leider steigt die Länge des Ergebnisausdruckes enorm mit steigender Anzahl der herausabstrahierten Variablen. Dies kann leicht an einem Beispiel gezeigt werden: Angenommen, wir wollen aus einem Term der Form "Q R" die Variablen x_1 bis x_n herausabstrahieren. Als Kurznotation schreiben wir Q' statt $[x_1]$ Q, Q'' statt $[x_2]([x_1]$ Q) etc., analog R', R'' etc. Falls jedes x_k sowohl in Q als auch in R jeweils mindestens einmal vorkommt, erhalten wir durch sukzessives Abstrahieren folgende Resultate:

```
Q R
S Q' R'
S (B S Q'') R''
S (B S (B (B S) Q''')) R'''
S (B S (B (B S) (B (B (B S)) Q'''')))) R''''
...
```

Man sieht sofort, daß die Länge quadratisch anwächst. Der Grund für dieses untragbare Verhalten ist darin zu suchen, daß spätere Abstraktionen die von früheren Abstraktionen eingebauten Kombinatoren mitverarbeiten müssen. Zur Vermeidung dieser Schwierigkeit führen wir wie in /Turner79b/ einen neuen Kombinator S' ein, definiert durch:

S' kt f g x = kt (f x) (g x)

S' entspricht in seiner Funktion dem Kombinator S; es "umgeht" lediglich einen Anfangsterm kt. Die neue Abstraktionsregel lautet dann:

S (B kt f) g $\Longrightarrow$ S' kt f g

kt ist hierbei ein beliebiger Term, der ***nur aus Konstanten*** besteht. Mit Verwendung von S' steigt dann die Länge des Kombinatorausdrucks aus unserem obigen Beispiel nur noch linear mit wachsender Zahl der abstrahierten Variablen:

```
Q R
S Q' R'
S' S Q'' R''
S' (S' S) Q''' R'''
S' (S' (S' S)) Q'''' R''''
...
```

Analog zu den Verbesserungen B und C für S führen wir schließlich noch B' und C' ein:

```
B' kt f g x  =  kt f (g x)
C' kt f x y  =  kt (f y) x
```

Die Abstraktionsregeln für B' und C' sind:

```
B (kt f) g   ⟹  B' kt f g
C (B kt f) x ⟹  C' kt f x
```

Auch hier darf kt nur aus Konstanten bestehen.

2.5 Abstraktion von Listen

Listen bilden als einzige Datenstruktur einen wesentlichen Pfeiler der Programmiersprache SASL. Ihre häufige Verwendung rechtfertigt die Behandlung der primitiven Listenoperationen auf dem Niveau der Kombinatoren. Dem Paarkonstruktor ":" entspricht dabei der zweistellige P-Kombinator mit exakt der gleichen Bedeutung. Unser Abstraktionsalgorithmus wandelt dann "a : b" in den Kombinatorausdruck "P a b" um. "[a,b,c]" - eine Kurzschreibweise für "a : (b : (c : []))" - wird so zu "P a (P b (P c []))".

Im Einklang mit der Strategie der Lazy Evaluation werden die Argumente von P erst dann weiter reduziert, wenn auf die Komponenten der Liste zugegriffen wird; dadurch wird die Auswertung irrelevanter Listenteile vermieden und die Verwendung unendlicher Listenstrukturen erst ermöglicht. Kombinatoren, deren Reduktionen ein Vorkommen des P-Kombinators berühren, sind beispielsweise die Funktionen für den Listenzugriff, die als Kombinatoren hd und tl nach

```
hd (P x y) = x
tl (P x y) = y
```

reduziert werden.

Listen können in SASL auch als formale Parameter (strukturierte Parameter, vgl. 1.3.1) auftauchen, z.B. in

def f (a:b:c) = plus a b

Um auch derartige Ausdrücke abstrahieren zu können (dies ist mit den bisher eingeführten Regeln noch nicht möglich), führen wir den Kombinator U (für uncurry) ein:

U f (P x y) = f x y

Die neue Abstraktionsregel lautet dann:

[P x y] E ⟹ U ([x] ([y] E))

Auch hier wird das Gesetz der Abstraktion erfüllt:

([P x y] E) (P x y) = U ([x] ([y] E)) (P x y) =
= ([x] ([y] E)) x y = ([y] E) y = E

Vor der Reduktion von U wird also (ähnlich wie bei hd und tl) das zweite Argument reduziert. Nur wenn das Endergebnis dieser Reduktion die Form "(P x y)" hat, ist die U-Reduktion anwendbar. Damit unterscheiden sich die Listenkombinatoren von den Grundkombinatoren insoweit, als sie einen Teil ihrer Argumente reduzieren, bevor ihre eigene spezifische Regel angewandt wird.

Dies ist im Falle des U-Kombinators nicht ganz unproblematisch. Nehmen wir einmal an, der Kombinatorausdruck

U (K (K E_1)) E_2

sei zu reduzieren. Das Ergebnis sollte E_1 lauten, auch wenn sich bei der Reduktion von E_2 Probleme ergeben. Diese können sowohl darin bestehen, daß E_2 nicht auf die Form P x y reduziert wird, möglicherweise terminiert die Reduktion von E_2 aber auch nicht. Letztere Situation tritt tatsächlich auf, wenn bei der Abstraktion von WHERE-Ausdrücken mit mehreren gegenseitig rekursiven Funktionen der Fixpunktoperator auf eine Liste von Funktionen angewandt wird. Bei der Reduktion eines solchen Kombinatorausdrucks liegt genau dann der o.a. Fall vor, wenn der Rekursionsanfang einer der Funktionen erreicht wird. An dieser Stelle wird

bei herkömmlicher Implementierung der U-Reduktion die rekursive Definition ein weiteres Mal expandiert und die Reduktion bricht nicht ab. Eigentümlicherweise weist Turner in /Turner79a, S. 42/ besonders auf die Notwendigkeit einer strikten U-Reduktion hin, ohne die Probleme anzusprechen, die im Zusammenhang damit auftreten.

Wir haben das Problem gelöst, indem wir die Reduktion des Listenarguments von U verschieben, bis feststeht, daß dessen Komponenten tatsächlich benötigt werden. Dazu ersetzen wir den U-Kombinator durch einen Kombinator U' mit gleicher Abstraktionsregel, der aber abweichend von der oben angegebenen nach

U' f x = f (hd x) (tl x)

reduziert wird. Nach wie vor gilt für die Abstraktion

[P x y] E ⟹ U' ([x] ([y] E))

und das Gesetz der Abstraktion:

```
([P x y] E) (P x y)  =  U' ([x] ([y] E)) (P x y)  =
                     =  ([x] ([y] E)) (hd (P x y)) (tl (P x y)) =
                     =  ([x] ([y] E)) x y  =  ([y] E) y  =  E
```

x wird erst bei Zugriff auf hd x bzw. tl x reduziert. Sollte sich bei dieser Gelegenheit herausstellen, daß x keine Liste ist, wird der Fehler statt bei der U-Reduktion nun von hd oder tl bemerkt und gemeldet. Die Striktheit von U wird also für den Fall, daß f <u>nicht</u> konstant ist, nicht aufgehoben. Eine nachträgliche Bestätigung für unsere Vorgehensweise fand sich in /Boutel84/, wo neben dem U-Kombinator auch ein U'-Kombinator mit der veränderten Reduktionsregel angegeben wird, allerdings ebenfalls ohne Hinweis auf den Verwendungszweck.

2.6 Die Abstraktion von WHERE-Ausdrücken

2.6.1 Alternativen bei der Behandlungsweise

Prinzipiell können WHERE-Ausdrücke auf zwei Arten in Kombinatorausdrücke umgeformt werden: Zum einen kann man zuerst den WHERE-Ausdruck durch eine Reihe globaler Definitionen ersetzen und diese dann einzeln wie gezeigt abstrahieren. Man erspart sich damit eine gesonderte Behandlung von WHERE-Ausdrücken bei der Abstraktion; hat aber Mehraufwand durch das automatische Umwandeln in Listen von Definitionen. Insbesondere müssen in größerem Umfang Umbenennungen durchgeführt werden, da nur noch globale Bezeichner (die natürlich unterschiedlich sein müssen) erlaubt sind. Letztendlich sind globale Definitionen auch aufzufassen als ein unsichtbares WHERE, denn wenn der Benutzer z.B.

```
map (plus 1) [1, 2, 3]
```

eingibt, so bedeutet das nichts anderes als

```
map (plus 1) [1, 2, 3]  where  map  = ...
                               plus = ...
                               u.s.w.
```

Zusätzliche Schwierigkeiten bei einer Umwandlung eines WHERE-Ausdrucks in eine Reihe globaler Definitionen ergeben sich dadurch, daß innerhalb des WHERE Variable vorkommen dürfen, die außerhalb definiert sind, wie z.B. in

```
def f x y  =  g x  where  g n = y-n
```

Um hier g global definieren zu können, müßte es in eine zweistellige Funktion umgewandelt werden; die notwendigen Umformungen sind ziemlich aufwendig.

Die andere Alternative besteht in einer gesonderten Behandlung der lokalen Definitionen im WHERE-Ausdruck.

Zur Umgehung der Umbenennungsprobleme und aus Effizienzgründen entscheiden wir uns gegen eine Transformation in Listen WHERE-freier globaler Definitionen, stattdessen beschreiben wir im folgenden Abschnitt spezielle Abstraktionsregeln für WHERE-Ausdrücke.

2.6.2 Die direkte Abstraktion von WHERE-Ausdrücken

Als typischer Fall für gebundene Variable wurde schon weiter oben der Fall einer lokalen Variablendefinition in einem WHERE-Ausdruck wie dem folgenden genannt:

$$E_1 \text{ where } x = E_2 \qquad (20)$$

Dies wird abstrahiert zu

$$([x]\ E_1)\ E_2$$

Falls eine lokale Funktion definiert wird, abstrahiert man zunächst die formalen Parameter aus der Funktionsdefinition und behandelt das Ergebnis dann wie in (20).

$$\begin{aligned} & E_1 \text{ where } f\ x = E_2 \\ \Rightarrow\ & E_1 \text{ where } f = [x]\ E_2 \\ \Rightarrow\ & ([f]\ E_1)([x]\ E_2) \end{aligned}$$

Die in (20) beschriebene Lösung funktioniert jedoch nicht, wenn die lokalen Definitionen rekursiv sind. In diesem Fall benötigen wir noch einen weiteren Kombinator, den Fixpunktkombinator Y, zur Abstraktion:

$$\begin{aligned} & E \text{ where } x = \ldots x \ldots \qquad (21) \\ \Rightarrow\ & ([x]\ E)(Y\ ([x]\ (\ldots x \ldots))) \end{aligned}$$

Dabei gilt, daß "Y f" für jede Funktion f der Fixpunkt von f ist:

$$Y\ f \ = \ f\ (Y\ f)$$

Wie schon im nichtrekursiven Fall werden auch bei rekursiven Funktionsdefinitionen zunächst die formalen Parameter wegabstrahiert und dann die für (21) angegebene Methode angewandt.

$$\begin{aligned} & E \text{ where } f\ x = \ldots f \ldots \\ \Rightarrow\ & E \text{ where } f = [x](\ldots f \ldots) \\ \Rightarrow\ & ([f]\ E)(Y\ ([f]\ ([x]\ (\ldots f \ldots)))) \end{aligned}$$

Meist besitzt ein WHERE-Ausdruck nicht nur eine, sondern mehrere lokale Variablen- und Funktionsdefinitionen. Die einzelnen Definitionen können dabei sowohl in sich selbst als auch wechselseitig rekursiv sein.

```
Bsp.: E  where  f x = ...g...
                g y = ...f...
```

Derartige wechselseitige Rekursion wird wie folgt behandelt: Man abstrahiert die einzelnen Funktionszeilen wie bisher, faßt dann alle lokalen Definitionen zu einer Definitionsliste zusammen und abstrahiert diese wie für (21) gezeigt.

```
  ⇒ E  where  f = [x](...g...)
              g = [y](...f...)
  ⇒ E  where  (P f (P g [])) =
                        (P [x](...g...) (P [y] (...f...) []))
```

was (21) entspricht mit (P f (P g [])) anstelle von x.

Da eine Untersuchung jeder einzelnen Definition im Definitionspaket eines WHERE-Ausdruckes auf direkte oder indirekte (wechselseitige) Rekursion zu aufwendig ist, gehen wir vom "schlimmsten" Fall aus, daß nämlich alle Definitionen miteinander in rekursiver Beziehung stehen. Daher ist die allgemeine Vorgehensweise dieselbe wie im obigen Beispiel für gegenseitige Rekursion (o.B.d.A. seien zuerst die lokalen Funktionen und dann die lokalen Variablen definiert):

$$
\begin{array}{lll}
 & E \;\; \text{where} & f_1 \; x_{1,1} \; .. \; x_{1,k1} = E_1 \\
 & & \ldots \\
 & & f_n \; x_{n,1} \; .. \; x_{n,kn} = E_n \\
 & & y_1 = E_{n+1} \\
 & & \ldots \\
 & & y_m = E_{n+m} \\
\Rightarrow & E \;\; \text{where} & f_1 = [x_{1,k1}] \; (\; .. \; [x_{1,1}] \; E_1) ..) \\
 & & \ldots \\
 & & f_n = [x_{n,kn}] \; (\; .. \; [x_{n,1}] \; E_n) ..) \\
 & & y_1 = E_{n+1} \\
 & & \ldots \\
 & & y_m = E_{n+m}
\end{array}
$$

$\Longrightarrow$ E where (P f_1 (P ... (P f_n (P y_1 (P ... (P y_m [])....) =
(P [$x_{1,k1}$] (.. [$x_{1,1}$] E_1)..)
(P ... (P [$x_{n,kn}$] (.. [$x_{n,1}$] E_n)..)
(P E_{n+1} (P ... (P E_{n+m} [])....)

$\Longrightarrow$ ([(P f_1 (P ....)..)] E)
(Y ([(P f_1 (P ....)..)] (P [$x_{1,k1}$] ...)))

2.7 Fallunterscheidung durch Definition und Pattern-Matching

Während zum bisher beschriebenen Teil der Abstraktion bei Turner zumindest Hinweise über die Vorgehensweise zu finden sind, mußte die in diesem Abschnitt beschriebene Behandlung mehrzeiliger Funktionsdefinitionen und des Pattern-Matching neu entwickelt werden.

Zunächst zwei Definitionen, die wir zur Behandlung des Pattern-Matching benötigen:

Def. 1:

Sei M eine beliebige Menge, dann ist LM(M) die kleinste Menge mit:

a) $x \in M \Longrightarrow x \in LM(M)$
b) $x,y \in LM(M) \Longrightarrow (P\ x\ y) \in LM(M)$

LM(M) heißt Menge der Listenmuster über M
Ein Listenmuster Lm aus LM(M) heißt echt, wenn gilt: $Lm \notin M$

Def. 2:

VarBez := Menge der Variablenbezeichner
KonstBez := Menge der Konstantenbezeichner, d.i.
Zahlen, Character, TRUE, FALSE und []
LMV := LM(VarBez + { [] })
LMK := LM(KonstBez + VarBez)

Pattern-Matching kann auf zwei Weisen auftreten:

1. In einer zu einem WHERE-Ausdruck lokalen Definition wird einem echten Listenmuster Lm aus LMV ein Ausdruck zugewiesen.

   Bsp.: E_1 where Lm = E_2

2. In einer Funktionsdefinition tritt als formaler Parameter ein Listenmuster aus LMK auf.

   Bsp.: E_1 where f Lm = E_2

Betrachten wir zunächst den ersten Fall:
Sicherheitshalber muß Lm daraufhin untersucht werden, ob es tatsächlich in LMV enthalten ist, denn nur dann ist die Bildung von "[Lm] E_1" definiert. Findet der Test in Lm eine Konstante außer "[ ]", so wird ein Fehler in der Compilierung gemeldet.

Der zweite Fall hängt eng mit den impliziten Fallunterscheidungen bei auf mehrere Zeilen verteilten Funktionsdefinitionen zusammen und wird daher mit diesen zusammen behandelt:

In SASL ist es gestattet, eine Fallunterscheidung in einer Funktionsdefinition über Ersetzung von formalen Parametern durch Konstante oder Listenmuster zu definieren. So ist z.B. die folgende Definition zwar ungewöhnlich, aber erlaubt:

```
f FALSE  = [ ]
f 0 x    = x
f a a    = a - 1
f y TRUE = 2 * y
f [ ]    = suc
f (x:y)  = x y
f 9 [1,2,x,[3,y],x] = 0
f [[3],[y,a,y],a] a = 1
```

Hier sind alle schwierigen Fälle vereint:

- verschiedene Argumentzahl
- Fallunterscheidung durch
  a) Konstanten
  b) komplexe Listenmuster mit Konstanten an beliebigen Stellen
  c) Verwendung gleichnamiger formaler Parameter

Unsere Vorgehensweise wird wie folgt sein:
Zuerst abstrahieren wir die einzelnen Zeilen der Funktionsdefinition für sich und erhalten dabei für jede Zeile bestimmte Bedingungen über ihre Anwendbarkeit. Dann fassen wir alle Zeilen zu einem großen geschachtelten IF-Ausdruck zusammen, in dem die erste Zeile, deren Bedingungen erfüllt sind, benutzt wird.

