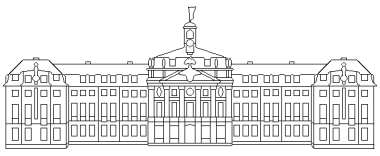
Übertragung der LCC – Optimierungstechniken auf das kellerartige Laufzeitsystem nach Honschopp

Diplomarbeit

vorgelegt von

Guido Wessendorf

August 1992



Westfälische
Wilhelms–Universität
Münster

Institut für numerische und instrumentelle Mathematik
I N F O R M A T I K

Herrn Professor Dr. W.–M. Lippe danke ich
für die Vergabe des Themas dieser Arbeit.

Inhaltsverzeichnis

Einleitung

In dieser Diplomarbeit werden statische Optimierungstechniken, ähnlich denen der Low Cost Call–Optimierung nach Felgentreu [Fe87], für die applikative Programmiersprache LISP/N [Li/Si79] und einem kellerartigen Laufzeitsystem [Ho/Li/Si83] vorgestellt.

Applikative Programmiersprachen, die auf dem λ–Kalkül basieren (wie z.B. LISP), haben inzwischen eine weite Verbreitung gefunden, insbesondere im Umfeld der „Künstlichen Intelligenz“. Im Gegensatz zu ALGOL60–artigen Sprachen, in denen bei Funktions–Aufrufen lediglich funktionale Argumente zugelassen sind, werden in LISP–artigen Sprachen sowohl funktionale Argumente als auch funktionale Ergebnisse unterstützt.

Dabei stellt sich immer wieder die Frage nach effizienten Implementierungs–Techniken für diese Sprachen. Aus diesem Grund wurde von Lippe und Simon die Sprache LISP/N [Li/Si79] entwickelt. Sie basiert auf dem Sprachumfang von pure LISP, bietet die bevorzugte statische Variablen–Bindung (static scoping) und beruht auf der der λ–Semantik genügenden Reduktionsstrategie *Call By Name* (das „N“ im Namen von LISP/N deutet dies an). Für die zur Zeit gebräuchliche Von Neumann–Rechnerarchitektur wurde für LISP/N eine effiziente Laufzeitkeller–Technik vorgeschlagen [Ho/Li/Si83] und Honschopp gab auf dieser Grundlage einen Compiler und ein Laufzeitsystem an [Ho83].

Ein maßgeblich neues Prinzip dieses Laufzeitsystems, nämlich die mögliche Speicherplatz–Freigabe bereits in der Situation „Funktions–Aufruf“ und nicht erst, wie sonst üblich, in der Situation „Funktions–Ende“, bietet einen großen Spielraum für weitreichende Optimierungsmethoden. Da LISP/N für ein Compiler–System vorgesehen ist (im Gegensatz zu vielen Interpreter–Systemen), kann gerade die Übersetzungs–Phase eines LISP/N–Programms für Analysen und Maßnahmen ausgenutzt werden, um dann zur kritischen Laufzeit sehr effizient optimieren zu können.

Wir werden die jeweiligen Optimierungs–Techniken in den nachfolgenden kurzen Kapitel–Beschreibungen ansprechen. Zur Dokumentation der erziehlten Verbesserungen wurden jeweils Speicherplatz– und Laufzeitmessungen durchgeführt. Dazu wurde das LISP/N–System von Honschopp [Ho83] nach Turbo–Pascal portiert und ist dann entsprechend modifiziert und erweitert worden.

Im folgenden beschreiben wir kurz den Inhalt der folgenden Kapitel:

Im *ersten Kapitel* werden die Grundlagen für die folgenden Kapitel geschaffen. Dabei wird im wesentlichen die Syntax und die Semantik von LISP/N definiert und viele, für das Verständnis der Arbeit notwendige Begriffe eingeführt.

Im *zweiten Kapitel* stellen wir das neue Laufzeitsystem aus der Diplomarbeit von Honschopp [Ho83] vor.

Im *dritten Kapitel* beschreiben wir die LCC–Optimierungstechniken für einen LISP–Interpreter mit statischer Variablen–Bindung und Shallow Binding von Felgentreu [Fe87] vor. Durch Standardisierung und Markierung des zu interpretierenden Programms wird bereits in der Einlese–Phase über eine dann zur Laufzeit effiziente Optimierung entschieden.

Im *vierten Kapitel* führen wir eine formale Untersuchung derjenigen Situationen durch, die im Honschopp–Laufzeitsystem zu einer frühzeitigen Speicherplatzfreigabe („HOpt“) führen, und dadurch die Klasse der nach Honschopp optimierten Aufrufe („HOpt–Klasse“) angeben. Trotz der rein dynamischen Natur der Optimierung kann für jeden Aufruf in einem „echten“ LISP/N–Programm statisch entschieden werden, ob dynamisch optimiert wird. Mit gewissen Auflagen an die Parameter eines Aufrufs können wir viele der aus der Literatur bereits bekannten Klassen von Funktionsaufrufen der HOpt–Klasse unterordnen. Den Abschluß dieses Kapitels bildet eine knappe Gegenüberstellung mit der LCC–Optimierung.

Im *fünften Kapitel* wird die nicht definitionsgemäß erfolgte Implementation des „GDV“–Verweises in [Ho83] aufgezeigt: Häufig werden unnötig viele neue GDV–Verweise eingeführt und dadurch oft sowohl eine HOpt verhindert als auch ein erhöhter Verwaltungsaufwand betrieben. Diese Ineffizienz werden wir beseitigen und so einen definitionsgemäßen GDV zurückgewinnen.