Beginnen wir mit der Behandlung der einzelnen Zeilen der Funktionsdefinition. Eine solche Zeile hat die Form:

$$f\ fp_1\ ..\ fp_n = E$$

Bei der Abstraktion der formalen Parameter fp_1 bis fp_n muß wie üblich der letzte Parameter (fp_n) zuerst und der erste (fp_1) zuletzt aus dem definierenden Ausdruck E herausabstrahiert werden. Dazu ersetzen wir in den formalen Parametern eventuell vorkommende Konstanten durch neue, bisher nicht verwendete Variablenbezeichner und abstrahieren die so "entschärften" formalen Parameter dann nacheinander aus dem definierenden Ausdruck heraus. Dies funktioniert, da ja die neuen Bezeichner nicht vorkommen und daher lediglich ein K-Kombinator vorgesetzt wird; dieser "schluckt" später bei der Reduktion die entsprechende Konstante. So erhalten wir nach dem sukzessiven Herausabstrahieren der formalen Parameter den Kombinatorausdruck für den definierenden Ausdruck der gerade behandelten Zeile.

Nun müssen noch die Bedingungen erzeugt werden, unter denen die gerade bearbeitete Zeile "zuständig" ist. Wir geben dazu jedem formalen Parameter fp_k einen neuen, bisher nicht verwendeten Name NEU_k. Da jede Variable oder Konstante auch ein Listenmuster aus LMK ist, wird im folgenden nur die Behandlung von Listenmustern als formalen Parametern beschrieben. Der jedem Listenmuster gegebene neue Name ersetzt dieses als formaler Parameter. Von diesem Namen ausgehend kann man mit hd und tl auf jeden Teil des Listenmusters zugreifen; dieser Ausdruck zum Erreichen eines Teils des Listenmusters heißt <u>Zugriffsname</u>.

Beispiel:

f a 3 (1:x:(3:[ ]):y) = E

Setze als neue Namen NEU_1 - NEU_3 ein, dann sind folgende Zugriffsnamen für die einzelnen Parameter bzw. Parameterteile definiert:

a	⟹	Zugriffsname "NEU_1"
3	⟹	Zugriffsname "NEU_2"
1:x:(3:[ ]):y	⟹	Zugriffsname "NEU_3"
1	⟹	Zugriffsname "hd(NEU_3)"
x:(3:[ ]):y	⟹	Zugriffsname "tl(NEU_3)"
x	⟹	Zugriffsname "hd(tl(NEU_3))"
(3:[ ]):y	⟹	Zugriffsname "tl(tl(NEU_3))"
(3:[ ])	⟹	Zugriffsname "hd(tl(tl(NEU_3)))"
3	⟹	Zugriffsname "hd(hd(tl(tl(NEU_3))))"
[ ]	⟹	Zugriffsname "tl(hd(tl(tl(NEU_3))))"
y	⟹	Zugriffsname "tl(tl(tl(NEU_3)))"

Die durch ein Listenmuster Lm mit Zugriffsnamen Zn erzwungenen Bedingungen BED(Lm,Zn) werden wie folgt erzeugt:

Lm ∈ KonstBez ⟹ BED(Lm,Zn) = (Zn = Lm)

Lm ∈ VarBez ⟹ BED(Lm,Zn) = { }

Lm = (P Lm_1 Lm_2) ⟹ BED(Lm,Zn) =
(NELIST Zn) & BED(Lm_1,hd(Zn)) & BED(Lm_2,tl(Zn))

NELIST ist dabei das Prädikat "Not Empty LIST".

Auf diese Weise werden alle durch die Struktur der formalen Parameter und durch die darin enthaltenen Konstanten gestellten Bedingungen gesammelt. Um auch diejenigen Bedingungen zu erhalten, die durch gleichnamige formale Parameter entstehen, werden alle Variablennamen aus den Parameter-Listenmustern gesammelt und miteinander verglichen. Bei Namensgleichheit wird als zusätzliche Bedingung die Gleichheit der entsprechenden Zugriffsnamen gefordert.

Die gesammelten Bedingungen zur Anwendbarkeit einer Zeile werden nun in einer großen Konjunktion zusammengefaßt (sie enthält im übrigen nur die ***neuen*** Namen NEU_1 bis NEU_n, über die die Zugriffsnamen definiert wurden). Diese neuen Namen der formalen Parameter werden nun aus der gebildeten Konjunktion herausabstrahiert; danach ist auch der den Bedingungen entsprechende Kombinatorausdruck variablenfrei.

Nun hat man zu jeder Zeile der Funktionsdefinition

a) den definierenden Kombinatorausdruck (variablenfrei)

b) die Bedingungen, die zu seiner Anwendung erfüllt sein müssen (ebenfalls variablenfrei).

Sollte in einer Zeile keine Bedingung herausgekommen sein, so gilt diese Zeile für alle Fälle ("catch-all case") und alle evtl. noch danach folgenden Zeilen dieser Funktionsdefinition sind als überflüssig zu streichen. Existiert keine solche Zeile ohne Bedingungen, so gibt es Fälle, in denen die definierte Funktion nicht anwendbar ist; als Service für den Benutzer wollen wir ihm in diesem Fall eine entsprechende Fehlermeldung zukommen lassen und fügen

f = error "nichtdefinierter Fall in Funktion f"

als neue letzte Zeile ohne Bedingungen hinzu.

Nachdem wir so für jede Zeile einer Funktionsdefinition

$$f\ Lm_{1,1}\ ..\ Lm_{1,n(1)} = E_1$$
$$...$$
$$f\ Lm_{m,1}\ ..\ Lm_{m,n(m)} = E_m$$

sowohl den definierenden Ausdruck als auch die dafür gültigen Bedingungen in variablenfrei abstrahierter Form erzeugt haben (wegen eventuell vorhandener überflüssiger Zeilen bzw. einer zusätzlichen abschließenden Fehlermeldung muß nicht unbedingt $m' = m$ gelten):

$$(BED_1, DEF_1)$$
$$...$$
$$(BED_{m'}, DEF_{m'})$$

können wir die gesamte Funktion wie folgt darstellen:

```
f   x1 .. xn(1)           = IF (BED1 x1 .. xn(1))
                               (DEF1 x1 .. xn(1))
                               (F2 x1 .. xn(1))
F2 x1 .. xn(2)            = IF (BED2 x1 .. xn(2))
                               (DEF2 x1 .. xn(2))
                               (F3 x1 .. xn(2))
...
Fm'-1 x1 .. xn(m'-1)      = IF (BEDm'-1 x1 .. xn(m'-1))
                               (DEFm'-1 x1 .. xn(m'-1))
                               (DEFm' x1 .. xn(m'-1))
```

Dabei sind die x_k ($1 <= k <= max\{n(1),..,n(m'-1)\}$) neue Variablennamen. Da diese neuen formalen Parameter *nicht* in den BED_k und DEF_k vorkommen, ist ein erneutes zeilenweises Herausabstrahieren dieser Parameter aus den IF-Ausdrücken als zu umständlich abzulehnen. Statt dessen geben wir einen Algorithmus an, der abhängig von n(k) aus BED_k, DEF_k und F_{k+1} (bzw. DEF_{k+1}) einen Kombinatorausdruck Q' erzeugt, der gleichwertig ist zu

```
Q = [xn(k)] ([xn(k)-1] (...([x1] (IF (BEDk x1 .. xn(k))
                    (DEFk x1 .. xn(k)) (Fk+1 x1 .. xn(k)))..)
```

es soll also gelten:

$$Q\ x_1\ ..\ x_{n(k)} = Q'\ x_1\ ..\ x_{n(k)}$$

Die im folgenden definierte Abbildung **IF-Abstr** liefert zur Parameterzahl n und den Kombinatorausdrücken für die Bedingungen (B), den Then-Fall (T) und den Else-Fall (E) einen derartigen Kombinatorausdruck, ohne daß die evtl. aufwendige echte Abstraktion benutzt wird.

Def. 3:

IF-Abstr(n,B,T,E) :=
n = 0 ⟶ IF B T E
n = 1 ⟶ S (S' IF B T) E
n > 1 ⟶ S' **box**(S,n-2) (**box**(IF,n) B T) E

wobei **box** definiert ist durch:
box(E,n) :=
n = 0 ⟶ E
n > 0 ⟶ (S' **box**(E,n-1))

Beispiel:

box(IF,3) = (S' (S' (S' IF)))

IF-Abstr(3,B,T,E) = S' (S' S) ((S' (S' (S' IF))) B T) E

Daß **IF-Abstr** die gewünschte Eigenschaft besitzt, zeigen die nächsten zwei Sätze:

Satz 1:

n >= 0; B, T, E beliebige Kombinatorausdrücke;
x eine beliebige Variable. Dann gilt:

IF-Abstr(n,(B x),(T x),(E x)) = (**IF-Abstr**(n+1,B,T,E)) x

Beweis durch Fallunterscheidung:

n = 0 ⟶

IF-Abstr(0,(B x),(T x),(E x)) = IF (B x) (T x) (E x)
(**IF-Abstr**(1,B,T,E)) x = S (S' IF B T) E x =
= (S' IF B T x) (E x) = IF (B x) (T x) (E x)

n = 1 ⟶

IF-Abstr(1,(B x),(T x),(E x)) = S (S' IF (B x) (T x)) (E x)
(**IF-Abstr**(2,B,T,E)) x =
= S' **box**(S,0) (**box**(IF,2) B T) E x =
= S' S ((S' (S' IF)) B T) E x =
= S (S' (S' IF) B T x) (E x) = S (S' IF (B x) (T x)) (E x)

n > 1 ⟶

IF-Abstr(k+2,B,T,E) x =
= S' **box**(S,k) (**box**(IF,k+2) B T) E x =
= **box**(S,k) (**box**(IF,k+2) B T x) (E x) =
= S' **box**(S,k-1) (S' **box**(IF,k+1) B T x) (E x) =
= S' **box**(S,k-1) (**box**(IF,k+1) (B x) (T x)) (E x) =
= **IF-Abstr**(k+1,(B x),(T x),(E x))

Satz 2:

n,B,T,E wie im vorigen Satz; dann gilt:

IF-Abstr(n,B,T,E) x_1 .. x_n =
IF (B x_1 .. x_n) (T x_1 .. x_n) (E x_1 .. x_n)

Beweis durch vollständige Induktion über n:

I.A.: n=0 ⟶ trivial

I.V.: Gelte Behauptung für n
z.z.: Behauptung gilt für n+1

$$
\begin{aligned}
&\textbf{IF-Abstr}(n+1,B,T,E)\ x_1 \,..\, x_{n+1} \overset{\text{(Satz 1)}}{=} \\
&= \textbf{IF-Abstr}(n,(B\ x_1),(T\ x_1),(E\ x_1))\ x_2 \,..\, x_{n+1} \overset{\text{(I.V.)}}{=} \\
&= \text{IF}\ ((B\ x_1)\ x_2 \,..\, x_{n+1})\ ((T\ x_1)\ x_2 \,..\, x_{n+1}) \\
&\qquad ((E\ x_1)\ x_2 \,..\, x_{n+1}) = \\
&= \text{IF}\ (B\ x_1 \,..\, x_{n+1})\ (T\ x_1 \,..\, x_{n+1})\ (E\ x_1 \,..\, x_{n+1})
\end{aligned}
$$

Der mit Hilfe von **IF-Abstr** und **box** gebildete Kombinatorausdruck ist im übrigen nicht länger als ein durch übliche Abstraktion aus dem gleichwertigen IF-Ausdruck erzeugter.

Damit kann der gesuchte Kombinatorausdruck für die Funktion f bestimmt werden zu:

$$
\begin{aligned}
&\textbf{IF-Abstr}(n(1),BED_1,DEF_1, \\
&\quad \textbf{IF-Abstr}(n(2),BED_2,DEF_2, \\
&\qquad \textbf{IF-Abstr}(... \\
&\qquad\quad .... \\
&\qquad\qquad \textbf{IF-Abstr}(n(m'-1),BED_{m'-1},DEF_{m'-1},DEF_{m'})..)
\end{aligned}
$$

2.8 Die Behandlung von ZF-Ausdrücken

Ähnlich wie beim Pattern-Matching und bei der Behandlung mehrzeiliger Funktionsdefinitionen ist in der Literatur (z.B. /Turner83/ oder /Richards84/) auch zum Thema ZF-Ausdrücke (vgl. 1.4.3) keine präzise Information zur Semantik zu finden. Statt dessen werden diese Sprachelemente fast ausschließlich durch Beispiele beschrieben. Daher waren wir bei der Implementierung in diesen Fällen auf eigene Ideen und Interpretationen angewiesen. Eine Beschreibung der Semantik eines ZF-Ausdrucks findet sich in Kapitel 1.4.3.

ZF-Ausdrücke stellen immer eine Liste dar. Die allgemeine Form für einen ZF-Ausdruck ist

$Z = [E;Q_1;..;Q_n]$

wobei E ein beliebiger Ausdruck ist und

$Q_k = V_k \leftarrow E_k$, V_k Variable,
E_k listenwertiger Ausdruck (Generator) oder
$Q_k = E_k$, E_k boolescher Ausdruck (Filter)

Dabei dürfen in E alle V_k vorkommen, in den E_k jedoch nur V_j mit $j<k$. Die Q_k heißen Qualifier.

Da ZF-Ausdrücke von dem bisher geschilderten Compilierungsverfahren nicht erfaßt werden, müssen wir für sie eine spezielle Behandlungsweise einführen. Es sind zwei Methoden denkbar:

1) Man entwickelt eine besondere Abstraktionsmethode für ZF-Ausdrücke, die deren mengenähnlichen Eigenschaften ausnutzt.

2) Man wandelt ZF-Ausdrücke vor Beginn der Abstraktion mit einem geeigneten Algorithmus in gleichbedeutende ZF-freie Ausdrücke um; diese werden dabei jedoch im allgemeinen länger und weniger übersichtlich als die ursprüngliche ZF-Form.

Da die erste Methode nur mit großem Aufwand - wenn überhaupt - realisierbar erscheint, haben wir uns bei unserer Implementierung von SASL für den zweiten Weg, also die Umwandlung in ZF-freie Ausdrücke, entschieden.

Im folgenden wird unser Umwandlungs-Algorithmus sowie ein Beweis für seine Korrektheit vorgestellt.

Def. 4:

Seien $V_{k1}, V_{k2}, .., V_{km}$ $(1<=k1<k2<..<km<=n)$ die im umzuwandelnden ZF-Ausdruck Z auftretenden Generatorvariablen. Dann definieren wir die **Tupelfunktion T**:

$T(Z) := [V_{km}, V_{km-1}, .., V_{k1}]$

T(Z) ist also die (umgedrehte) SASL-Liste aller in Z generierten Variablen.

Def. 5:

Ein ZF-Ausdruck $Z = [E;Q_1;..;Q_n]$ ist in Normalform, wenn gilt: $E = T(Z)$

Def. 6:

$NZ_n := [T(NZ_n);Q_1;..;Q_n]$

Als Erstes können wir feststellen, daß ein zu Z gleichwertiger Ausdruck durch

$Z' = \text{map f } [T(Z);Q_1;..;Q_n] \text{ where f } T(Z) = E$

gegeben ist. Wir haben damit die durch E definierte Funktion aus dem ZF-Ausdruck herausgezogen und wenden die so durch Wegabstrahieren der V_{kj} aus E erhaltene Funktion mit map (vgl. Anhang A) auf alle Wertetupel der vom ZF-Ausdruck $[T(Z);Q_1,..Q_n]$ gebildeten Liste an.

Im weiteren brauchen wir uns also nur noch ZF-Ausdrücke in Normalform

$NZ_n = [T(NZ_n);Q_1;..;Q_n]$

in gleichwertige ZF-freie Ausdrücke umzuwandeln. Ein Verfahren, welches das leistet, wollen wir nun angeben. Das Verfahren arbeitet rekursiv mit der Anzahl der Q_k:

Sei NZ_n ein ZF-Ausdruck in Normalform mit n Qualifiern, dann definieren wir **norm**(NZ_n) so:

1) n=0, d.h. der ZF-Ausdruck enthält keine Qualifier:
   norm(NZ_0) := [[ ]]

2) n>0, hier sind zwei Fälle zu unterscheiden:
   a) $Q_n = E_n$ mit E_n boolescher Ausdruck (Filter):
      norm(NZ_n) := filter f **norm**(NZ_{n-1}) where f $T(NZ_n) = E_n$

   b) $Q_n = V_{kj} \leftarrow E_n$ mit E_n listenwertiger Ausdruck (Generator):
      norm(NZ_n) := ocp f **norm**(NZ_{n-1}) where f $T(NZ_{n-1}) = E_n$

Dabei ist "filter" die im SASL-Prelude (vgl. Anhang) enthaltene Listenfilterfunktion und "ocp" (ordered cartesian product) eine wie folgt definierte Funktion auf Listen von Listen:

```
def ocp f []    = []
    ocp f (a:x) = (join (f a) a) ++ ocp f x
                  where
                  join []    e = []
                  join (a:x) e = (a:e) : join x e
```

Wir zeigen zunächst je eine Behauptung über "join" und "ocp", bevor wir den Beweis für die Gleichwertigkeit von NZ_n und **norm**(NZ_n) durchführen.