Im *sechsten Kapitel* wird, motiviert durch die Erkenntnisse und Techniken in [Fe87], eine statische Analysetechnik für den LISP/N–Compiler vorgestellt: Durch Untersuchung des „relevanten lokalen Kontext“ (welcher nicht identisch mit demjenigen in [Fe87] ist), und Einfügen entsprechender Markierungen in den Programmtext, kann dem Compiler mitgeteilt werden, wie er das Programm zu übersetzen hat. Das Laufzeitsystem hat dann unmittelbar die Informationen zur Verfügung, um zum einen den GDV–Verweis geziehlter anwenden zu können und zum anderen sogenannte „dicke Parameter“ effizienter zu handhaben.

Im *siebten Kapitel* stellen wir eine Methode vor, wie statisch darüber entschieden werden kann, ob durch eine andere Auswertungsreihenfolge der Parameter mehrstelliger Standardfunktionen eine Speicherplatz–Einsparung zur Laufzeit bewirkt werden kann. Die vorgestellten Entscheidungskriterien basieren wieder auf einer Untersuchung von relevanten lokalen Kontexten, und bei einer durchgeführten Parameter–Permutation ist dann sichergestellt, daß häufig optimiert, aber nie eine Verschlechterung eintreten kann.

Im *achten Kapitel* wird für die möglichen zusätzlichen Parametergruppen (pending Parameter) eines Aufrufs ein eigener Kellerspeicher eingerichtet. Der Verwaltungsaufwand für diese Parameter kann dadurch drastisch reduziert werden und es sind neben Speicherplatzeinsparungen auch Laufzeitgewinne möglich. Dadurch, daß eine Linkage erst *nach* einer möglichen Speicherplatzfreigabe in das zugehörige Activation Record geschrieben wird, kann weitere Laufzeit eingespart werden.

Im *neunten Kapitel* wird die Auswertungsstrategie Call By Name von LISP/N in den meisten Aufruf–Situationen durch die semantisch äquivalente, jedoch wesentlich effizientere Strategie Call By Need ersetzt. Die vorgestellte Realisierung beruht dabei auf sehr einfachen, dynamisch schnell durchzuführenden Kriterien und Maßnahmen. Die festgestellten Einsparungen sowohl an Speicherplatz als auch an Laufzeit verdeutlichen die Relevanz von Call By Need, zeigen aber andererseits auch, daß die Call By Name–Strategie nur zu theoretischen Zwecken dienen kann.

Im *zehnten Kapitel* werden in knapper Form Bemerkungen und Hinweise zur vorgenommenen Implementation gegeben und ein exemplarischer Beispiel–Lauf des neuen LISP/N–Systems kommentiert.

Wir schließen mit einem kurzen Ausblick auf mögliche Erweiterungen und altenative Konzepte. Der Anhang enthält einige Beispiel–Programme, Statistiken und die kompletten Pascal–Programme des neuen LISP/N–Systems.

Kapitel 1

Grundlagen

In diesem Kapitel führen wir die wesentlichen Grundlagen zur Syntax und zur Semantik von LISP/N ein.

1.1 Die Syntax von LISP/N

Die Syntax von LISP/N führen wir durch Angabe einer kontextfreiartigen Grammatik mit dem dazugehörigen Produktionensystem, welches diese Grammatik erzeugt, ein. Zunächst jedoch folgende

Definition 1.1 Wir führen folgende Mengenbezeichnungen ein:

- Die Menge *S–AUSD* von S–Ausdrücken (Konstanten), d.h. eine Menge von Zeichenfolgen, die durch unten definierte Produktionen P auf <S–Ausdruck> reduziert werden können.
- Die Menge *DL* der Begrenzungssymbole (delimitation):
  DL := { `BEGIN`, `END`, `IF`, `THEN`, `ELSE`, `FI`, `FUNC`, `MODE`, `VOID`, `S-EXPR`, `;` , `,` , `(` , `)` , `:` , `{` , `}` }.
- Die Menge *SFKT* der Standardfunktions–Identifikatoren, d.h.
  SFKT := { `ATOM`, `CAR`, `CDR`, `CONS`, `EQ` }.
- Die Menge *SIDF* der Standardidentifikatoren, d.h.
  SIDF := SFKT ∪ { `IN`, `SMODE1`, `SMODE2` }.
- Die Menge *NSIDF* der Nichtstandardidentifikatoren, d.h. diejenigen Zeichenfolgen, die sich mittels P auf <Nichtstandardidentifikator> reduzieren lassen.

Die Mengen S–AUSD, DL, SIDF und NSIDF seien paarweise disjunkt.

Bemerkung 1.1 Die S–Ausdrücke `T` und `F` sind spezielle S–Ausdrücke und stehen für die beiden Wahrheitswerte *true* und *false*. `NIL` ist ein spezielles Atom und bezeichnet die leere Liste „`()`“. Zur Vereinfachung der Listenschreibweise wird folgende Abkürzung eingeführt:

$$(s_1.(s_2.(s_3.\ \ldots\ .(s_n.\texttt{NIL})\ \ldots\ ))) = (s_1\ \ s_2\ \ s_3\ \ \ldots\ \ s_n),$$

wobei $n \geq 1$ und s_i mit $1 \leq i \leq n$ S–Ausdrücke seien.

Definition 1.2 Die kontextfreiartige Grammatik G von LISP/N ist durch das Quadrupel $G = (N, T, P, A)$ gegeben, wobei N, T, P und A wie folgt definiert sind:

- Die Menge N der Nichtterminalsymbole. Sie ergibt sich aus den links von „::=“ in P stehenden Symbolen,
- Die Menge T der Terminalsymbole, wobei T = S–AUSD ∪ SIDF ∪ NSIDF ∪ DL ist,
- A ist das Axiom <Programm> $\in N$ und
- P ist das Produktionensystem. Wir geben die Produktionen von P in der Backus–Naur–Form (BNF) an:

```
<Programm> ::= BEGIN
                  <Mode-Deklarationsteil>
                  <Funktions-Deklarationsteil>
                  <Hauptprogramm>
               END
<Mode-Deklarationsteil> ::= <leer> |
                            <Mode-Deklaration>; [<Mode-Deklaration>;]
<Mode-Deklaration> ::= MODE <Mode-Identifikator> = <strukturierter Mode>
<Mode-Identifikator> ::= <Nichtstandardidentifikator> | SMODE1 | SMODE2
<strukturierter Mode> ::= FUNC (<Mode-Liste>) <Ergebnis-Mode>
<Mode-Liste> ::= <leer> | <Mode> [,<Mode>]
<Ergebnis-Mode> ::= <Mode> | VOID
<Mode> ::= <Mode-Identifikator> | S-EXPR
<Funktions-Deklarationsteil> ::=
   <leer> | <Nichtstandardfunktion>; [<Nichtstandardfunktion>;]
<Nichtstandardfunktion> ::=
   <Mode-Identifikator>:
      <Funktionsidentifikator> (<formale Parameterliste>) <Ergebnis-Mode>;
         <Rumpf>
<Funktionsidentifikator> ::= <Nichtstandardidentifikator>
<formale Parameterliste> ::=
   <leer> | <Mode>: <formaler Parameter> [,<Mode>: <formaler Parameter>]
<formaler Parameter> ::= <Nichtstandardidentifikator>
<Rumpf> ::= { <Funktions-Deklarationsteil> <Anweisungsteil> }
<Anweisungsteil> ::= <Ausdruck>
<Ausdruck> ::= <leer> | <S-Ausdruck> | <Standardfunktion> |
               <Nichtstandardidentifikator> | <Applikation> | <Konditional>
<Applikation> ::=
   <Caller> (<aktuelle Parameterliste>) [(<pending Parameterliste>)]
<Caller> ::= <Standardfunktion> | <Nichtstandardidentifikator>
<aktuelle Parameterliste> ::= <Parameterliste>
<pending Parameterliste> ::= <Parameterliste>
<Parameterliste> ::= <leer> | <aktueller Parameter> [,<aktueller Parameter>]
<aktueller Parameter> ::= <Ausdruck> | IN[1]
```

[1] Nur im Hauptprogramm zulässig.

```
<Konditional> ::= IF <if-Teil> THEN <then-Teil> ELSE <else-Teil> FI
<if-Teil> ::= <Ausdruck>
<then-Teil> ::= <Ausdruck>
<else-Teil> ::= <Ausdruck>
<Standardfunktion> ::= ATOM | CAR | CDR | CONS | EQ
<Hauptprogramm> ::= <Anweisungsteil>
<S-Ausdruck> ::=  "<innerer S-Ausdruck> | T | F
<innerer S-Ausdruck> ::= <Atom> | <punktierte Liste> | <Liste> | NIL
<Atom> ::= <Buchstabe> [<Buchstabe oder Zahl>]
<punktierte Liste> ::= (<innerer S-Ausdruck> . <innerer S-Ausdruck>)
<Liste> ::= () | (<innerer S-Ausdruck> [_<innerer S-Ausdruck>])
<Nichtstandardidentifikator> ::= <Buchstabe> [<Buchstabe oder Zahl>]
<Buchstabe oder Zahl> ::= <Buchstabe> | <Zahl>
<Buchstabe> ::= A | ... | Z | a | ... | z
<Zahl> ::= 0 | ... | 9
<leer> ::= ε     (die leere Zeichenreihe)
```

Hinweise: Das Symbol „_“ bezeichnet einen syntaktisch signifikanten Zwischenraum (blank). Der Standardidentifikator IN ist nur im Hauptprogramm zulässig und liefert zur Laufzeit einen über die Standardeingabe (Tastatur) eingegebenen S–Ausdruck als Ergebnis.

Viele im Laufe dieser Arbeit verwendete Begriffe lassen sich mit dem Produktionensystem P erklären. Dazu folgende

Definition 1.3 Ein Wort $w \in T^*$, das sich mit Hilfe der Produktionen aus P auf ein Nichtterminalsymbol <...> $\in N$ reduzieren läßt, wird als „...“ bezeichnet.

Wir zerlegen ein Programm nun in „Grundsymbole“ und definieren das „Vorkommen“ von Grundsymbolen und syntaktischen Zeichenfolgen in einem Programm:

Definition 1.4

1. Ein *Grundsymbol* („token“) ist ein Wort $w \in T^*$, mit $w \in$ S–AUSD $\cup$ SIDF $\cup$ NSIDF $\cup$ DL.

2. Sei Π ein Programm, $\Pi = w_1 w_2 w_3 \dots w_n$, wobei die w_i mit $1 \leq i \leq n$ Grundsymbole maximaler Länge sind. Dann heißt das Paar (w_i, i) *Vorkommen von w_i an der Position i in* Π.
   (Durch die Forderung „maximaler Länge“ der Grundsymbole wird ausgeschlossen, daß ein Vorkommen eines Wortes in kleinere Teilvorkommen zerlegt werden kann. Beispiel: In dem Aufruf h(CARD) hat CARD maximale Länge, jedoch nicht das Grundsymbol CAR.)