Satz 3:

[a:x ; a ← L] = join L x

Beweis:

Sei L = [l_1,..,l_n]
[a:x ; a ← L] = [(l_1:x),(l_2:x),..,(l_n:x)] =
= (l_1:x) : .. : (l_{n-1}:x) : (l_n:x) : [] =
= (l_1:x) : .. : (l_{n-1}:x) : (l_n:x) : join [] x =
= (l_1:x) : .. : (l_{n-1}:x) : join [l_n] x =
...
= join [l_1,..,l_n] x =
= join L x

Satz 4:

[a:l ; l ← LL ; a ← f l] = ocp f LL

Beweis:

a) LL = []:
[a:l ; l ← [] ; a ← f l] = [] = ocp f []

b) LL = [l_1,..,l_n]:
[a:l ; l ← LL ; a ← f l] =
= [a:l_1 ; a ← f l_1] ++ [a:l_2 ; a ← f l_2] ++ .. ++
++ [a:l_n ; a ← f l_n] =
(Satz 3)
= join (f l_1) l_1 ++ join (f l_2) l_2 ++ .. ++
++ join (f l_n) l_n =
= ocp f [l_1,..,l_n] = ocp f LL

Satz 5:

$\mathbf{norm}(NZ_n) = NZ_n$

Beweis durch Induktion über n:

I.A.: $n=0 \rightarrow NZ_0 = [\mathbf{T}(NZ_0)] = [[]] = \mathbf{norm}(NZ_0)$

I.V.: Gelte $\mathbf{norm}(NZ_n) = NZ_n$

z.z.: $\mathbf{norm}(NZ_{n+1}) = NZ_{n+1}$

Fall 1: Q_{n+1} ist Filter, also boolescher Ausdruck

$NZ_{n+1} = [\mathbf{T}(NZ_{n+1});Q_1;..;Q_n;E_{n+1}] =$

$= [\mathbf{T}(NZ_n);Q_1;..;Q_n;E_{n+1}] =$

$=$ (filter f $[\mathbf{T}(NZ_n);Q_1;..;Q_n]$ where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

$=$ (filter f NZ_n where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

(I.V.)

$=$ (filter f $\mathbf{norm}(NZ_n)$ where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

$= \mathbf{norm}(NZ_{n+1})$

Fall 2: Q_{n+1} ist Generator:

$NZ_{n+1} = [\mathbf{T}(NZ_{n+1});Q_1;..;Q_n;\ V_{kj} \leftarrow E_{n+1}] =$

$= [V_{kj} : \mathbf{T}(NZ_n);Q_1;..;Q_n;\ V_{kj} \leftarrow E_{n+1}] =$ (*)

$=$ ($[V_{kj}$: T; T $\leftarrow [\mathbf{T}(NZ_n);Q_1;..;Q_n]$; $V_{kj} \leftarrow$ f T]
where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

$=$ ($[V_{kj}$: T; T $\leftarrow NZ_n$; $V_{kj} \leftarrow$ f T] where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

(Satz 4)

$=$ (ocp f NZ_n where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

(I.V.)

$=$ (ocp f $\mathbf{norm}(NZ_n)$ where f $\mathbf{T}(NZ_n) = E_{n+1}$) $=$

$= \mathbf{norm}(NZ_{n+1})$

Da in (*) V_{kj} nur vom letzten Qualifier (der V_{kj} generiert) abhängt und $\mathbf{T}(NZ_n)$ nur von den Q_1 bis Q_n, können wir anstelle von $\mathbf{T}(NZ_n)$ eine Variable T setzen, die aus der durch Q_1 bis Q_n bestimmten Menge generiert wird; diese Menge ist genau $[\mathbf{T}(NZ_n);Q_1;..;Q_n]$.

Somit ist gezeigt, daß $\mathbf{norm}(NZ_n)$ ein zu NZ_n gleichwertiger Ausdruck ist, der ohne das äußere ZF-Konstrukt auskommt. Abschließend definieren wir noch die Funktion **ZF-Elim**, die in einem beliebigen SASL-Ausdruck S alle ZF-Ausdrücke so ersetzt, daß ein zu S gleichwertiger ZF-freier Ausdruck entsteht:

a) S = [E;Q_1;..;Q_n] ⟹
ZF-Elim(S) := map f **norm**([**T**(S);**ZF-Elim**(Q_1);..;**ZF-Elim**(Q_n)])
where f **T**(S) = **ZF-Elim**(E)

b) S = f x_1 .. x_n ⟹
ZF-Elim(S) := **ZF-Elim**(f) **ZF-Elim**(x_1) .. **ZF-Elim**(x_n)

c) S ∈ VarBez + KonstBez ⟹
ZF-Elim(S) := S

2.9 Zusammenfassung des Abstraktionsvorgangs

Die oben geschilderten Einzelverfahren werden nun zusammen zur Umwandlung beliebiger SASL-Programme in variablenfreie Kombinatorausdrücke verwandt.

1) Das SASL-Programm wird von einem Parser in eine Kombination umgewandelt, d.h. alle Operatoren werden durch ihre curried version ersetzt.

2) Mit der in 2.8 definierten Umwandlungsfunktion **ZF-Elim** werden alle ZF-Ausdrücke in gleichwertige ZF-freie Ausdrücke geändert.

3) Es wird die Funktion **GenAbstr** auf die entstandene ZF-freie Kombination angewandt.

GenAbstr prüft, ob es sich um einen WHERE-Ausdruck handelt, in diesem Fall werden die lokalen Definitionen mit den in 2.6 und 2.7 geschilderten Methoden behandelt und anschließend abstrahiert. Handelt es sich nicht um einen WHERE-Ausdruck, so wird **GenAbstr** auf Funktion und Argumente des Ausdrucks einzeln angewandt; **GenAbstr** angewandt auf Variablen- oder Konstantenbezeichner liefert diese ungeändert zurück.

2.10 Übersicht über die verwandten Abstraktionsregeln und Kombinatoren

a) Grundkombinatoren I, K, S

$$I\ x = x$$
$$K\ y\ x = y$$
$$S\ f\ g\ x = f\ x\ (g\ x)$$

$$[x]\ x = I$$
$$[x]\ y = K\ y \qquad (x \neq y)$$
$$[x]\ (E_1\ E_2) = S\ ([x]\ E_1)\ ([x]\ E_2)$$

b) zusätzliche Regeln und Kombinatoren

I) Vereinfachungsregeln

$$S\ (K\ E_1)\ (K\ E_2) \Rightarrow K\ (E_1\ E_2)$$
$$S\ (K\ E_1)\ I \Rightarrow E_1$$

II) zusätzliche Kombinatoren zur Variablenabstraktion

$B\ f\ g\ x = f\ (g\ x)$ (entspricht Komposition)

$C\ f\ x\ y = f\ y\ x$ (Argumenttausch)

$$S\ (K\ E_1)\ E_2 \Rightarrow B\ E_1\ E_2$$
$$S\ E_1\ (K\ E_2) \Rightarrow C\ E_1\ E_2$$

III) Kombinatoren zur Mehrfachabstraktion

$$S'\ kt\ f\ g\ x = kt\ (f\ x)\ (g\ x)$$
$$B'\ kt\ f\ g\ x = kt\ f\ (g\ x)$$
$$C'\ kt\ f\ x\ y = kt\ (f\ y)\ x$$

(kt beliebiger Term, der ***nur aus Konstanten*** besteht)

$$S\ (B\ kt\ f)\ g \Rightarrow S'\ kt\ f\ g$$
$$B\ (kt\ f)\ g \Rightarrow B'\ kt\ f\ g$$
$$C\ (B\ kt\ f)\ x \Rightarrow C'\ kt\ f\ x$$

c) Kombinatoren zur Listenabstraktion

```
P x y         =  x : y          (Paarungskombinator)
U f (P x y)   =  f x y
U' f x        =  f (hd x) (tl x)

[P x y] E  ⟹  U' ([x] ([y] E))
```

d) Kombinator zur direkten Abstraktion von WHERE-Ausdrücken

```
Y f  =  f (Y f)
```

3. Kombinatorgraphen und ihre Reduktion

3.1 Prinzipielle Vorgehensweise

Bevor wir jetzt eine Repräsentation und einen Reduktionsalgorithmus für die Kombinatorausdrücke angeben, fassen wir kurz zusammen, welches Ziel wir mit ihrer Verwendung verfolgen. Ein Durchlauf der Interpreter-Schleife beinhaltet das Einlesen eines SASL-Ausdrucks, seine Transformation in einen äquivalenten Kombinatorausdruck und die Reduktion der darin vorkommenden Kombinatoren. Wenn keine Reduktionsregel mehr anwendbar ist, liegt der Wert des SASL-Ausdrucks in Normalform vor (die Definition von "Normalform" präzisieren wir für Kombinatorgraphen in 3.1.3) und kann in seiner externen Repräsentation (soweit vorhanden) ausgegeben werden.

Zwei Probleme sind mit dieser Vorgehensweise eng verknüpft. Da die Auswertung von Teilausdrücken, isoliert betrachtet, nicht zu terminieren braucht, ist es a priori überhaupt nicht klar, ob und in welchen Fällen der Endzustand der Reduktion erreicht wird oder ob seine Definition nicht modifiziert werden muß. Zum anderen wurde bereits in Abschnitt 1.2.2 erwähnt, daß mehrfache Reduktionen desselben Kombinatorausdrucks möglichst zu vermeiden seien. Unter den Kombinatoren befinden sich aber auch solche, bei deren Reduktion Argumente unausgewertet kopiert werden, z.B.:

```
S f g x = f x (g x)
U' f x  = f (hd x) (tl x)
```

Ohne weitere Vorkehrungen würden die weiteren Reduktionsschritte in aller Regel die mehrfache Reduktion von x erfordern.

Beide Probleme werden durch die Darstellung von Kombinatorausdrücken durch Kombinatorgraphen und die Einhaltung einer bestimmten Reihenfolge bei der Graphreduktion gelöst.

3.1.1 Kombinatorgraphen

Für die Beschreibung der Reduktion von Kombinatorgraphen benötigen wir einige Begriffe.

Definition:

Jedem Kombinator und jeder Konstante x ordnen wir einen Knoten v_x zu.

Definition:

Jedem Kombinatorausdruck A wird eindeutig ein gerichteter **Kombinatorgraph** KG(A) mit Knotenmenge V, Kantenmenge E und ausgezeichneter Wurzel W zugeordnet. Diese Abbildung ist definiert durch:

(i) A = x Kombinator oder Konstante:
KG(A) := (V,E,W) mit $V = \langle v_x \rangle$
$E = \emptyset$
$W = v_x$

(ii) $A = A_1 \; A_2 \; \ldots \; A_n$ Applikation von Komb.,
Sei $KG(A_i) = (V_i, E_i, W_i)$:
KG(A) := (V,E,W) mit

$$V = \Sigma \, V_i + \{w_1, \ldots, w_n\}$$

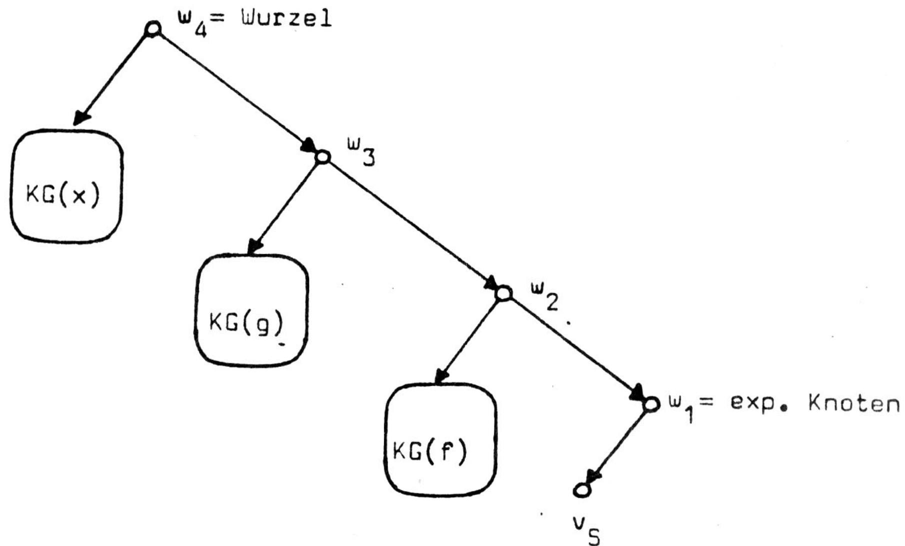
$$E = \Sigma \, E_i + \{(w_{i+1}, w_i) \mid 1 <= i <= n-1\} + \{(w_i, W_i) \mid 1 <= i <= n\}$$

$$W = w_n$$

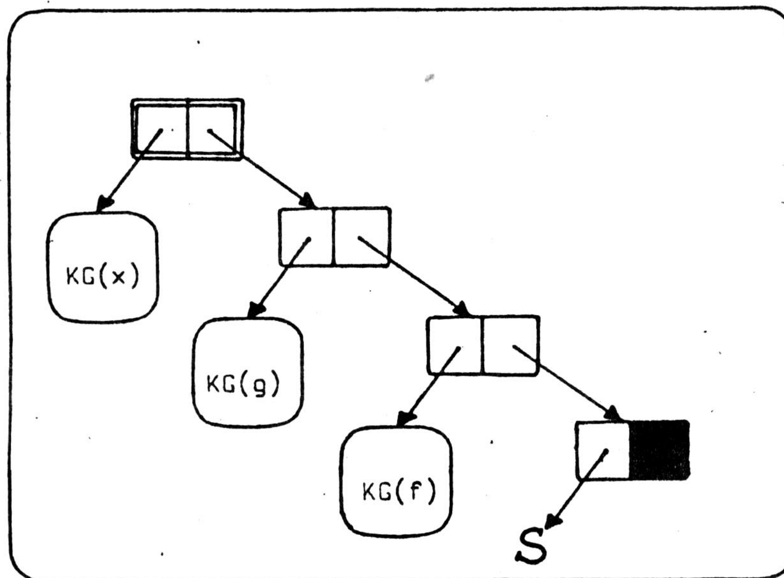
("+" steht für disjunkte Vereinigung)

Im Fall (ii) heißt w_1 **exponierter Knoten** von KG(A). $KG(A_i)$ ist der **linke Subgraph** von w_i, während der Subgraph mit Wurzel w_{i-1} **rechter Subgraph** von w_i heißt.

So entspricht dem Kombinatorausdruck A = S f g x etwa der folgende Kombinatorgraph KG(A), in dem w_4 Wurzel und w_1 exponiert ist. Der linke Subgraph von w_1 besteht im Beispiel nur aus einem Knoten:



Weil in unserem System Kombinatorgraphen durch Lisp-Listen und Kombinatoren durch Atome dargestellt werden, gehen wir bei der graphischen Veranschaulichung zur gewohnten "box-and-arrow"-Notation über. Die Verweise auf die linken bzw. rechten Subgraphen werden dabei mit dem CAR- bzw. CDR-Feld einer Listenzelle identifiziert. Wir heben zudem die Wurzel durch doppelte Umrahmung und den exponierten Knoten durch ein ausgefülltes CDR-Feld hervor. Damit erhält der obige Kombinatorgraph die Gestalt

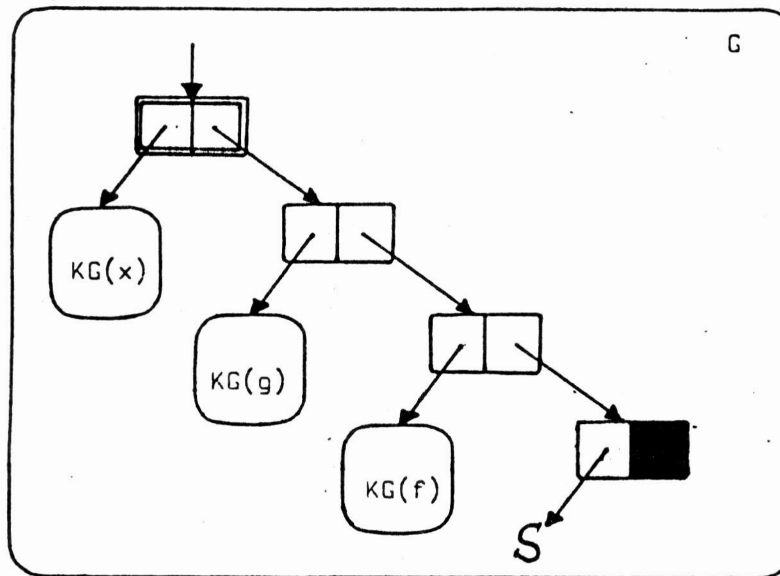


In Listennotation entspricht KG(A) der Liste (<KG(x)> <KG(g)> <KG(f)> S). Die Definitionen von Kombinatorausdrücken und -graphen gewährleisten gemeinsam, daß Kombinatorgraphen stets azyklisch sind. Wir werden später diese Bedingung abschwächen und zumindest als Reduktionsergebnisse auch zyklische Kombinatorgra-

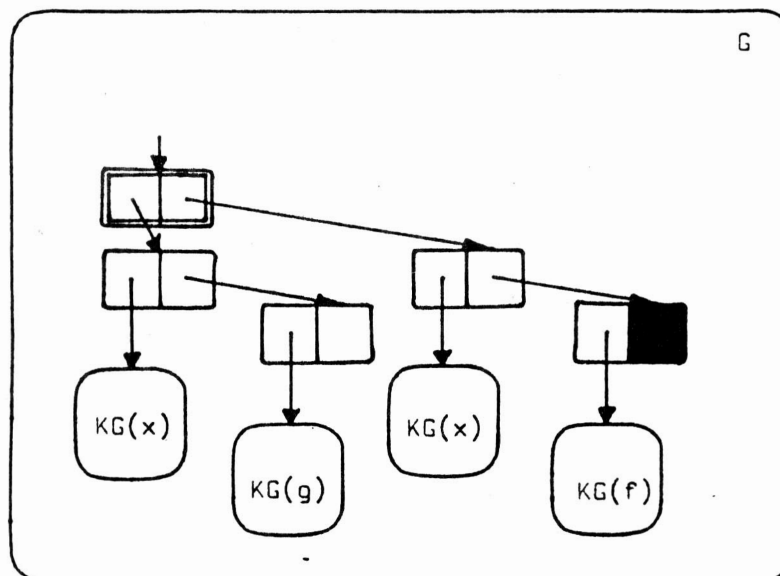
phen zulassen.