3. Eine *syntaktische Zeichenfolge* S in einem Programm $\Pi = w_1 w_2 w_3 \dots w_n$ ist eine zusammenhängende Folge von Grundsymbolen aus Π, also $S = w_a \dots w_b$ mit $1 \leq a \leq b \leq n$.
   Wir sprechen auch vom Vorkommen (S, k) einer syntaktischen Zeichenfolge S in einem Programm Π. Dann bezeichnet k die Position des *ersten* Grundsymbols von S in Π.

Definition 1.5 Sei $\Pi = w_1 w_2 w_3 \ldots w_n$, mit $n \geq 1$, eine endliche Folge von Grundsymbolen, die sich mit Hilfe der Produktionen P zu <Programm> reduzieren läßt, dann heißt Π ein *syntaktisches Programm*.

Bemerkung 1.2 Es ist algorithmisch entscheidbar, ob Π ein syntaktisches Programm ist.

Wir vereinbaren folgende Redeweisen:

Definition 1.6 Applikationen werden, wie auch in anderen Programmiersprachen üblich, *Funktionsaufrufe* oder kurz *Aufrufe* genannt. Ist die aufgerufene Funktion eine Nichtstandardfunktion (kurz: *NSF*), so sprechen wir auch von einem *NSF–Aufruf*. Ist die aufgerufene Funktion eine Standardfunktion (kurz: *SF*), so sprechen wir auch von einem *SF–Aufruf*.

Definition 1.7 Gemäß dem Produktionensystem P sprechen wir allgemein von einem *aktuellen Parameter*, wenn wir die Argumente eines Aufrufs in der aktuellen–, oder im Falle eines NSF–Aufrufs auch der möglichen pending–Parameterliste(n), meinen.

Wenn wir den Aufruf c der NSF f mit aktuellen Parametern meinen, so schreiben wir anstatt $c = f(\ldots)(\ldots)^*$ (das „$*$“ bedeutet, daß endlich viele weitere pending Parameterlisten möglich sind), kurz $c = f(\ldots)$.

Wenn wir explizit einen aktuellen Parameter innerhalb einer pending Parameterliste meinen, so sprechen wir auch von einem „*pending Parameter*“ (kurz: *PP*).

Bemerkung 1.3 Ein syntaktisches Programm Π ist als eine alles umfassende NSF ohne Funktionsidentifikator und ohne formale Parameter anzusehen und zur Laufzeit immer die erste aufgerufene Funktion.

Definition 1.8 Ein aktueller Parameter eines NSF–Aufrufs, der eine Applikation oder ein Konditional ist, heißt *dicker Parameter*.

Definition 1.9 Sei (E, k) das Vorkommen einer syntaktischen Zeichenfolge E in einem Programm Π. Dann heißt (E, k) *linker Ast*, falls E in Π entweder

- innerhalb eines Aufrufs $f_s(a_1, a_2)$ mit $f_s \in \{CONS, EQ\}$ vorkommt oder
- innerhalb eines Konditionals IF a_1 THEN a_2 ELSE $a_{2'}$ FI vorkommt

und (E, k) das Vorkommen von a_1 ist. Falls (E, k) das Vorkommen von a_2 oder $a_{2'}$ ist, so heißt (E, k) *rechter Ast*.

Um einige der soeben eingeführten Begriffe zu verdeutlichen, wollen wir ein syntaktisches LISP/N–Programm Π als Beispiel betrachten:

Beispiel 1.1

```
BEGIN
  MODE m1 = FUNC (SMODE1) SMODE1;
  m1: twice (SMODE1: ff) SMODE1;
        { SMODE1: p (S-EXPR: x) S-EXPR; { ff(ff(x)) };
          p };
  SMODE1: g (S-EXPR: x) S-EXPR;
           { IF ATOM(CONS(x,"D)) THEN "NIL
                                 ELSE twice(CDR)(x)
             FI };
  g((A B C))
END
```

In diesem Beispiel sind dicke Parameter kursiv und linke Äste unterstrichen dargestellt:

Der Aufruf ff(x) ist ein dicker Parameter und in dem Konditional der NSF g kommen zwei linke Äste geschachtelt vor: Der äußere linke Ast ist ATOM(CONS(x,"D)), und der innere linke Ast ist x.

Der Aufruf c = twice(CDR)(x) der NSF *twice* erfolgt mit einer aktuellen Parameterliste „(CDR)“ und einer pending Parameterliste „(x)“. Gemäß Definition 1.7 hat der Aufruf c die aktuellen Parameter „CDR“ und „x“.

Bemerkung 1.4 Um die Darstellung von Beispiel–Programmen übersichtlicher und kürzer zu gestalten, werden im folgenden die Programme nicht in ihrer syntaktischen LISP/N–Form dargestellt, sondern eine dem λ–Kalkül ähnliche Notation verwendet. Das obige Beispiel würde dann wie folgt notiert:

```
twice = λ ff . { p = λ x . { ff(ff(x)) }
                   p }
g = λ x . { IF ATOM(CONS(x,D)) THEN NIL
                                ELSE twice(CDR)(x)
            FI }
g((A B C))
```

Bei der Notation werden folgende Konventionen eingehalten:

- Nur Nichtstandardidentifikatoren werden in Kleinbuchstaben dargestellt.
- Einem Atom wird nicht mehr „"“ vorangestellt, sondern es wird groß geschrieben.
- Rümpfe werden in „{“ und „}“ eingeschlossen.