3.1.2 Anwendung einer Reduktionsregel im Kombinatorgraphen

Reduktionsregeln für Kombinatorausdrücke entsprechen bei den Kombinatorgraphen lokale Transformationen von Teilgraphen. Wann immer ein Kombinatorgraph die linke Seite einer Reduktionsregel als Teilgraphen enthält, kann dieser durch den Teilgraphen ersetzt werden, der der rechten Seite entspricht. So überführt z.B. die Anwendung der S-Reduktion den Kombinatorgraphen



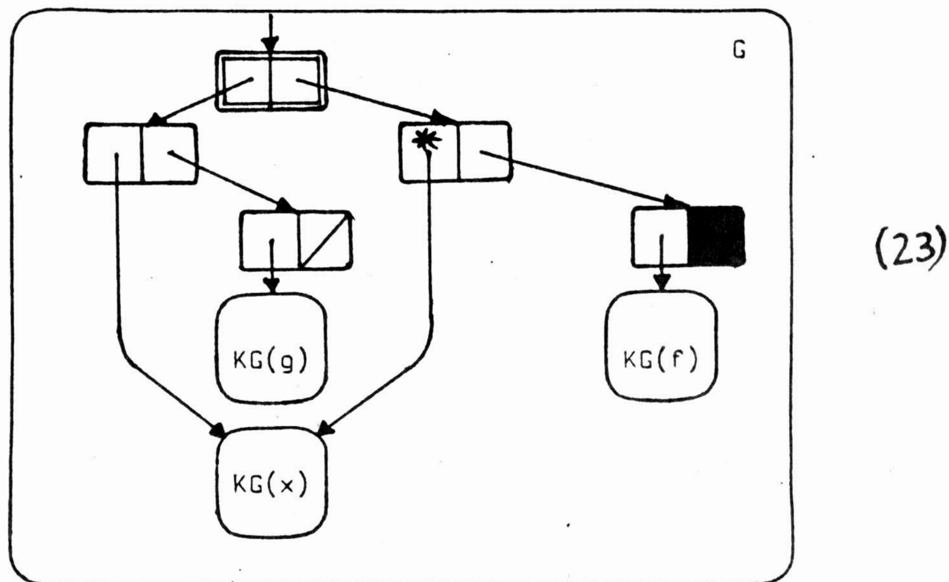
in den Graphen



(22)

Man erkennt hierbei, daß der die Reduktionsregel bestimmende Kombinator stets der linke Sohn des exponierten Knotens in dem Subgraphen ist und daß die Argumente des Kombinators die linken Subgraphen der Knoten zwischen Wurzel und exponiertem Knoten sind. Wichtig ist weiterhin, daß Verweise auf die Wurzel des Teilgraphen auch nach der Reduktion noch intakt sind, da dieser beibehalten wird.

Diese Darstellung der Transformation läßt jedoch noch nicht den enormen Vorteil erkennen, den die Graphreduktion bei der Vermeidung von Mehrfachreduktionen bietet. Da die inneren Knoten des Kombinatorgraphen nur Zeiger auf Teilgraphen enthalten, ist es gar nicht erforderlich, eine zweite Kopie von KG (x) anzulegen. Viel platzsparender ist es, beide Verweise auf dieselbe Kopie im Speicher zu richten, so daß statt (22) vielmehr

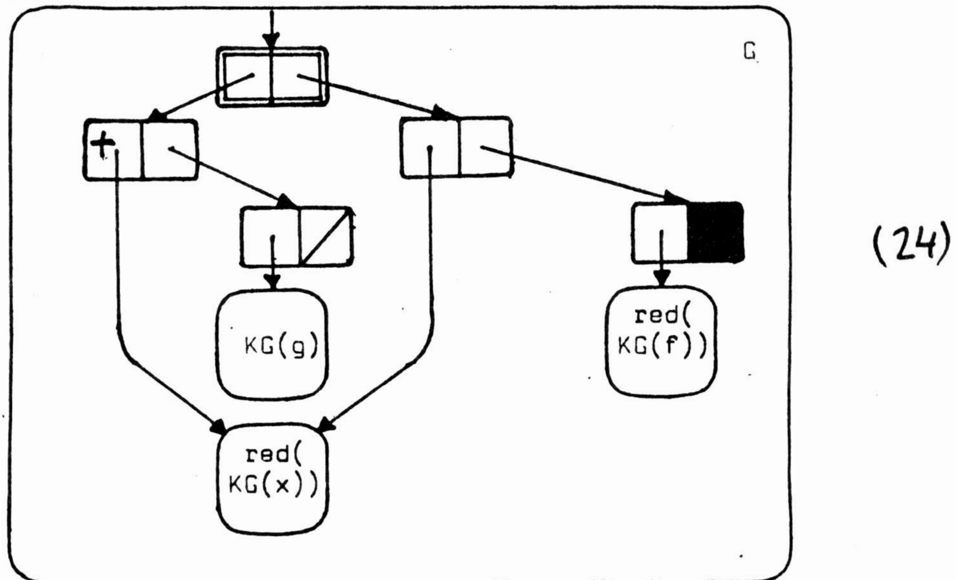


(23)

entsteht. Auf diese Weise wird erreicht, daß bei der S-Reduktion (und analog bei den meisten anderen Reduktionsregeln auch) keine komplette Kopie des unter Umständen umfangreichen Teilgraphen KG(x) hergestellt werden muß, sondern nur die Inhalte von vier Listenzellen verändert werden müssen.

Mindestens so schwer wiegt eine andere Konsequenz dieser Technik, die erst bei darauffolgenden Reduktionsschritten zutage tritt. Wir bleiben bei dem Beispiel (23) und nehmen an, nach der Reduktion von KG (f) solle nun das mit einem * bezeichnete Vorkommen von KG (x) reduziert werden. Da die Reduktionen die Zeiger auf

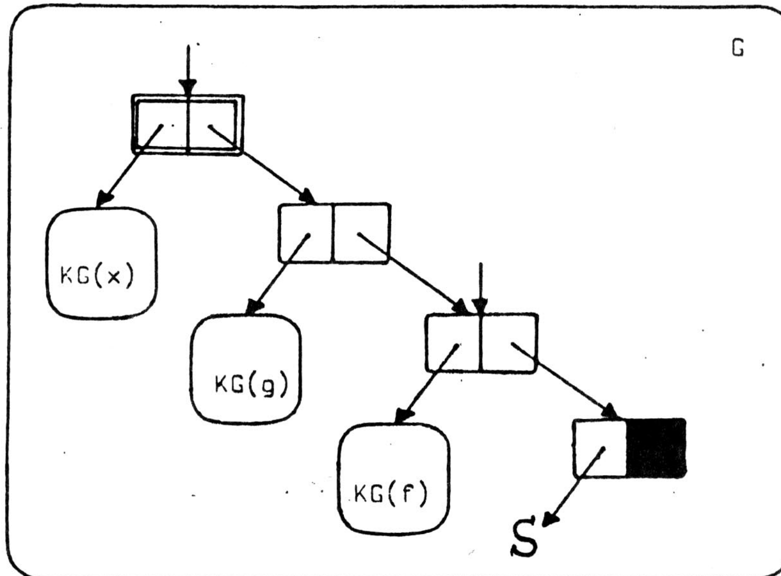
die Wurzeln von Teilgraphen nicht verändern, erhält (23) die Form (in der Graphik bezeichnet red (G) das Reduktionsergebnis des Graphen G)



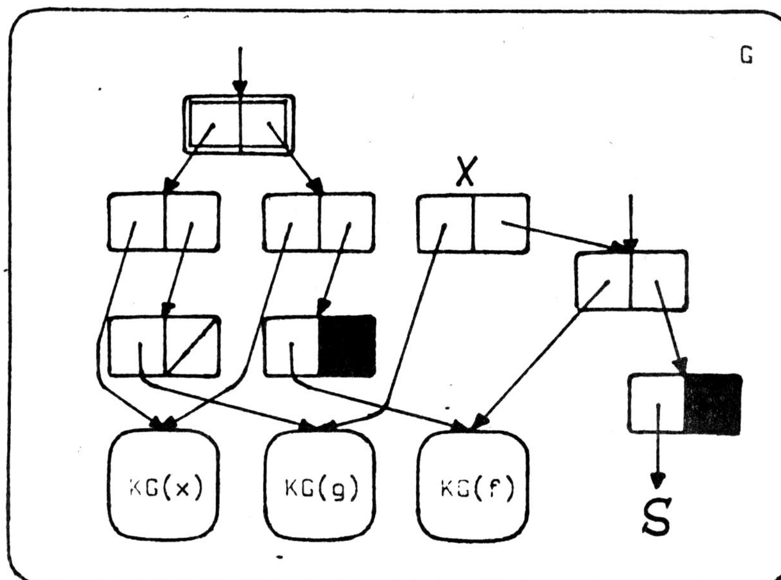
(24)

Ergibt sich später die Notwendigkeit, auch das andere Vorkommen von KG (x) (in (24) mit + gekennzeichnet) zu reduzieren, so geschieht das in einem Schritt, da sofort festgestellt wird, daß red(KG(x)) bereits in Normalform vorliegt.

Nicht immer steht ein reduzierbarer Teilgraph nur über die Wurzel in Verbindung mit dem restlichen Graphen, auch auf die übrigen inneren Knoten können Zeiger aus anderen Teilen des Graphen deuten. Aus diesem Grunde ist es wesentlich, bei der Transformation für alle Knoten außer der Wurzel ausschließlich unbenutzte Listenzellen zu verwenden, anstatt die Listenzellen des ursprünglichen Teilgraphen mit neuen Inhalten zu füllen. Wenn wir etwa die Ausgangsposition unseres Beispiels dahingehend abändern, daß ein weiterer Zeiger auf einen inneren Knoten des Teilgraphen gerichtet ist,

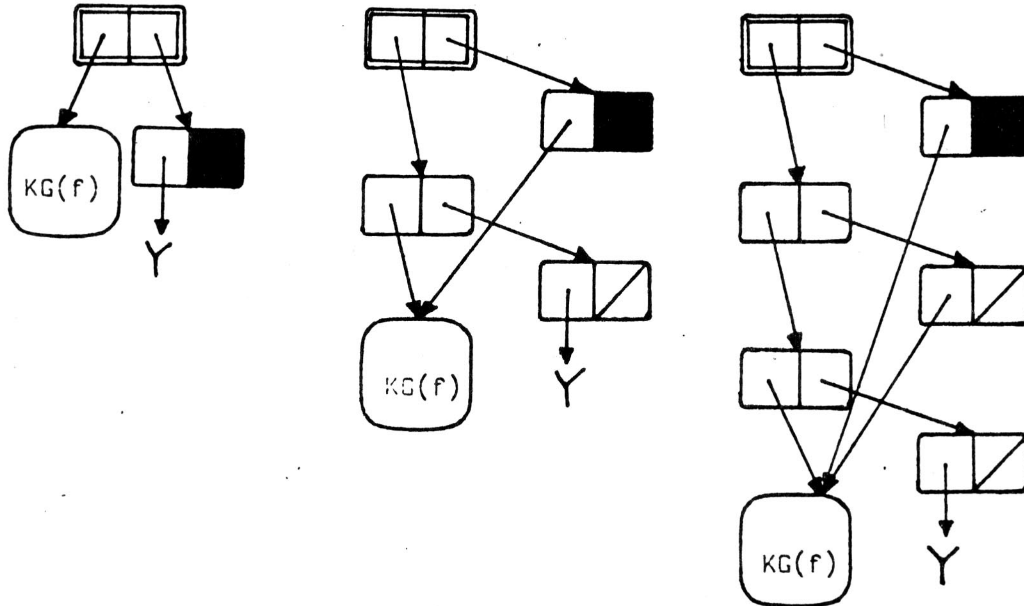


so muß nach Reduktion die Situation

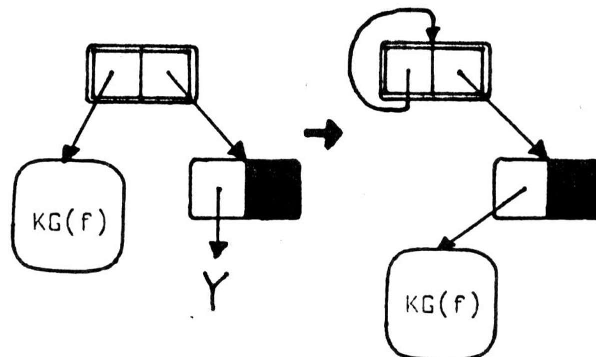


hergestellt werden. Knoten wie der mit X markierte, auf die nach der Reduktion keine Referenz mehr existiert, werden bei der nächsten Garbage Collection wieder dem Vorrat freier Listenzellen zugeführt.

Für den Fixpunktkombinator Y ergibt sich bei der Interpretation seiner Reduktionsregel als Graphtransformation sogar eine weit elegantere Beschreibung. Anstatt nach der Regel aus Abschnitt 2.6.2 schrittweise



zu transformieren, kann man in nur einem Schritt zum Limes dieser Folge von Reduktionsergebnissen übergehen, indem man Gebrauch von zyklischen Graphen macht. Dann nämlich ist



eine äquivalente (und sogar schnellere) Reduktionsregel für den Y-Kombinator.

3.1.3 Auswahl des zu reduzierenden Teilgraphen

In einem gegebenen Kombinatorgraphen sind im allgemeinen mehrere Reduktionsregeln gleichzeitig anwendbar. Sofern kein echter Parallelrechner zur Verfügung steht, mit dem simultan alle Reduktionsschritte ausgeführt werden können, muß jedoch eine Entscheidung über die Reihenfolge der Reduktionen getroffen werden.

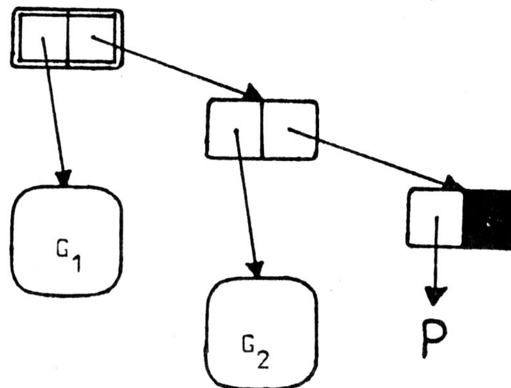
Zwar garantiert das Church-Rosser-Theorem, daß zwei terminierende Folgen von Reduktionsschritten das gleiche Ergebnis liefern, doch

hängt es von der Reihenfolge der Schritte und von der Definition des Endzustandes ab, ob dieser überhaupt erreicht wird. Natürlich sucht man eine solche Reihenfolge, die immer dann terminiert, wenn es überhaupt eine terminierende Folge gibt (der Fall, daß gar keine solche Folge existiert, ist auch möglich, er entspricht einer Endlosschleife).

Dazu präzisieren wir zunächst unsere Vorstellungen vom Endzustand der Reduktion.

Definition:

> Ein Kombinatorgraph G heißt **in Normalform**, wenn eines der beiden folgenden Kriterien gilt:
>
> (i) keine Reduktionsregel ist anwendbar oder
> (ii) G hat die Gestalt



Bemerkung: (i) deckt den Fall ab, daß der zugrundeliegende Kombinatorausdruck auf eine Konstante der Typen Zahl, Zeichen, Boolean, Funktion oder ⊥ reduziert wurde, während (ii) das Endresultat der Reduktion einer Liste ist. (i) und (ii) schließen sich jedoch nicht aus.

Nun können wir die Eignung der beiden Auswertungsreihenfolgen beurteilen, die typischerweise bei Implementierungen von Programmiersprachen Verwendung finden. Es handelt sich dabei um die Strategien der **normal-order evaluation** (im Zusammenhang mit der Semantik von Programmiersprachen hat sich der Begriff **call-by-name** eingebürgert) und **applicative-order evaluation** (auch: **call-by-value**). Im Falle von call-by-value werden alle Argumente eines Funktionsaufrufs ausgewertet, bevor die Funktion darauf angewandt wird, die Auswertung eines Ausdrucks beginnt also mit dem am

weitesten links stehenden Funktionssymbol auf der tiefsten Schachtelungsebene. Im Gegensatz dazu steht die Strategie call-by-name, bei der stets der äußerste linke Funktionsaufruf ausgewertet wird. Die Argumente werden dabei unausgewertet in den Funktionskörper substituiert. Diese Charakterisierung der beiden Auswertungsfolgen läßt sich unmittelbar auf die Situation bei den Kombinatorgraphen übertragen, wo dem Funktionsymbol der linke Sohn des exponierten Knotens und den Argumenten die linken Subgraphen der Vorgänger des exponierten Knotens entsprechen (vgl. 3.1.2). Um die Graphreduktion zu betonen, spricht man dann von **applicative-order reduction** bzw. **normal-order reduction**.