Es folgen einige grundlegende und häufig benutzte Bezeichnungen (siehe auch [La73]):

Definition 1.10 Der *statische Vorgänger* $SV(f)$ einer NSF f ist die kleinste f echt umfassende NSF oder gemäß Bemerkung 1.3 das Programm Π selbst.

Definition 1.11 Das *statische Niveau* $SN(f)$ einer NSF f ist induktiv definiert durch

$$SN(f) := \begin{cases} 1 & \text{, falls } SV(f) = \Pi \\ SN(SV(f)) + 1 & \text{, sonst.} \end{cases}$$

Das Programm Π habe das statische Niveau 0. Also SN(Π):=0.

Definition 1.12 Das *dynamische Niveau* $DN(c)$ eines Aufrufs $c = f(\ldots)$ einer NSF f ist die Anfangsadresse des für c angelegten Activation Records (kurz: AR) im Activation Record–Keller (siehe Kapitel 2). Ist die durch c aufgerufene NSF f bekannt, so schreiben wir auch DN(f).

Definition 1.13 Wir sprechen vom *dynamischen Niveau des dynamischen Vorgängers* (kurz: DN(DV(c))) eines Aufrufs $c = f(\ldots)$ einer NSF f, wenn wir die Anfangsadresse des unmittelbar vor dem AR_c angelegten AR im AR–Keller meinen.

Definition 1.14 Sei $c = f(\ldots)$ der Aufruf der NSF f, und sei c' ein Aufruf vom SV(f). Wir sprechen von *einem dynamischen Niveau des statischen Vorgängers* DN(SV(c)) von c, wenn wir das zum Zeitpunkt des Aufrufs c gültige DN(c') meinen. Ist die durch c gerufene NSF f bekannt, so schreiben wir auch DN(SV(f)).

Bemerkung 1.5 *Ein* dynamisches Niveau des statischen Vorgängers bedeutet nicht unbedingt das zum letzten Aufruf des statischen Vorgängers gehörige AR, d.h. es können in der Zwischenzeit weitere Aufrufe vom SV erfolgt sein, auf die dann aber gemäß der statischen Verweiskette (siehe [La73]) nicht verwiesen werden darf. Die „most recent“–Eigenschaft vom SV ist also nicht zwingend.

Im folgenden interessieren wir uns für Vorkommen von Identifikatoren. Dabei unterscheiden wir zwischen angewandten und deklarierenden Vorkommen:

Definition 1.15 Das Vorkommen (I, k) eines Identifikators $I \in$ NSIDF in einem syntaktischen Programm Π heißt *deklarierend*, falls sich I mit Hilfe der Produktionen von P auf

1. <Mode Identifikator>,
2. <Funktionsidentifikator> oder
3. <formaler Parameter>

reduzieren läßt. Alle übrigen Vorkommen von Identifikatoren in Π, die nicht deklarierende Vorkommen sind, heißen *angewandte Vorkommen*.

Zu jedem angewandtem Vorkommen eines Nichtstandardidentifikators in einem Programm existiert eine eindeutig bestimmte Umgebung (Enviroment), in der alle „sichtbaren“ deklarierenden Vorkommen liegen. Die eindeutige Zuordnung eines angewandten Vorkommens zu einem „zugehörigen deklarierenden Vorkommen“ wird durch eine Bindungsrelation erreicht ([Ki87]) und führt uns zu der Klasse der „gebundenen“ Programme:

Definition 1.16 Sei Z eine Nichtstandardfunktion in einem syntaktischen Programm Π, d.h. $Z \equiv Z_1{:}Z_2(Z_P)Z_3;Z_R$ wobei Z_1 der Mode Identifikator, Z_2 der Funktionsidentifikator, Z_P die formale Parameterliste, Z_3 der Ergebnis–Mode und Z_R der Rumpf ist. Dann heißt das Paar (Z_P, Z_R) *erweiterter Funktionsrumpf* von Z.

Definition 1.17 Entsprechend Definition 1.4 und der Situation in Definition 1.16 sei $((Z_P, Z_R), k)$ das *Vorkommen des erweiterten Funktionsrumpfes* (Z_P, Z_R).

Definition 1.18 Die zugeordnete *Reichweite range(I, k)* eines deklarierenden Vorkommens (I, k) in einem syntaktischem Programm Π ist das kleinste (I, k) echt umfassende Vorkommen eines erweiterten Funktionsrumpfes, bzw. das Vorkommen von Π selbst.

Definition 1.19 Der *Gültigkeitsbereich scope(I, k)* eines deklarierenden Vorkommens (I, k) ist die zugeordnete Reichweite range(I, k) mit Ausnahme echt innerer Reichweiten range(I, k'), welche einem deklarierendem Vorkommen (I, k') mit gleichem Identifikator I zugeordnet sind.