Betrachten wir nun die Situation (ii) in der Definition der Normalform. Der exponierte Knoten verweist auf den Kombinator P, G_1 und G_2 müssen aber durchaus noch nicht ihre Normalform erreicht haben. Die Entscheidung für applicative-order reduction würde bedeuten, daß G_1 und G_2 erst einmal reduziert würden, bevor festgestellt würde, daß ($\langle G_1 \rangle$ $\langle G_2 \rangle$ P) sich unverändert reproduziert. Womöglich käme es dazu aber gar nicht mehr, weil etwa bei Vorliegen einer unendlichen Liste die Reduktion von G_1 und / oder G_2 schon nicht abbrechen würde. Anders im Fall von normal-order reduction: hier wird zuerst der Kombinator am exponierten Knoten betrachtet. Die Auswertung der Argumente richtet sich dann nach der Reduktionsregel für diesen Kombinator und unterbleibt daher in unserem Beispiel völlig.

C. P. Wadsworth hat in /Wadsworth71/ den Nachweis geführt, daß die Verwendung von normal-order reduction garantiert, daß der Endzustand gemäß unserer Reduktion stets erreicht wird, sofern es überhaupt einen terminierende Folge von Reduktionsschritten gibt. Tatsächlich betrachtet Wadsworth in seiner Arbeit eine Reduktionsmaschine für den λ-Kalkül, letzten Endes handelt es sich aber auch bei SASL nur um eine syntaktisch gezuckerte Variante des λ-Kalküls, so daß das Resultat übertragbar ist. Als weitere Vereinfachung erhalten wir, daß in der Definition der Normalform eines Kombinatorgraphen das Kriterium (ii) durch (i) subsumiert wird, d.h. es wird stets solange reduziert, wie es eine Reduktionsregel für den Kombinator am exponierten Knoten gibt.

3.1.4 Die Ausgabekomponente

In einem System, das auf Lazy Evaluation beruht, muß eine Komponente den ersten Anstoß für die Auswertung geben. Bei SASL wird diese Aufgabe von der Ausgabekomponente, dem **Printer**, erfüllt. Der Printer gibt die externe Repräsentation des Wertes eines Kombinatorausdrucks auf das Ausgabemedium aus und reduziert zu diesem Zweck den zugehörigen Kombinatorgraphen. Während in einem ausgebauten SASL-System Teile der Fehlerbehandlung in den Aufgabenbereich des Printers verlagert werden (vgl. 3.2.6), ist der von solchen Details entkleidete Ausgabealgorithmus sehr einfach zu beschreiben. Wir verwenden dazu in diesem Abschnitt eine Pascal-artige Notation:

```
procedure printer (KG);

KG' := reduce(KG);
if KG' = (y x P)
  then begin
         printer(x);
         printer(y)
       end
  else if KG' ist Konstante der Typen Zahl,
          Zeichen, Boolean, ⊥ oder [ ]
       then gib externe Repräsentation von KG' aus
       else gib "function" aus (Funktionen haben
            keine externe Repräsentation)
```

Wie man sieht, wird eine Liste ausgegeben, indem der Ausgabealgorithmus sukzessive auf ihre Komponenten angewendet wird und diese dabei reduziert.

3.1.5 Ein Reduktionsalgorithmus

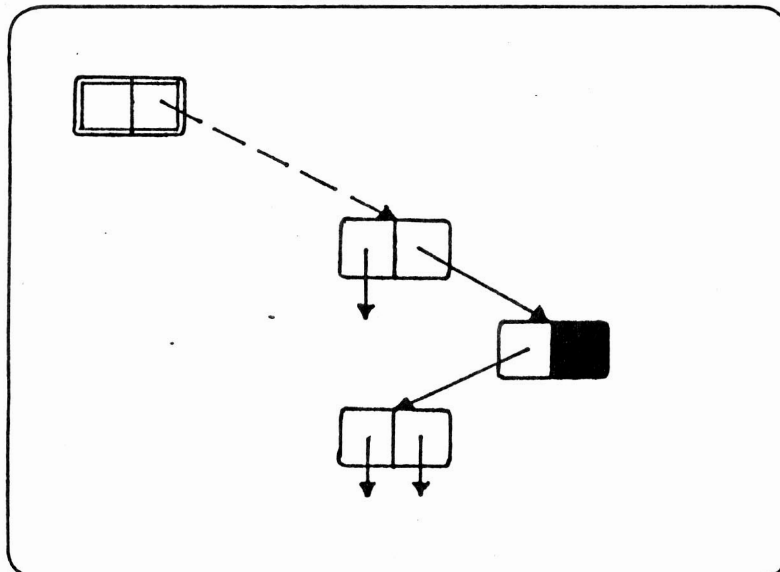
Die Überlegungen der letzten Abschnitte versetzen uns in die Lage, einen Algorithmus für die Reduktion von Kombinatorgraphen zu skizzieren. Er besteht im wesentlichen aus einer Iteration von Anwendungen der Reduktionsregeln, die durch den Kombinator am exponierten Knoten bestimmt werden:

```
procedure reduce (KG);

while not KG in Normalform do
  begin
    verfolge ausgehend von der Wurzel des KG
    die Verweise auf die rechten Subgraphen
    bis zum exponierten Knoten v_e;
    comb := linker Sohn von v_e;
    wende die Reduktionsregel zu comb an (diese
    kann bedingt sein und/oder die Reduktion
    einiger Argumente erfordern ⟹ rekursiver
    Aufruf von reduce);
  end;
return KG
end
```

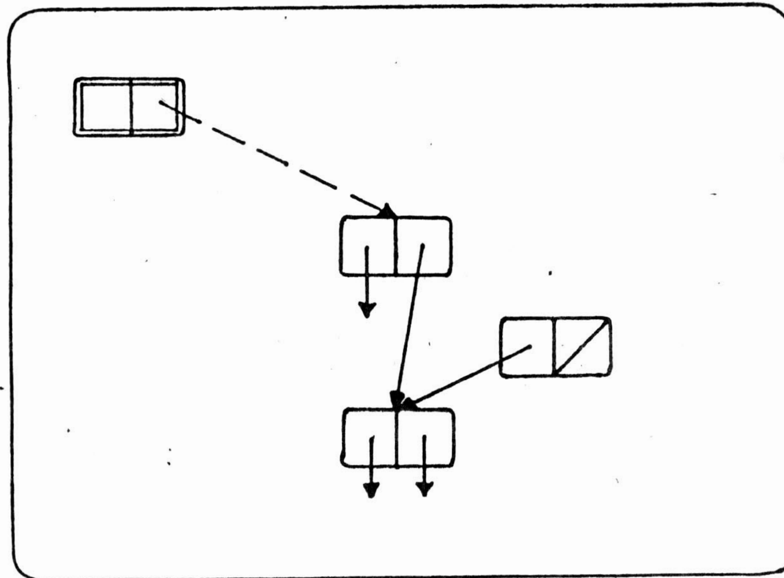
Während der Algorithmus hinsichtlich seiner Effizienz verbesserungsfähig ist, enthält er in dieser Form jedoch noch einen logischen Fehler, der korrigiert werden muß.

Nach Ausführung eines Reduktionsschrittes kann nämlich nicht garantiert werden, daß der linke Sohn des exponierten Knotens wieder ein Kombinator ist, ebenso kann die nachfolgend dargestellte Situation eintreten, in der an die Stelle des Kombinators ein ganzer Subgraph getreten ist.



Bei dem äquivalenten Kombinatorausdruck entspricht diese Konfiguration einer redundanten Linksklammerung, die natürlich entfallen

kann. Wir beheben das Problem dementsprechend, indem wir den Graphen in der Umgebung des exponierten Knotens so umformen, daß alle Verweise auf die betroffenen Knoten konsistent bleiben, die Rolle des exponierten Knotens im Gesamtgraphen jedoch von dem exponierten Knoten im Subgraphen übernommen wird:



Nach unter Umständen mehreren solchen Transformationen wird ein exponierter Knoten erreicht, dessen linker Sohn ein Kombinator ist, und der ursprüngliche Reduktionsalgorithmus kann wieder aufgenommen werden.

Wir berücksichtigen die Veränderung in unserem Algorithmus und erhalten:

procedure reduce (KG);

while **not** KG in Normalform **do**
  begin
    verfolge ausgehend von der Wurzel des KG
    die Verweise auf die rechten Subgraphen
    bis zum exponierten Knoten v_e;
    while **not** linker Sohn von v_e ist Kombinator **do**
      begin
        w := Wurzel des linken Subgraphen von v_e;
        hänge w als rechten Subgraphen an den
        Vater von v_e (auf dem Pfad von der Wurzel
        aus);
        verfolge von w aus die Verweise auf die

```
        rechten Subgraphen bis zum exponierten
        Knoten v_e
      end;
    comb := linker Sohn von v_e;
    wende die Reduktionsregel zu comb an (diese
    kann bedingt sein und/oder die Reduktion
    einiger Argumente erfordern ⟹ rekursiver
    Aufruf von reduce);
  end;
return KG
end
```

Im folgenden Abschnitt werden wir die verbalen Teile des Algorithmus weiter verfeinern und näher auf die in der konkreten Implementation verwendeten Datenstrukturen eingehen.

3.2 Datenstrukturen und Steuerung des Reduktionsalgorithmus

Aus dem in 3.1.5 skizzierten Reduktionsalgorithmus ergibt sich eine Aufteilung des Programms in zwei völlig verschiedene Gruppen von Funktionen. Einer relativ kleinen Zahl von Funktionen, die zusammen die Schleife bilden, in der die Reduktionsregeln aufgerufen werden, und die die notwendigen Datenstrukturen pflegen, stehen viele uniforme Funktionen gegenüber, die die einzelnen Reduktionsregeln enthalten. Während wir in den Punkten 3.2.2 bis 3.2.4 die erste Gruppe, die sogenannte Steuerung, und die damit assoziierten Datenstrukturen beschreiben, gehen wir in Abschnitt 3.2.5 auf die Form der Reduktionsregeln ein. Zum Verständnis der dort beschriebenen Techniken schicken wir einige Bemerkungen über die interne Repräsentation der Kombinatorgraphen voraus.

3.2.1 Abbildung der Kombinatorgraphen in Lisp-Listen

Wie wir bei der Darstellung in 3.1 schon stillschweigend angenommen haben, werden Kombinatorgraphen in unserem System durch Lisp-Listen repräsentiert, obwohl für die Implementierung prinzipiell jede Sprache in Betracht kommt, die die Manipulation dynamischer Datenstrukturen gestattet. Wir werden jedoch in Abschnitt 4.1 noch näher auf die Argumente eingehen, die Lisp als besonders geeignet erscheinen lassen.

Die Entscheidung für Lisp-Listen stellt insoweit keine Einschränkung der Repräsentierbarkeit von Kombinatorgraphen dar, als mit Hilfe der strukturverändernden Lisp-Operationen RPLACA und RPLACD sich auch zirkuläre Graphen herstellen und modifizieren lassen, so daß alle Kombinatorgraphen dargestellt werden können. Wir weisen an dieser Stelle noch einmal darauf hin, daß der Kombinatorgraph zu einem Ausdruck der Form

$$E_1 \; E_2 \; \ldots \; E_n \qquad (25)$$

nicht

$$(KG(E_1) \; KG(E_2) \; \ldots \; KG(E_n)) \qquad (26)$$

ist, sondern

$$(KG(E_n) \ldots KG(E_2)\ KG(E_1)) \quad (27)$$

wie aus der Definition in 3.1.1 hervorgeht. Für die Entscheidung gegen die intuitive Darstellung (26) spricht ein wesentlicher Effizienzvorteil. Da Kombinatorausdrücke der Form (25) implizit linksgeklammert sind, müssen sie bei der Reduktion schrittweise, von rechts beginnend, zerlegt werden:

$$\begin{aligned} & E_1\ E_2 \ldots E_{n-1}\ E_n \\ =\ & (E_1\ E_2 \ldots E_{n-1})\ E_n = \ldots \\ =\ & ((\ldots(E_1\ E_2) \ldots)\ E_{n-1})\ E_n \end{aligned}$$

Wählte man nun (26) als zugehörigen Kombinatorgraphen, so müßte diese Liste dementsprechend von rechts abgebaut werden. Der Zugriff auf die Komponenten einer Liste ist aber in Lisp (mit Rücksicht auf die interne Struktur) nur am Listenkopf effizient möglich. Um auf das letzte Element einer Liste zugreifen zu können, muß dagegen entweder die ganze Liste durchlaufen werden (zeitaufwendig!) oder die Liste doppeltverkettet abgespeichert werden (platzaufwendig!). Stattdessen entschieden wir uns für (27), um mit CAR und CDR schnell auf die Kombinatorgraphen zugreifen zu können. Ein weiteres Argument für (27) findet sich in Abschnitt 3.2.2, wo die Organisation des Stacks beschrieben wird.

Während für den Bereich der Reduktionsmaschine Kombinatorgraphen also in der Form (27) vorliegen müssen, liefert der Abstraktionsalgorithmus aus Kapitel 2 einen Kombinatorausdruck der Gestalt (26). Daraus ergibt sich die Notwendigkeit, vor Beginn der Reduktion den Kombinatorausdruck auf allen Schachtelungsebenen umzudrehen, damit eine korrekte Startsituation für den Reduktionsalgorithmus gegeben ist. Da diese Operation jedoch nur einmal zu Beginn durchzuführen ist, ist sie einer ständigen Belastung des Zeitverhaltens durch ineffiziente Listenzugriffe in jedem Fall vorzuziehen.

3.2.2 Der Stack

Welche Informationen werden nun benötigt, um eine Reduktionsregel anwenden zu können? Nach unserem Algorithmus müssen zunächst einmal der exponierte Knoten des Kombinatorgraphen und der daran hängende Kombinator gefunden werden. Dieser Kombinator legt fest, welche Regel zur Anwendung gelangt. Die Argumente für den Kombinator sind, wie schon in 3.1.2 beobachtet, gerade die linken

Subgraphen der Vorgänger des exponierten Knotens. Dank unserer Listenrepräsentation ist es leicht, für einen gegebenen Kombinatorgraphen G den exponierten Knoten zu finden: von G werden sukzessive die Listenköpfe abgespalten, bis lediglich eine einelementige Liste übrigbleibt; diese Listenzelle ist der exponierte Knoten, das einzige Element ist der gesuchte Kombinator:

G = (c b a X)
$\rightarrow$ (b a X)
$\rightarrow$ (a X)
$\rightarrow$ (X) $\Leftarrow$ der exponierte Knoten
(Im Beispiel stehen X für einen beliebigen Kombinator und a, b, c für Argumente)

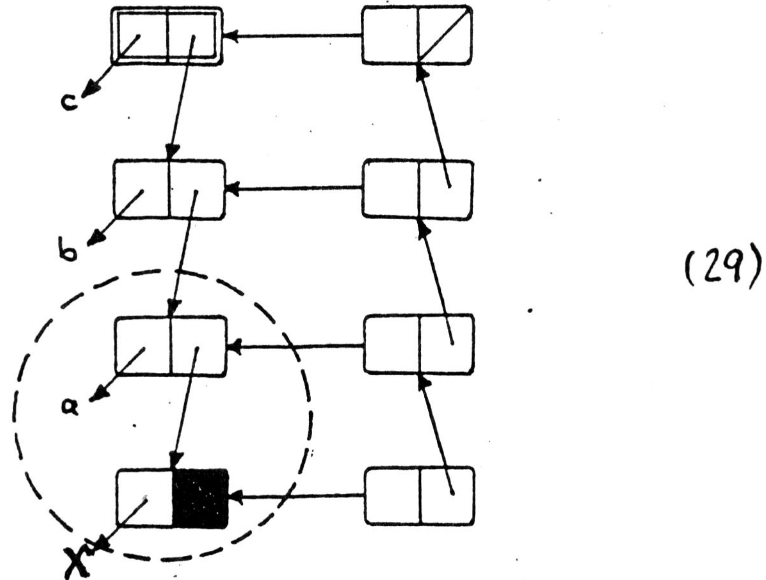
Es ergibt sich jedoch das Problem, daß die abgespaltenen Listenköpfe für die weitere Reduktion von Bedeutung sind, da sie die Argumente für den Kombinator enthalten. Um diese Informationen rekonstruieren zu können, bewahren wir alle Zwischenzustände auf dem Weg von der Wurzel zum exponierten Knoten auf einem Stack auf. In unserem Beispiel entwickelt sich der Stackinhalt folgendermaßen:

	Graph	Stack	
	(c b a X)	((c b a X))	(28)
$\rightarrow$	(b a X)	((b a X)(c b a X))	
$\rightarrow$	(a X)	((a X)(b a X)(c b a X))	
$\rightarrow$	(X)	((X)(a X)(b a X)(c b a X))	

Auf diese Weise stehen die Argumente für die anschließende Reduktion von X als Köpfe der Elemente auf dem Stack zur Verfügung.