Damit können wir jetzt die wichtige Zuordnung von angewandten zu definierenden Vorkommen festlegen:

Definition 1.20 Sei VNSI(Π) die Menge aller Vorkommen von Nichtstandardidentifikatoren in einem syntaktischem Programm Π. Dann führen wir eine *Bindungsrelation* δ_Π ein:

$$\delta_\Pi \subseteq \text{VNSI}(\Pi) \times \text{VNSI}(\Pi)$$

und $((I, k), (I', k')) \in \delta_\Pi \Longleftrightarrow$ die beiden folgenden Bedingungen sind erfüllt:

1. $I \equiv I'$ und
2. (a) (I, k) ist deklarierend und $k = k'$, oder
   (b) (I, k) ist angewandt und $(I, k) \in \text{scope}(I', k')$.

Definition 1.21 Ein syntaktisches Programm Π heißt *gebunden*, falls δ_Π auf VNSI(Π) linkstotal und rechtseindeutig ist.

Bemerkung 1.6

1. Es ist algorithmisch entscheidbar, ob ein syntaktisches Programm gebunden ist.
2. Die Linkstotalität sichert die Existenz eines deklarierenden Vorkommen zu einem angewandtem Vorkommen.
3. Die Rechtseindeutigkeit verhindert Mehrfach–Deklarationen in einem Gültigkeitsbereich.
4. Standardidentifikatoren dürfen nicht deklarierend vorkommen — Ein sogenanntes „Overloading“ ist in LISP/N somit nicht erlaubt.

Zusammen mit der Bindungsrelation führen wir weitere Redeweisen ein:

Definition 1.22 Der *statische Vorgänger* $SV(I,k)$ *eines deklarierenden Vorkommens* (I,k) eines Nichtstandardidentifikators I ist

- die NSF f, falls (I,k) in der formalen Parameterliste der NSF f vorkommt oder
- der SV(f), falls (I,k) das Vorkommen des Funktionsidentifikators der NSF f ist.

Definition 1.23 Der *statische Vorgänger* $SV(I,k)$ *eines angewandten Vorkommens* (I,k) eines Nichtstandardidentifikators I ist der statische Vorgänger des gemäß Bindungsrelation δ_Π zugehörigen deklarierenden Vorkommens (I,k') zu (I,k), d.h. $\mathrm{SV}(I,k) := \mathrm{SV}(I,k')$.

Definition 1.24 Das *statische Niveau* $SN(I,k)$ eines beliebigen Vorkommens (I,k) eines Nichtstandardidentifikators I ist $\mathrm{SN}(\mathrm{SV}(I,k)) + 1$.

Definition 1.25 Sei Π ein gebundenes Programm und (I,k) ein angewandtes Vorkommen des Identifikators I in Π. Sei (I',k') das gemäß Bindungsrelation δ_Π eindeutig zugeordnete deklarierende Vorkommen zu (I,k) in Π. Dann sprechen wir beim Vorkommen (I,k) eines Identifikators I entweder vom

- Vorkommen eines *formalen Identifikators*, falls sich I' in (I',k') mittels P zu <formaler Parameter> reduzieren läßt, oder vom
- Vorkommen eines *gewöhnlichen Identifikators*, falls sich I' in (I',k') mittels P zu <Funktionsidentifikator> reduzieren läßt.

Definition 1.26 Sei Π ein gebundenes Programm und f eine NSF in Π. Sei (I,k) das Vorkommen eines angewandten Nichtstandardidentifikators I in f. Falls das gemäß Bindungsrelation δ_Π zugehörige deklarierende Vorkommen (I,k') von (I,k) außerhalb von f vorkommt, so ist (I,k) *frei* in f.

Die Menge aller freien Nichtstandardidentifikator–Vorkommen in f wird mit *frei(f)* bezeichnet.

Bemerkung 1.7 Falls Π ein gebundenes Programm ist, so gilt aufgrund der Definitionen 1.21 und 1.26: $\text{frei}(\Pi) = \emptyset$.

Im folgenden wollen wir kurz festhalten, was unter einem „übersetzbarem“ Programm zu verstehen ist. (Für detaillierte Informationen und Beispiele sei dabei auf [Li/Si79] und [Ho83] verwiesen.):

Da LISP/N eine voll getypte Sprache ist, werden jeder Nichtstandardfunktion, jedem formalen Parameter und dem Ergebnis–Wert jeder Nichtstandardfunktion sogenannte „Modes“ zugeordnet. Modes können im Mode–Deklarationsteil definiert werden, oder es werden die drei Standardmodes S–EXPR, SMODE1 oder SMODE2 benutzt. Das Hauptprogramm hat als Ergebnis–Mode immer S–EXPR oder VOID (die leere Zeichenreihe).

Der Compiler überprüft beim Übersetzen eines Programms die Konsistenz der Mode–Deklarationen:

Definition 1.27 Ein gebundenes Programm Π ist genau dann *übersetzbar (echt)*, wenn die Modes

- der durch δ_Π aneinander gebundenen Identifikatoren,
- der aktuellen Parameter und der zugehörigen formalen Parameter, sowie
- der Anweisungsteile und dem Ergebnis–Mode von Nichtstandardfuntionen

gleich, d.h. textlich identisch sind.

Bemerkung 1.8 Es ist algorithmisch entscheidbar, ob ein gebundenes Programm echt ist.

1.2 Die Semantik von LISP/N

Die im vorherigen Abschnitt beschriebene Syntax von LISP/N bezieht sich auf das syntaktische und somit statische Erscheinungsbild eines Programms. Wenn aber das dynamische Verhalten des Programms, d.h. die Bedeutung zur Laufzeit gemeint ist, so sprechen wir von der *Semantik* eines Programms.

Um die Semantik eines LISP/N–Programms zu beschreiben, gehen wir in zwei Schritten vor: Zunächst wird die Semantik der fünf LISP/N–Standardfunktionen SF mit SF$\in$SFKT eingeführt. Danach wird durch Angabe einer „Kopierregel“ die Semantik von Programmen *mit* vom Programmierer definierten NSF auf die Semantik von Programmen *ohne* NSF zurückgeführt. Die LISP/N–Kopierregel ist eine Call By Name–Kopierregel und wurde aus

der ALGOL60–Kopierregel entwickelt [Li/Si79]. Sie stellt eine rein syntaktische Programmtransformations–Regel dar und ist vergleichbar mit der β– und α–Reduktion vom λ–Kalkül.

Zunächst also die Semantik der fünf Standardfunktionen (Sprachumfang von Pure–LISP) von LISP/N:

Definition 1.28 Seien s, s_1 und s_2 S–Ausdrücke. Das Symbol $\perp$ („bottom“) stehe für undefiniert. Dann werden die fünf Standardfunktionen von LISP/N durch folgende Übergangsfunktionen definiert:

$$[\![\mathrm{ATOM}(s)]\!] := \begin{cases} \mathtt{T} & \text{falls } s \text{ atomar ist} \\ \mathtt{F} & \text{sonst} \end{cases}$$

$$[\![\mathrm{CAR}(s)]\!] := \begin{cases} s_1 & \text{falls } s = (s_1.s_2) \\ \perp & \text{sonst} \end{cases}$$

$$[\![\mathrm{CDR}(s)]\!] := \begin{cases} s_2 & \text{falls } s = (s_1.s_2) \\ \perp & \text{sonst} \end{cases}$$

$$[\![\mathrm{CONS}(s_1, s_2)]\!] := (s_1.s_2)$$

$$[\![\mathrm{EQ}(s_1, s_2)]\!] := \begin{cases} \mathtt{T} & \text{falls } s_1 \text{ und } s_2 \text{ atomar und } s_1 = s_2 \\ \mathtt{F} & \text{falls } s_1 \text{ und } s_2 \text{ atomar und } s_1 \neq s_2 \\ \perp & \text{sonst} \end{cases}$$

Bemerkung 1.9 SF–Aufrufe werden gemäß der Auswertungsstrategie „Call By Value“ ausgewertet, da aufgrund der fest vorgegebenen Übergangsfunktionen (siehe Definition 1.28) lediglich die Argumente eines SF–Aufrufs ausgewertet werden müssen.

Da durch eine Kopierregel (sukzessive) Transformationen an syntaktischen Programmen durchgeführt werden sollen und ein Programm im allgemeinen bereits nach der ersten Anwendnung einer Kopierregel kein syntaktisches Programm mehr ist, muß aus formalen Gründen die Syntax von LISP/N geringfügig erweitert werden:

Definition 1.29 Sei $G^e = (N, T, P^e, A)$ die *erweiterte LISP/N–Grammatik*. Dabei entstehe P^e aus P, indem die Produktion

```
<Hauptprogramm> ::= <Anweisungsteil>
```

ersetzt wird durch

```
<Hauptprogramm> ::= <Anweisungsteil> | <generierter Anweisungsteil>
<generierter Anweisungsteil> ::= { [{] <generierter Ausdruck> [}] }
<generierter Ausdruck> ::= <Ausdruck>
```

Um Namenskonflikte während der (sukzessiven) Anwendung der Kopierregel zu vermeiden, führen wir zunächst den Begriff der „zulässigen Umbenennung“ von Nichtstandardidentifikatoren ein:

Definition 1.30 Sei $\Pi = w_1 w_2 w_3 \dots w_n mitn \geq 1$ ein gebundenes LISP/N–Programm. Wir sagen: Π geht durch *zulässige Umbenennung* in $\tilde{\Pi} = \tilde{w}_1 \tilde{w}_2 \tilde{w}_3 \dots \tilde{w}_n$ über, falls für alle $w_i \in$ NSIDF, die in $\tilde{w}_i \in$ NSIDF umbenannt worden sind, gilt:

- (w_i, i') sei das gemäß δ_Π zu (w_i, i) zugehörige deklarierende Vorkommen von w_i, d.h. $((w_i, i), (w_i, i')) \in \delta_\Pi$,
- $(\tilde{w}_i, i'')$ sei das gemäß $\delta_{\tilde{\Pi}}$ zu $(\tilde{w}_i, i)$ zugehörige deklarierende Vorkommen von $\tilde{w}_i$, d.h. $((\tilde{w}_i, i), (\tilde{w}_i, i'')) \in \delta_{\tilde{\Pi}}$

$\Longrightarrow i'' = i'$.