Die Verwendung eines Stacks zur Buchführung über den Pfad von der Wurzel zum exponierten Knoten eröffnet eine entscheidende Verbesserungsmöglichkeit gegenüber der Effizienz des Reduktionsalgorithmus in seiner bisherigen Form. Danach beginnt nach Anwendung der Reduktionsregel der Kreislauf aufs Neue, wobei wiederum von der Wurzel aus der exponierte Knoten aufgesucht wird. Da dieser Pfad im allgemeinen ein Anfangsstück mit dem Pfad beim letzten Schleifendurchlauf gemeinsam hat, bedeutet dies ein wiederholtes Durchlaufen desselben Teilpfades, wobei auch der Stack genau wie beim letzten Mal aufgebaut wird. Um zu erkennen, in welcher Weise der Stack dazu benutzt werden kann, dieses gemeinsame Anfangsstück nicht noch einmal abzuarbeiten, müssen wir tiefer in die

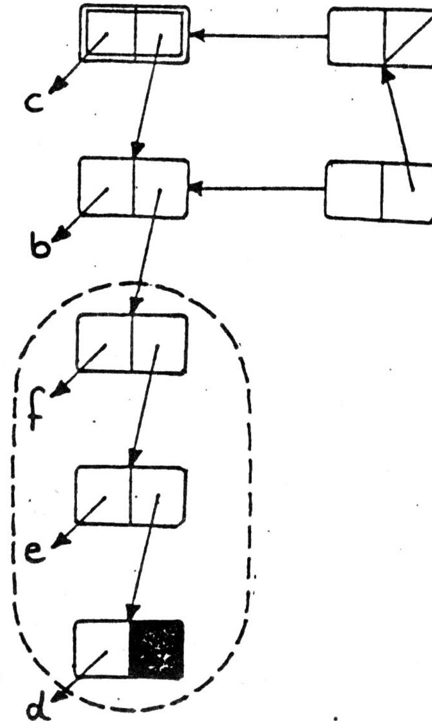
Beziehung zwischen dem Stack und dem Kombinatorgraphen eindringen. (28) vermittelt insofern einen unvollständigen Eindruck von der Situation, als daraus nicht ersichtlich wird, daß die Elemente des Stacks im Speicher auf dieselben Listenstrukturen (und nicht auf Kopien) zeigen. Tatsächlich liegt folgende Struktur vor:



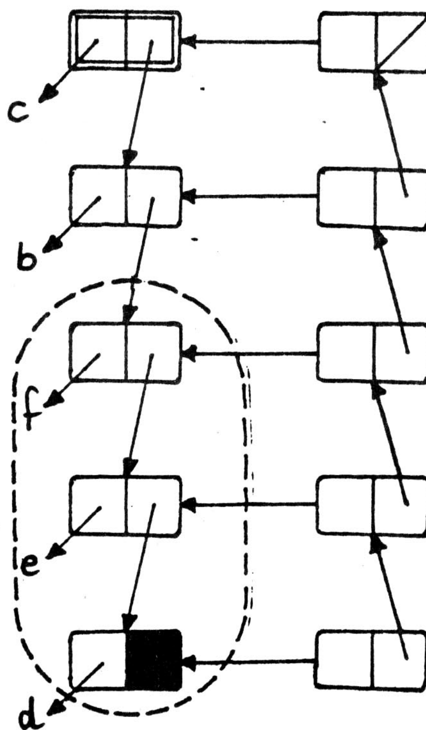
Wir können jetzt verfolgen, was bei Anwendung einer hypothetischen Reduktionsregel

X a = d e f

geschieht. Der in (29) gestrichelt abgegrenzte Teil des Kombinatorgraphen wird zusammen mit den Stackelementen, die darauf zeigen, entfernt und durch den Graphen (f e d) ersetzt:



Derjenige Teil des Stacks, der auf die obersten beiden Zellen des Kombinatorgraphen verweist, bleibt unangetastet. Anstatt ihn in gleicher Form zu reproduzieren, wird er lediglich um Verweise auf die neuen Knoten ergänzt.



Entscheidend für das Funktionieren dieser Technik ist das in (29) angedeutete **Structure Sharing** zwischen den Komponenten des

Stacks. Dieses beruht wiederum auf der Darstellung von Kombinatorausdrücken durch Listen in umgekehrter Reihenfolge, da sonst die Verkettung in die falsche Richtung weisen würde. Ohne diesen Mechanismus würden sich die Veränderungen im Kombinatorgraphen nicht auf die übernommenen Stackzellen auswirken, und der Stack hätte nach der Ergänzung den Inhalt

((d)(e d)(f e d)(b a X)(c b a X))

Weil aber auch die unveränderten Stackelemente über Zeiger im Kombinatorgraphen auf den neuen Teilgraphen verweisen, erhält der Stack ohne Manipulationen an dem beibehaltenen Teil den Wert

((d)(e d)(f e d)(b f e d)(c b f e d))

Auf diese Weise erspart man sich die kostspielige Rückkehr zur Wurzel des Kombinatorgraphen mit komplettem Neuaufbau des Stacks, die neben dem direkten Zeitaufwand auch einen gesteigerten Verbrauch an freien Listenzellen und damit häufigere Garbage Collections verursachen würde.

3.2.3 Die Funktion REDUCE

Die oberste Aufrufebene in der Reduktionsmaschine wird durch die Funktion REDUCE verkörpert. REDUCE bekommt als Argument einen Kombinatorgraphen G und liefert als Ergebnis einen Kombinatorgraphen red(G) mit gleichem Wert in Normalform (vgl. 3.1.3). Dazu führt REDUCE in einer Schleife die folgenden Schritte durch:

1. Verfolge CDR-Zellen des Kombinatorgraphen bis zum exponierten Knoten und sammle abgespaltene Listenköpfe auf dem Stack.

2. Rufe REDEXPO auf, das die passende Reduktionsregel anwendet (vgl. 3.2.4).

3. Falls keine Regel anwendbar war, liegt der Kombinatorgraph in Normalform vor $\Longrightarrow$ fertig!, sonst weiter mit 1.

Dabei ist zu beachten, daß der Stack in Schritt 1 wie oben beschrieben nur in dem Teil neu aufgebaut wird, der von der Anwendung der letzten Reduktionsregel unmittelbar betroffen wurde.

Wenn das Abbruchkriterium in Schritt 3 erfüllt ist und die Schleife verlassen wird, muß der resultierende Kombinatorgraph red(G) noch einer Nachbehandlung unterzogen werden. Damit die schon beschriebenen positiven Eigenschaften des Structure Sharings sowohl bei der Stackverwaltung als auch bei gleichen Subgraphen zur Geltung kommen, müssen G und red(G) die gleiche Listenzelle als Wurzel haben. Nur so ist gewährleistet, daß alle Verweise, die auf G gerichtet waren, und nicht nur derjenige, der die Reduktion auslöste, jetzt auf red(G) zeigen. Im Verlauf der Reduktion kann es aber vorkommen, daß die oberste Listenzelle des Graphen durch eine andere ersetzt wird. Wir lassen eine solche Veränderung während der Reduktion zu, da ein Test nach jedem Schritt zu aufwendig wäre, und machen sie am Schluß wieder rückgängig. Dazu wird noch vor Beginn der Reduktion die fragliche Listenzelle einer lokalen Variablen zugewiesen, wodurch verhindert wird, daß sie im Zuge einer Garbage Collection eingesammelt wird. Wenn am Ende red(G) vorliegt, überprüfen wir, ob sich die oberste Listenzelle verändert hat (hierzu eignet sich EQ, der Test auf interne Gleichheit), und kopieren ggf. ihren Inhalt in die ursprüngliche Listenzelle; diese wird dann als Ergebnis zurückgeliefert.

Kommentiertes Listing:

```
(defun REDUCE (E)

;Parameter:
;E = zu reduzierender Kombinatorgraph

;lokale Variablen:
;E1 = dient zur Aufbewahrung des obersten Listenknotens von E
;STACK = der lokale Stack für diesen Aufruf von REDUCE.
;WEITER = Flag, das angibt, ob die Reduktion fortgesetzt wird

  (do ((WEITER t)
       (STACK nil)
       (E1 E))
    ((null WEITER)
     ;falls die Reduktion abgeschlossen ist, wird geprüft, ob
     ;sich die oberste Listenzelle verändert hat.
     (setq E (car (last STACK)))
     (if (eq E E1)
         E  ; unverändert, keine weitere Aktion
```

```
          (rplaca (rplacd E1 (cdr E)) (car E))) )
                 ; sonst Inhalt von E in den ursprüngl.
                 ; Knoten E1 kopieren und diesen als Funktionswert
                 ; liefern
   ; sonst den nächsten Reduktionsschritt vorbereiten
   (cond
     ((null E)
       ; wenn E völlig abgebaut ist, ist das oberste Stack-
       ; element der exponierte Knoten, jetzt kann REDEEXPO
       ; aufgerufen werden
       (setq WEITER (REDEXPO STACK))
       ; wenn REDEXPO den Funktionswert NIL hat, keine
       ; weitere Aktion, WEITER erhält NIL und führt
       ; damit zum Abbruch der Schleife. Sonst wird der
       ; Stack durch die Reduktionsregel verändert. Das
       ; oberste Stackelement enthält dann den Ausdruck,
       ; mit dem der Stackaufbau fortgesetzt wird; dieses
       ; wird daher E zugewiesen.
       (if WEITER (setq E (pop STACK))) )
     ; wenn E noch nicht leer ist, wird das Zwischenergebnis
     ; auf den Stack gelegt und ein weiterer Listenkopf abge-
     ; spalten.
     (t (push E STACK)
        (pop E)) ) ) )
```

Man beachte insbesondere die Stellung von STACK als lokale Variable. Im Gegensatz zu anderen SASL-Implementationen, die nur einen Stack verwenden, besitzt in unserem System jeder rekursive Aufruf von REDUCE einen lokalen Stack. Damit läßt sich REDUCE elegant von zusätzlichem Code befreien, der die Abgrenzung von Teilstücken auf dem Stack verwalten würde.

3.2.4 Die Funktion REDEXPO

Wenn nach einigen Schleifendurchläufen in REDUCE der exponierte Knoten des Kombinatorgraphen erreicht worden ist, geht die Kontrolle an REDEXPO über, das den linken Sohn des exponierten Knotens inspiziert, um die passende Reduktionsregel zu bestimmen. Allerdings haben wir bereits gesehen (vgl. Abschnitt 3.1.5), daß es sich bei dem Sohn nicht in jedem Fall um einen Kombinator handelt; es existieren vielmehr insgesamt vier Alternativen, deren Vorliegen von REDEXPO sequentiell getestet wird. Im einzelnen können folgende Situationen vorkommen:

1. Der linke Sohn ist ein (nichttrivialer) Subgraph. Führe dann die in Abschnitt 3.1.5 beschriebene Transformation zur Beseitigung der redundanten Linksklammerung durch und gib die Kontrolle an REDUCE zurück, um den exponierten Knoten des Subgraphen zu finden.

2. Der linke Sohn ist tatsächlich ein Kombinator. Übergib die Kontrolle an die Funktion, die die entsprechende Reduktionsregel enthält, und liefere deren Ergebnis zurück.

3. Der linke Sohn ist der Name einer global definierten Funktion. Ersetze den Namen durch einen Zeiger auf den bei der Definition der Funktion erzeugten Kombinatorgraphen und gib die Kontrolle an REDUCE zurück.

4. Wenn 1. bis 3. nicht zutreffen, handelt es sich um eine Konstante, die nicht weiter reduziert werden kann. Melde an REDUCE den Abbruch der Reduktion.

REDEXPO teilt der aufrufenden Funktion über sein Ergebnis mit, ob ein Reduktionsschritt ausgeführt werden konnte. Falls keine Transformation anwendbar war, liegt der Graph in Normalform vor und die Schleife in REDUCE kann gleichfalls verlassen werden. Ein Funktionswert von NIL bedeutet dabei "Abbruch der Reduktion", jeder andere Wert führt zur Fortsetzung. In den Fällen 1 und 3 wird in jedem Fall eine Transformation vorgenommen, dementsprechend ist der Funktionswert dort stets ungleich NIL. Umgekehrt ist das Ende der Reduktion erreicht, wenn der Sohn des exponierten Knotens eine Konstante ist. Daher wird im Fall 4 NIL zurückgeliefert. Liegt Situation 2 vor, hängt die Anwendbarkeit einer Graphtransformation von der Reduktionsregel selbst ab, die ja z.B. Bedingungen an die Argumente enthalten kann. Daher verlangen wir auch von den Ergebnissen der Reduktionsregeln die gleiche Bedeutung wie bei dem Funktionswert von REDEXPO und leiten sie unverändert an REDUCE weiter.

Die Zuordnung von Reduktionsregeln zu Kombinatoren geschieht über eine globale Assoziationsliste RULES, die Paare von der Form

(Kombinator Funktionsname)

enthält. Die Entscheidung, diese Zuordnung aus dem Programm auszugliedern und in eine davon getrennte Datenstruktur zu verlegen, stellte sich insbesondere bei Entwicklung und Test der Reduktionsmaschine als sehr vorteilhaft heraus, da allein durch Editieren der Assoziationsliste neue Reduktionsregeln in das System eingefügt oder zu Testzwecken einzelne Regeln gezielt deaktiviert werden konnten.

Kommentiertes Listing:

```
(defun REDEXPO (STACK)

;lokale Variablen:
;SON = der linke Sohn des exponierten Knotens
;F    = Falls SON tatsächlich Kombinator ist, erhält
;       F als Wert das SON entsprechende Paar in
;       der Assoziationsliste, sein CDR ist der Aufruf
;       der Reduktionsfunktion.

  ; Damit die Reduktionsfunktionen dynamisch auf den STACK
  ; zugreifen können, muß er als Special-Variable deklariert
  ; werden:
  (declare (special STACK))
  (let ((SON (caar STACK))
        F)
    (cond
       ((consp SON)
       ; SON ist selbst ein Kombinatorgraph. Führe Transformation
       ; zur Eliminierung der Linksklammerung durch. Dabei erst
       ; Sonderfall für Wurzel = exponierter Knoten behandeln.
        (cond ((null (cdr STACK)) (rplaca STACK SON) )
              (t (rplacd (cadr STACK) SON)
                 (pop STACK) )))
       ((setq F (assoc SON RULES))
       ; SON ist Kombinator, d.h. für ihn besteht ein Eintrag in
       ; der Liste RULES. Werte die mit ihm verbundene Reduktions-
       ; funktion aus und gib deren Wert zurück.
        (eval (cdr F)) )
       ((member SON USERFUNS)
       ; SON ist der Name einer global definierten Funktion. Der
       ; zugehörige Kombinatorgraph ist Wert der globalen Variable
       ; gleichen Namens und ersetzt den Inhalt des exponierten
```

```
; Knotens.
 (rplaca (car STACK) (eval SON)) )
; SON ist Konstante. Keine Reduktion erforderlich.
(t nil)) ))
```

3.2.5 Eine generische Form für Reduktionsregeln

Die im vorigen Abschnitt eingeführte Assoziationsliste RULES ordnet jedem Kombinator den Namen einer Lisp-Funktion zu, die die zugehörige Reduktionsregel verkörpert. Trotz der Verschiedenartigkeit der Kombinatoren orientieren sich diese **Reduktionsfunktionen** durchgängig an einem gemeinsamen Grundmuster. Alle Reduktionsfunktionen verändern per Seiteneffekt einen kleinen Ausschnitt des Kombinatorgraphen und liefern als Ergebnis NIL, wenn nichts zu reduzieren war, bzw. irgend einen anderen Wert sonst. Darüber hinaus besitzt der Funktionswert keine Bedeutung. Bei der Graphtransformation dient der Stack als Schnittstelle zum Kombinatorgraphen; er wird nicht explizit als Parameter an die Reduktionsfunktion übergeben, stattdessen kann auf die gerade gültige Instanz des Stacks wegen des Dynamic Scopings in Lisp ohnehin zugegriffen werden.

Wir betrachten nun die Schritte näher, die bei der Anwendung der Reduktionsregel für einen n-stelligen Kombinator

$$X\ a_1\ \ldots\ a_n = E \qquad (30)$$

durchzuführen sind.

Zuerst wird überprüft, ob >= n+1 Elemente auf dem Stack vorhanden sind. Ist der Stack kürzer, so fehlen Argumente und die Reduktion unterbleibt; zudem muß an REDEXPO der Funktionswert NIL zurückgeliefert werden. Liegen aber genügend Argumente vor, so werden die obersten n Elemente des Stacks entfernt, da sie bei der darauffolgenden Transformation ungültig werden. Das oberste Stackelement nach dieser Operation ist gerade

$$(a_n\ \ldots\ a_1\ X)$$

Es empfiehlt sich, diesen Zusammenhang an dem Beispiel in Abschnitt 3.2.2 nachzuvollziehen.

Da diese Stackmanipulation zu Beginn aller Reduktionsfunktionen

vorkommt, wurde sie in eine eigene Hilfsfunktion POPT (POP and Test) verlegt. POPT entfernt n Elemente vom Stack. Falls danach noch mindestens ein weiteres Element auf dem Stack verbleibt (auf dem die Reduktionsregel operieren kann), wird NIL, sonst T zurückgeliefert:

```
(defun POPT (N)
  ; auf STACK wird über dynamische Bindung
  ; zugegriffen:
  (declare (special STACK))
  (let ((newstack (nthcdr n STACK)))
    (cond ((null newstack) t)
          (t (setq STACK newstack)
             nil) )))
```

Nachdem die Argumente des Kombinators auf diese Weise bereitgestellt worden sind, werden ggf. einige von ihnen selbst reduziert. Dazu reicht es, REDUCE rekursiv auf (CAR (CAR STACK)), (CADR (CAR STACK)), ... anzuwenden. An dieser Stelle kommt uns die Eigenschaft von REDUCE, bei der Reduktion die Wurzelzelle zu erhalten, sehr zustatten. Ihretwegen brauchen wir das Reduktionsergebnis nicht mit umständlichen Sequenzen von RPLACA und RPLACD im obersten Stackelement zu aktualisieren, sondern der dortige Zeiger weist automatisch auf den reduzierten Subgraphen.

Nach der Reduktion kann überprüft werden, ob die Argumente den Bedingungen der Reduktionsregel genügen und welche der alternativen Regeln im Einzelfall anzuwenden ist. Meist wird mit diesen Bedingungen die Einhaltung gewisser Typbeschränkungen gesichert (z.B. müssen die Argumente eines arithmetischen Kombinators numerisch sein).

Zum Schluß wird die Graphtransformation tatsächlich durchgeführt. Dazu wird die Wurzelzelle mit einem neuen Inhalt gefüllt, der dem Ausdruck E in der Regel (30) entspricht. Zur Modifikation des Zelleninhalts benutzen wir die Lisp-Funktionen RPLACA und RPLACD, die - anders als CONS - bestehende Listenstrukturen verändern, anstatt neue zu schaffen. Das Ergebnis von RPLACA bzw. RPLACD ist im übrigen die modifizierte Listenzelle, also insbesondere ungleich NIL und damit als Funktionswert der Reduktionsfunktion geeignet, um die Fortsetzung der Reduktion zu signalisieren.

Alternativ kann auch festgestellt werden, daß die Bedingungen für eine erfolgreiche Reduktion nicht erfüllt sind, obwohl genügend

Argumente vorhanden sind. In diesem Fall ist das Reduktionsergebnis $\perp$, was in der Regel auf einen Programmierfehler hindeutet. Daher ist die Ausgabe einer diagnostischen Fehlermeldung wünschenswert. Für die Behandlung von Fehlern (**error values**) hat sich in den bisherigen SASL-Implementationen noch kein Standard etabliert; unsere Variante wird im nächsten Abschnitt vorgestellt.

Abschließend betrachten wir die Funktion REDHD, die die Reduktionsregel für den Kombinator hd enthält. Sie bietet ein überschaubares Beispiel für alle in diesem Abschnitt dargestellten typischen Komponenten einer Reduktionsfunktion.

Kommentiertes Listing:

```
(defun REDHD ()
  ; auf STACK wird über dynamische Bindung
  ; zugegriffen:
  (declare (special STACK))
  ; hd ist einstellig, daher wird getestet ob ein
  ; Argument auf dem Stack liegt. Gleichzeitig
  ; wird das oberste Stackelement entfernt.
  (cond
    ((POPT 1) nil)
  ; TOS wird als Schreibabkürzung für das neue
  ; oberste Stackelement eingeführt.
    (t
     (let ((TOS (car STACK)))
       ; Da hd nur auf nichtleeren Listen definiert
       ; ist, muß das Argument reduziert werden ...
       (REDUCE (car TOS))
       ; ... und anschließend auf diese Eigenschaft
       ; getestet werden (Prädikat SASL-CONSP)
       (cond
          ((SASL-CONSP (car TOS))
          ; Die Bedingung ist erfüllt. Da (CAR TOS)
          ; von der Form (y x P) ist, erreicht man
          ; x, den hd der Liste als (CADAR TOS).
          (rplaca TOS (cadar TOS))
          (rplacd TOS nil) )
          ; Das Argument war keine Liste. Daher wird
          ; eine Fehlermeldung auf die Konsole
          ; ausgegeben (genauer Ablauf in 3.2.6)
          (t (MKERR `(HD AUF ,(CAR TOS) ANGEWANDT)) )))))
```

3.2.6 Die Behandlung von Laufzeitfehlern

Im vorhergehenden Abschnitt wurde die Frage aufgeworfen, welche Maßnahmen zu ergreifen sind, wenn eine Reduktion daran scheitert, daß die Argumente den an sie gestellten Einschränkungen nicht genügen. Die Fehlersuche würde durch informative Fehlermeldungen erheblich erleichtert; da der Bezug zwischen Kombinatorgraph und SASL-Quelltext aber im Verlauf der Reduktion verloren geht, ist es außerordentlich schwierig, solche Fehlermeldungen zu generieren. Noch illusorischer ist es, dem Benutzer Zugang zum aktuellen Kombinatorgraphen zu geben (analog zum Break Package in Lisp), so daß nach einigen manuellen Modifikationen die Reduktion wieder aufgemommen werden kann. Unsere Bemühungen konzentrieren sich daher eher darauf, Fehlersituationen im Augenblick ihres Entstehens aufzudecken und Informationen über die daran beteiligten Argumente zu geben.

Dieses verkleinerte Problem scheint auf den ersten Blick sehr einfach zu lösen zu sein, da die Applikation der Reduktionsregeln innerhalb von Lisp-Funktionen stattfindet. Diese können bei Bedarf Meldungen mit den Werten von ausgewählten Argumenten ausgeben, auf die über den Stack Zugriff besteht. Leider läßt sich diese Methode nicht auf benutzerdefinierte SASL-Funktionen übertragen, da für diese kein Lisp-Code, sondern ein Kombinatorgraph erzeugt wird. Es gilt also, die Fehlerbehandlung innerhalb des Kombinatorkonzepts selbst zu realisieren, damit bei der Abstraktion von globalen Definitionen entsprechende Subgraphen in den Kombinatorgraphen eingebaut werden können (vgl. hierzu die Darstellung in 2.7).

Wir führen dazu einen einstelligen Pseudokombinator BOTOM ein, der nach der Regel

BOTOM x = BOTTOM

reduziert wird. Das Besondere an diesem Kombinator ist, daß x <u>kein</u> SASL-Objekt ist, sondern eine Fehlermeldung in Form einer Lisp-Liste. Die Anwendung der Reduktionsregel bewirkt als Seiteneffekt die Ausgabe der Liste auf die Konsole.

Die Erzeugung einer Fehlermeldung zerfällt demnach in zwei Phasen. In der ersten Phase wird bei der Anwendung der Reduktionsregel für einen Kombinator X festgestellt, daß die Bedingungen für

die Ausführung nicht erfüllt sind. Stattdessen wird nach einer zusätzlichen Regel

$$X\ a_1\ \dots\ a_n = \text{BOTOM fehlermeldung} \qquad (31)$$

reduziert. Im zweiten Schritt hängt BOTOM am exponierten Knoten, wird seinerseits zu BOTTOM reduziert und löst dabei die Fehlermeldung aus. Hat die Fehlersituation ihren Ursprung in einer vom Benutzer eingeführten SASL-Funktion, so enthält der zugehörige Kombinatorgraph bereits von Anfang an Vorkommen von BOTOM und die erste Phase entfällt.

Da Transformationen nach (31) sehr häufig anzutreffen sind, gibt es dafür den Macro

```
(defmacro' MKERR (MSG)
  `(rplacd (rplaca TOS ,MSG) 'BOTOM))
```

die als Parameter die gewünschte Fehlermeldung erhält. Ein Anwendungsbeispiel findet sich in der in Abschnitt 3.2.5 vorgestellten Reduktionsfunktion REDHD.

Die tatsächliche Ausgabe der Fehlermeldung erfolgt als Seiteneffekt bei der Reduktion von BOTOM. Die dafür vorgesehene Funktion PRT-ERROR arbeitet die in Form einer Lisp-Liste übergebene Fehlermeldung elementweise ab, wobei Atome unverändert ausgegeben werden (hierunter fällt auch der Klartext der Fehlermeldung). Listen werden nur durch das Wort "LISTE" wiedergegeben, um nichtterminierende Ausgaben innerhalb von Fehlermeldungen zu vermeiden. Funktionale Objekte werden in gleicher Weise wie bei der regulären Ausgabekomponente (vgl. 3.1.4) behandelt.

4. Einzelheiten der Implementierung

In Kapitel 3 haben wir den Reduktionsalgorithmus und seine Implementierung bewußt auf einem Niveau beschrieben, das eine Übertragung auf andere Implementierungssprachen ohne prinzipielle Veränderungen gestatten sollte. Wir kommen nun auf die Wahl von Lisp zurück und stellen die Konsequenzen dieser Entscheidung dar (Punkte 4.1 und 4.2). Die letzten beiden Abschnitte beschäftigen sich mit der Stellung von SASL als Subsystem in der Lisp-Umgebung.

4.1 Die Entscheidung für Lisp

Der Entwurf der Reduktionsmaschine orientierte sich an der Zielsetzung, im Sinne des Rapid Prototyping möglichst schnell zu einem lauffähigen Programm zu gelangen. Dieses Ziel wurde hauptsächlich durch die Entscheidung für Lisp als Implementierungssprache erreicht, da viele Lowlevel-Komponenten (wie Garbage Collection) nicht neu entwickelt werden mußten. Zudem trägt diese Sprache den Erfordernissen sowohl der Abstraktion als auch der Graphreduktion Rechnung.

Im Bereich des Abstraktionsalgorithmus ist der Vorteil von Lisp klar ersichtlich, da die Transformation der eingegebenen SASL-Ausdrücke in Kombinatorgraphen intensive Symbolmanipulation bedingt. Zudem gestattet die Codierung der SASL-Ausdrücke als Lisp-Listen, auf die Entwicklung eines SASL-Parsers zu verzichten und sich stärker auf die Probleme von Abstraktion und Reduktion zu konzentrieren.

Im Bereich der Reduktion erscheint zunächst eine imperative Programmiersprache geeigneter, um die in /Turner79a/ vorgeschlagene und ganz auf von Neumann-Rechner abgestellte Stack-Maschine zu realisieren. Es zeigte sich jedoch, daß durch disziplinierte Verwendung der nicht-funktionalen Sprachelemente in Lisp eine adäquate und effiziente Beschreibung des Reduktionsalgorithmus möglich ist. Ein gutes Beispiel für diese Behauptung stellt die generische Form einer Reduktionsregel in unserem System dar (vgl. Abschnitt 3.2.5).

Die ursprüngliche Version wurde auf einem Apple II im Dialekt muLisp-80 entwickelt. Inzwischen konnte die Portierung auf eine

Symbolics Lisp-Maschine unter Common Lisp jedoch abgeschlossen werden.

4.2 Effizienzsteigernde Maßnahmen

Mit der Entscheidung für Lisp nahmen wir gegenüber einer Lösung in der Maschinensprache des Zielrechners eine verminderte Arbeitsgeschwindigkeit in Kauf. Die naheliegende Lösung, häufig durchlaufene Programmteile - wie die Funktionen REDUCE und REDEXPO - durch handoptimierten Maschinencode zu ersetzen, wurde auf dem Apple mit Erfolg erprobt, jedoch zugunsten besserer Portabilität zurückgestellt. Wir konzentrierten uns stattdessen auf die nachfolgend beschriebenen maschinenunabhängigen Optimierungsansätze.

4.2.1 Recycling von CONS-Zellen

So konnten wir ein in /Stoye84/ dargestelltes Phänomen in unserem System nachvollziehen und in abgewandelter Form zur Effizienzverbesserung ausnutzen. Stoye et al. hatten festgestellt, daß ein sehr großer Anteil der bei der Graphreduktion verbrauchten freien Listenzellen nach verhältnismäßig kurzer Zeit wieder freigesetzt wird. Die durch diesen hohen Umsatz ausgelösten häufigen Garbage Collections können das Zeitverhalten kleinerer Systeme merklich belasten.

Im Zuge der Reimplementierung einiger Programmteile in Maschinensprache konnten wir zumindest bei den Grundkombinatoren Abhilfe schaffen. Beispielsweise werden bei der Reduktion des B-Kombinators (der häufigsten Reduktion überhaupt) drei neue Listenzellen verbraucht. Anstatt diese Listenzellen der Free-List des Systems zu entziehen, verwenden wir direkt die gleichzeitig freiwerdenden drei Stackzellen wieder und versehen sie mit neuem Inhalt. Das Geschwindigkeitsverhalten des Systems verbesserte sich daraufhin um etwa 10%.

In einer weiteren Ausbaustufe der Common Lisp-Version werden wir versuchen, durch die Einführung geeigneter Makros das Recycling von CONS-Zellen zu ermöglichen, ohne daß die Übersichtlichkeit des Quellcodes der Reduktionsfunktionen darunter leidet.

Im gleichen Zug werden wir auch die übrigen Anregungen aus

/Stoye84/ auf Verwertbarkeit untersuchen. Wegen der in dieser Hinsicht unflexiblen Hardware des Apple II konnten die Vorschläge zur Einführung von Reference Counts in den Kombinatorgraphen und zur Umgehung der Rekursion im Reduzierer bislang nicht weiter verfolgt werden.

4.2.2 Superkombinatoren

Während es sich anbietet, die Reduktionsfunktionen der Grundkombinatoren in Maschinensprache zu verlegen, stellt sich am entgegengesetzten Ende des Spektrums die Frage, inwieweit nicht Funktionen des Preludes Kombinatoren gleichgestellt werden sollten. Sie werden zwar nicht erst während des Abstraktionsprozesses in den Kombinatorausdruck eingebaut, sondern sind schon in dem ursprünglichen SASL-Ausdruck enthalten, können aber bei der Reduktion in völlig gleicher Weise behandelt werden.

Die Abgrenzung dieser Kategorie fällt schwer, da sie nach oben offen ist. Ganz sicher finden sich hier die arithmetischen und logischen Operationen, die Typzugehörigkeitsprädikate sowie das Gleichheitsprädikat. Charakteristisch für diese Kombinatoren sind bedingte Reduktionsregeln (wie bei den Listenkombinatoren), die die Reduktion eines oder mehrerer Argumente vor ihrer Anwendung erfordern.

Ein typisches Beispiel bietet die Reduktionsregel für die Addition

```
plus m n = m+n
```

die nur anwendbar ist, wenn n und m vorher zu Zahlen reduziert werden konnten.

Das entscheidende Argument für die Behandlung von primitiven Funktionen als Kombinatoren liegt in der Effizienz. So ist in der Regel auch bei komplexen Reduktionsregeln die Auswertung durch sorgfältig optimierte Lisp-Funktionen der Abarbeitung einer SASL-Definition geschwindigkeitsmäßig überlegen. Es stellt sich also die Frage, in welchem Umfang Prelude-Funktionen, die sonst zu Beginn einer Sitzung in Quellform geladen würden, auf Kombinatorniveau implementiert werden. In /Stoye84, S.165/ und /Hughes82/ sehen die Autoren die Verwendung von solchen **Superkombinatoren** als konsequente Weiterführung des Kombinatorkonzepts.

Auch wir machten von Superkombinatoren Gebrauch und verlegten u.a. häufig vorkommende Listenoperationen wie member, filter, map etc. auf die Ebene der Implementierungssprache Lisp. Im Anhang sind diejenigen Prelude-Funktionen, die wir als Superkombinatoren realisiert haben, besonders gekennzeichnet.

Eine Sonderstellung nehmen schließlich die in Abschnitt 2.8 zur Elimination von ZF-Ausdrücken eingeführten Funktion ocp und join ein. Um ihre effiziente Behandlung zu gewährleisten, haben wir diese beiden Funktionen als Kombinatoren CP und J realisiert. Dabei konnten die Definitionen von ocp für CP und von join für J unverändert übernommen werden.

4.2.3 Telescoping

Schließlich kann es in Einzelfällen zu Verbesserungen führen, die Reduktionsregeln der Kombinatoren daraufhin zu überprüfen, ob mehrere Reduktionsschritte zusammengelegt werden können. Oftmals ist in bestimmten überschaubaren Situationen die Strategie der Lazy Evaluation gar nicht erforderlich, man kann sie dann eliminieren und vermeidet ein überflüssiges Aufblähen des Kombinatorgraphen.

Beispielhaft für die Anwendung dieses Prinzips ist die Reduktion des Superkombinators length. Length wird nach der Regel

```
length [ ]     = 0
length (P a b) = 1 + (length b)
```

reduziert. Wie man sofort sieht, ist im zweiten Fall eine weitere Reduktion von length notwendig. Da die Vorteile der Lazy Evaluation wegen der Striktheit der Addition in keinem Fall zum Tragen kommen, können wir alle Reduktionsschritte für length bis zum Erreichen der leeren Liste sowie die Additionen in einem einzigen Aufruf der Reduktionsfunktion für length durchführen. Dadurch erreichen wir:

1. höhere Geschwindigkeit, da die Rückkehr zu REDUCE und REDEXPO zwischen den Reduktionsschritten entfällt.

2. geringeren Speicherplatzbedarf und niedrigeren

Umsatz an freien Listenzellen, da die Zwischenstadien des Kombinatorgraphen nicht erzeugt werden.

4.3 Das System aus der Sicht des Benutzers

Der SASL-Interpreter präsentiert sich dem Benutzer als ein Subsystem der Lisp-Umgebung. Bedingt durch die Ersatzdarstellung der SASL-Ausdrücke als Lisp-Listen können die Ressourcen des Lisp-Systems (etwa syntaxorientierte Editoren) voll für die Bearbeitung von SASL-Programmen genutzt werden.

Ein SASL-Programm besteht typischerweise aus einer Anzahl von globalen Funktionsdefinitionen (dem **Script**) und einem Funktionsaufruf am Schluß. Unser System erlaubt jedoch zugunsten einer interaktiven Arbeitsweise, globale Definitionen und Auswertungen von SASL-Ausdrükken in beliebiger Reihenfolge zu mischen. Dazu existieren zwei Toplevel-Funktionen in Lisp: SASL-EVAL und SASL-DEF.

SASL-EVAL erhält als Parameter einen SASL-Ausdruck E in Lisp-Notation. In E vorkommende WHERE-Ausdrücke und ZF-Ausdrücke werden durch Abstraktion eliminiert (vgl. 2.6 und 2.8), anschließend wird der entstandene Kombinatorausdruck reduziert und das Ergebnis von der Ausgabekomponente in die externe Repräsentation überführt.

Globale Funktionen oder Variablen werden durch SASL-DEF eingeführt. SASL-DEF bekommt als Parameter eine (evtl. mehrzeilige) Funktionsdefinition in Form einer Liste, deren Elemente die Zeilen sind, die wiederum jeweils aus zwei Listen bestehen: der ersten mit Funktionssymbol und formalen Parametern und der zweiten mit dem definierenden Ausdruck.

Bsp.: (SASL-DEF '(((F x_1 .. x_n) E_1) ... ((F y_1 .. y_m) E_k)))

Damit entspricht die Notation derjenigen in der lokalen Definition eines WHERE-Ausdrucks. SASL-DEF abstrahiert sofort die formalen Parameter, so daß ein Kombinatorgraph, der F entspricht, als Wert des Atoms F abgespeichert werden kann. Parallel dazu wird auch die originale SASL-Definition in L-Notation aufbewahrt. Trat beim Definieren kein Fehler auf, so wird F in die globale Liste USERFUNS aufgenommen, die bei Anwendung von F von der Funktion REDEXPO zur Ermittlung der Art des Reduktionsschritts

getestet wird. USERFUNS enthält zu Beginn einer Sitzung nur die Namen der Preludefunktionen, die nicht als Superkombinatoren realisiert sind.

Zusätzlich zu SASL-DEF steht in unserem System noch die Funktion SASL-REDEF zu Verfügung; SASL-REDEF definiert bereits eingegebene Definitionen in ihrer alten Form noch einmal. Dies kann nötig werden, wenn eine von mehreren wechselseitig rekursiven Funktionen geändert wurde und zuvor bereits eine Graphreduktion durchgeführt worden war.

4.4 Die Einbettung von SASL in Lisp

Bei jeder Implementierung einer Programmiersprache in einer anderen stellt sich die Frage nach der Kommunikation zwischen den beiden Sprachen. Dabei stehen prinzipiell zwei gegensätzliche Philosophien zur Wahl:

I) Die Implementierungssprache bleibt für den Benutzer der neuen Sprache völlig unsichtbar.

II) Teile der Implementierungssprache stehen neben der neuen Sprache weiterhin zur Verfügung.

Während etwa Compiler typischerweise in die erste Kategorie fallen, verfolgt man gerade in Lisp häufig die zweite Philosophie, indem man die Sprache modular um neue Konstrukte (z.B. Flavors) erweitert.

Obwohl wir in den bisherigen Ausführungen Lisp ausschließlich im Sinne von I) betrachtet haben, könnte das SASL-System auch zu einer vollwertigen Extension von Lisp ausgebaut werden. Analog zum Flavor-System ließen sich die Programmteile, in denen Lazy Evaluation, nichtstrikte Funktionen etc. vorteilhaft ausgenutzt werden können, in SASL formulieren. Der Aufruf dieses Programms aus Lisp heraus könnte dann über die bereits vorgestellte Funktion SASL-EVAL erfolgen, die ihre Ergebnisse direkt an die aufrufende Lisp-Funktion zurückgäbe anstatt sie auf den Bildschirm auszugeben.

Alternativ wäre zu untersuchen, ob SASL eine geeignete Ausgangsbasis für eine Vereinheitlichung funktionaler und logischer Programmiersprachen darstellt, wie sie beispielsweise im LISPLOG-

Projekt /Boley85/ angestrebt wird. Es ist denkbar, daß die Probleme, die sich bei einer uneingeschränkten Verwendung der in LISPLOG erlaubten Durchgriffe von Prolog auf Lisp ergeben können, durch eine Beschränkung auf Aufrufe einer rein funktionalen Sprache wie SASL teilweise entschärft werden können.

Anhang: Das SASL-Prelude

Dieser Anhang enthält alle in unserem SASL-System vordefinierten Funktionen mit einer kurzen Beschreibung ihrer Bedeutung.
Die Funktionen zerfallen in drei Klassen:

1) Solche, die sich nicht aus einfacheren Funktionen zusammensetzen lassen und daher direkt als Kombinatoren reduziert werden müssen. Diese Funktionen nennen wir primitiv.
2) Funktionen, die zwar mit Hilfe aus den unter 1) fallenden primitiven Funktionen selbst definiert werden können, die wir aber aus Effizienzgründen direkt als Kombinatoren implementiert haben (dazu zählen auch die Superkombinatoren, vgl. 4.2.2.). Solche Funktionen sind mit einem Stern "*" markiert.
3) Sonstige nützliche Funktionen, deren SASL-Definitionen bereits vorab in die Liste der benutzerdefinierten Funktionen aufgenommen wurden.

Soweit Funktionen sich auf einfachere zurückführen lassen, ist auch ihre SASL-Definition aufgenommen.

abs n *

Absolutbetrag von n

abs n = n < 0 → -n ; n

---

all L

Konjunktion über eine Liste von booleschen Werten

all = foldr and TRUE

---

and x y *

Konjunktion; der zweite Parameter wird nur ausgewertet, wenn der erste TRUE ergibt

and x y = x → y ; FALSE

Alternative Schreibweise: x & y

---

any L

Disjunktion über eine Liste von booleschen Werten

any = foldr or FALSE

---

append x y *

Listenkonkatenation von **x** und **y**

```
append [ ]   y = y
append (a:x) y = a : append x y
```

Alternative Schreibweise: x ++ y

---

arctan x

inverse trigonometrische Funktion - primitiv

---

boolean b *

Typzugehörigkeitsprädikat

```
boolean TRUE  = TRUE
boolean FALSE = TRUE
boolean x     = FALSE
```

---

char x

Typzugehörigkeitsprädikat - primitiv

---

cjustify n x

Formatierfunktion: zentriert **x** in **n** Zeichen Breite durch Vor- und Nachsetzen von Spaces

```
cjustify n x = spaces lmargin : x : spaces rmargin
               where
               margin  = n - printwidth x
               lmargin = margin div 2
               rmargin = margin - lmargin
```

---

code c

liefert zu einem Character c dessen ASCII-Code - primitiv

---

concat L

verkettet alle Elemente einer Liste von Listen mit append

```
concat L = foldr append [ ]
```

---

cons x y *

Paarbildungsoperation - primitiv

Alternative Schreibweise: x:y

---

converse f x y *

Vertauschen der Argumente

converse f x y = f y x

---

cos x

trigonometrische Funktion - primitiv

---

count m n

bezeichnet die Liste der ganzen Zahlen zwischen m und n

count m n = m > n ⟶ [] ; m : count (m+1) n

Alternative Schreibweise: m..n

---

decode n

liefert den Character mit ASCII-Code n - primitiv

---

div m n

ganzzahlige Division - primitiv

---

dot f g *

Komposition der Funktionen f und g

dot f g = h WHERE h x = f (g x)

Alternative Schreibweise: f . g

---

drop n L

liefert den Rest der Liste L nach Entfernen der ersten n Elemente

```
drop n (a:x) = n > Ø ⟶ drop (n-1) x ; a:x
drop n []    = []
```

---

entier x

liefert die größte ganze Zahl kleiner oder gleich x - primitiv

---

eq x y *

Gleichheitsoperation - primitiv

```
eq x y = function x & function y ⟶ ⊥
         not (list x) | not (list y) ⟶ x=y (elementar)
         eq (hd x) (hd y) & eq (tl x) (tl y)
```

Alternative Schreibweise: x = y

---

exp x

Exponentialfunktion - primitiv

---

filter p L *

liefert die Liste derjenigen Elemente von L, auf die das Prädikat p zutrifft

```
filter p (a:x) = (p a) ⟶ a : filter p x
                          filter p x
filter p []    = []
```

---

foldl f r L *

"faltet" die Liste L nach links mit der Operation f. r ist das neutrale Element von f.
Für die endliche Liste $L=[a_1,...,a_n]$ gilt:

foldl f r L = f a_n (f a_{n-1} (...(f a_1 r)...)

```
foldl f r (a:x) = foldl f (f r a) x
foldl f r []    = r
```

---

foldr f r L *

"faltet" die Liste L nach rechts mit der Operation f. r ist das neutrale Element von f.
Für die endliche Liste $L=[a_1,...,a_n]$ gilt:

foldr f r L = f a_1 (f a_2 (...(f a_n r)...)

```
foldr f r (a:x) = f a (foldr f r x)
foldr f r []    = r
```

---

for a b f

liefert das Bild der Liste der Zahlen zwischen a und b unter der Abbildung f

```
for a b f = map f (a..b)
```

---

from n

liefert die Liste der ganzen Zahlen ab n in aufsteigender Folge

```
from n = n : from (n+1)
```

Alternative Schreibweise: n...

---

function x

Typzugehörigkeitsprädikat - nicht implementiert

---

gt/ge/lt/le x y

arithmetische Vergleichsoperatoren - primitiv

Alternative Schreibweisen: x > y usw.

---

hd x

liefert das erste Element einer nichtleeren Liste - primitiv

```
hd (a:x) = a
```

---

if b x y

bedingter Ausdruck - primitiv

Alternative Schreibweise: b → x ; y

---

intersection L M

Mengendurchschnitt von L und M

```
intersection L M = filter (member L) M
```

---

iterate f x

liefert die Liste [x, f x, f (f x), f (f (f x)), ...]

```
iterate f = g WHERE g x = x : g (f x)
```

---

lay L

Formatierfunktion: wendet "show" auf jedes Element der Eingabeliste L an und fügt dazwischen Zeilenvorschübe ein

```
lay []     = []
lay (a:x)  = show a : '<CR>' : lay x
```

---

layn L

Formatierfunktion wie "lay" mit zusätzlicher Zeilennumerierung

```
layn L = f 1 L
         where
         f n []    = []
         f n (a:x) = n : ") " : show a : '<CR>' : f (n+1) x
```

---

length L *

liefert die Länge einer Liste. Dieser Operator liefert nur auf endlichen Listen einen von $\perp$ verschiedenen Wert.

```
length []    = 0
length (a:x) = 1 + length x
```

Alternative Schreibweise: # L

---

list x

Typzugehörigkeitsprädikat - primitiv

---

listdiff L M

Listendifferenz von L und M. Bei mehrfach vorkommenden Elementen wird in für jedes Vorkommen in M jeweils das erste Vorkommen in L entfernt. Beispiel:

```
listdiff [1,4,1,9,3] [5,1,4,6,4] = [1,9,3]

listdiff []     M     = []
listdiff L      []    = L
listdiff (a:x) (b:y) = a=b ⟶ listdiff x y
                        listdiff (a : listdiff x [b]) y
```

Alternative Schreibweise: L -- M

---

ljustify n x

Formatierfunktion: füllt hinter x soviele Spaces ein, daß eine Gesamtlänge von n erreicht wird

```
ljustify n x = x : spaces (n - printwidth x)
```

---

log x

natürlicher Logarithmus - primitiv

---

logical x *

Typzugehörigkeitsprädikat

```
logical TRUE  = TRUE
logical FALSE = TRUE
logical x     = FALSE
```

---

map f L *

liefert das Bild der Liste L unter der Abbildung f

```
map f []    = []
map f (a:x) = f a : map f x
```

---

member L a *

Test auf Vorkommen des Elements a in der Liste L

```
member L a = any (map (eq a) L)
```

----

minus m n

Subtraktion - primitiv

Alternative Schreibweise: m-n

----

mkset x

mkset eliminiert alle mehrfachen Vorkommen von Elementen in x

```
mkset []    = []
mkset (a:x) = a : mkset (filter (neq a) x)
```

----

neq x y *

Ungleichheitsoperator

```
neq x y = not (eq x y)
```

Alternative Schreibweise: x ~= y

----

not b *

logische Negation

```
not TRUE  = FALSE
not FALSE = TRUE
```

Alternative Schreibweise: ~b

----

number x

Typzugehörigkeitsprädikat - primitiv

----

or x y

Disjunktion; der zweite Parameter wird nur ausgewertet, wenn der erste FALSE ergibt

or x y = x → TRUE ; y

Alternative Schreibweise: x | y

---

plus m n

Addition - primitiv

Alternative Schreibweise: m+n

---

printwidth x

liefert die Länge der für die Ausgabe verwendeten Repräsentation des Objekts x in Zeichen - primitiv

---

product L

liefert das Produkt aller Elemente der Liste L

product L = foldr times 1 L

---

rem m n

Rest bei der ganzzahligen Division - primitiv

---

reverse L

liefert die Liste, die die Elemente von L in umgekehrter Reihenfolge enthält

reverse L = foldl cons []

---

rjustify n x

Formatierfunktion: fügt vor x soviele Blanks, daß x rechtsbündig innerhalb von insgesamt n Zeichen angeordnet ist

rjustify n x = spaces (n - printwidth x) : x

---

show x

Falls x keine Liste ist, ist show die Identität. Listen werden durch Einfügen von Trennsymbolen wie Klammern und Kommata für die Ausgabe aufbereitet - primitiv

---

sin x

trigonometrische Funktion - primitiv

---

spaces n *

liefert eine Liste aus n Blanks

```
spaces 0 = [ ]
spaces n = %  : spaces (n-1)
```

---

sqrt x

Wurzelfunktion - primitiv

---

sum L

liefert die Summe aller Elemente von L

```
sum L = foldr plus 0
```

---

take n L

liefert die Liste der ersten n Elemente von L

```
take n [ ]       = [ ]
take n (a : x) = n > 0 → a : take (n-1) x; [ ]
```

---

times m n

Multiplikation - primitiv

Alternative Schreibweise: m*n

---

tl x

liefert den Rest der Liste x nach Entfernen des ersten Elements, falls x schwach-terminierende Liste ist, kann tl x eine Nichtliste sein - primitiv

```
tl (a:x) = x
```

---

union x y

Mengenvereinigung

```
union x y = filter (not . member y) x ++ y
```

---

until f g x

g wird sooft auf x angewendet, bis das Ergebnis das Prädikat f erfüllt

```
until f g x = f x → x; until f g (g x)
```

---

while f g x

g wird solange auf x angewendet, wie das Ergebnis das Prädikat f erfüllt

```
while f g x = f x → while f g (g x); x
```

---

zip x

Matrixtransposition, x muß daher eine Liste von Listen sein

```
zip x = hd x = [] → []; map hd x : zip (map tl x)
```

Literatur:

/Boley85/
Boley, H.; Kammermeier, F. et al.:
LISPLOG: Momentaufnahmen einer LISP/PROLOG-Vereinheitlichung
Memo SEKI-85-03
Kaiserslautern, 1985

/Boutel84/
Boutel, B.E.:
Data Structure in a Graph Reduction Machine
ANZARP Report 84-004, Wellington

/Curry, Feys58/
Curry, H.B.; Feys, R.:
Combinatory Logic, Vol. I
North Holland, 1958

/Gordon79/
Gordon, M.J., Milner, A.J., Wadsworth, C.P.:
Edinburgh LCF
A Mechanized Logic of Computation
Lecture Notes in Computer Science, vol. 78
Berlin 1979

/Hughes82/
Hughes, R.J.M.:
Super Combinators: A New Implementation Method for Applicative Languages
Proc. ACM Symposium on Lisp and Functional Programming 198

/Richards84/
Richards, H.:
Programming in SASL
Burroughs Austin Research Center, ARC 84-21
Austin 1984

/Scheevel84/
Scheevel, M.:
The SASL Prelude
Burroughs Austin Research Center
Austin 1984

/Schönfinkel24/
Schönfinkel, M.:
Über die Bausteine der mathematischen Logik
in: Mathematische Annalen, Vol. 92, 1924, S.305-316

/Steele, Sussman78/
Steele, G.L.; Sussman,G.J.:
The Revised Report on SCHEME: A Dialect of LISP
Memo Nr. 452, AI-Lab, MIT, Cambridge 1978

/Steele84/
Steele, G.L.:
Common Lisp - The Language
Burlington 1984

/Stoye84/
Stoye, W.R., Clarke, T.J.W., Norman, A.C.:
Some Practical Methods for Rapid Combinator Reduction
Proc. ACM Symposium on Lisp and Functional Programming
Austin 1984

/Turner79a/
Turner, D.A.:
A New Implementation Technique for Applicative Languages
in: Software - Practice and Experience, Vol.9(1), 1979,
S.31-49

/Turner79b/
Turner, D.A.:
Another Algorithm for Bracket Abstraction
in: The Journal of Symbolic Logic, Vol. 44(2), 1979,
S.267-270

/Turner83/
Turner, D.A.:
SASL Language Manual (Revised Version)
Canterbury 1983

/Wadsworth71/
Wadsworth, C.P.:
Semantics and Pragmatics of the λ-calculus
DPhil Thesis
Oxford 1971

/White79/
White, J.L.:
NIL - A Perspective
Proceedings of the Macsyma Users Conference
Washington, 1979