Auf dieser Grundlage können wir jetzt die Kopierregel angeben:

Definition 1.31 (Kopierregel) Sei Π ein übersetzbares LISP/N–Programm und sei $c = f(a_1^1, \dots, a_{n_1}^1) \dots (a_1^k, \dots, a_{n_k}^k)$ ein Aufruf der NSF f im Hauptprogramm von Π mit k Parameterlisten, wobei $k \geq 1$ und $n_i \geq 0$ für $1 \leq i \leq k$, und c komme nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor. Die zugehörige Funktionsdeklaration zu f sei von der Form:

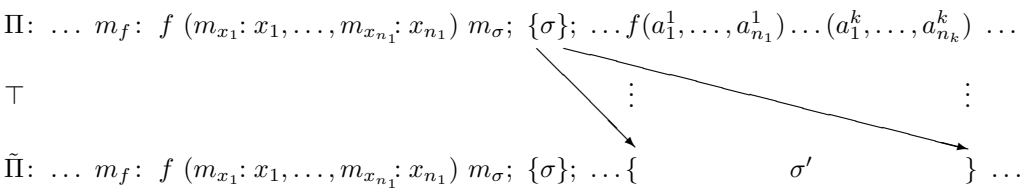
$$m_f\colon\ f\ (m_{x_1}\colon x_1, \dots, m_{x_{n_1}}\colon x_{n_1})\ m_\sigma;\ \{\sigma\};$$

wobei m_f der Mode von f, m_{x_i} der Mode des i–ten formalen Parameters, m_σ der Ergebnis–Mode und σ der Rumpf der NSF f sei.
Dann wird der Aufruf c ersetzt durch den generierten Ausdruck $\{\sigma'\}$, wobei σ' aus dem Rumpf σ durch folgende Modifikationen hervorgeht:

- Lokale Identifikatoren in σ' werden in solche zulässig umbenannt, die in Π nicht vorkommen.
- Für $i = 1, \dots, n_1$ wird jedes Vorkommen der formalen Parameter x_i in σ durch den zugehörigen aktuellen Parameter a_i^1 ersetzt.
- Falls $k > 1$ ist, werden alle NSF–Aufrufe $f'(b_1^1, \dots, b_{m_1}^1) \dots (b_1^l, \dots, b_{m_l}^l)$ sowie alle Funktionsidentifikatoren f'' in σ, mit Ausnahme derjenigen,
  - deren zugehörige Funktionsdeklarationen zu σ lokal sind,
  - die innerhalb von if–Teilen vorkommen, oder
  - die innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen,

  durch $f'(b_1^1, \dots, b_{m_1}^1) \dots (b_1^l, \dots, b_{m_l}^l)(a_1^2, \dots, a_{n_2}^2) \dots (a_1^k, \dots, a_{n_k}^k)$, bzw. durch $f''(a_1^2, \dots, a_{n_2}^2) \dots (a_1^k, \dots, a_{n_k}^k)$ ersetzt.

Zur Veranschaulichung soll folgendes Ersetzungs–Schema dienen:



Bemerkung 1.10 Durch die Angabe der erweiterten LISP/N–Grammatik kann analog zu den Definitionen 1.5 bzw. 1.21 erklärt werden, wann ein übersetzbares Programm nach (mehrmaliger) Anwendung der Kopierregel wieder syntaktisch bzw. gebunden ist. Dann kann gezeigt werden (siehe [Li/Si79]), daß durch (mehrmalige) Anwendung der Kopierregel ein echtes Programm echt bleibt und dadurch insbesondere zur Laufzeit keine weiteren Kontrollen nötig sind (z.B. über die korrekte Anzahl der aktuellen Parameter eines Aufrufs).

Nachdem nun die wichtigsten syntaktischen und semantischen Grundlagen für die funktionale Programmiersprache LISP/N aufgeführt wurden, stellen wir im nächsten Kapitel das neue Laufzeitsystem aus [Ho83] vor.

Kapitel 2

Das Laufzeitsystem nach U. Honschopp

Ein Laufzeitsystem ist nötig, weil ein Übersetzer (Compiler) lediglich eine statische Transformation von einem Programmtext (in unserem Fall: ein LISP/N–Programm) in eine semantisch äquivalente, maschinennähere Sprache (Code) vornimmt. Somit verfügt ein Compiler nicht über die dynamisch, also erst zur Laufzeit anfallenden Eingabedaten, wie dies bei einem Interpreter der Fall ist. Da der Ablauf eines Programms aber stark von den Eingabedaten abhängen kann und die Allokation von Speicherplatz erst dynamisch vorgenommen wird, kommt der erzeugte Code ohne ein Laufzeitsystem nicht aus.

Erreicht der Compiler beispielsweise einen formalen NSF–Aufruf ohne aktuelle Parameterliste (siehe Beispiel 2.1), so wird an dieser Stelle lediglich ein Sprung in das Laufzeitsystem (LZS) mit den statisch verfügbaren Informationen erzeugt. Um welche Informationen es sich dabei handelt, werden wir im nächsten Abschnitt noch sehen. Das Laufzeitsystem ermittelt dann zur Laufzeit die gerufene Funktion und fügt die zuletzt abgespeicherte pending Parameterliste als aktuelle Parameterliste hinzu, um zu einem vollständigen Aufruf zu gelangen.

Zwischen den Situationen NSF–Aufruf und NSF–Ende, d.h. bei der Auswertung eines Anweisungsteils, muß dann der vom Compiler erzeugte Code auf die jeweils gültigen Werte von Variablen (formalen Identifikatoren) zurückgreifen können. Allgemein kann man sagen, daß das Laufzeitsystem dafür Sorge tragen muß, daß die gemäß der Kopierregel vorgegebene Semantik in die Tat umgesetzt wird. Ist die Behandlung eines Laufzeitsystem–Aufrufs abgeschlossen, so wird an der Stelle im erzeugten Code fortgefahren, wo der Sprung ins Laufzeitsystem erfolgte.

In ALGOL60–artigen Sprachen ist es üblich, die Organisation des Laufzeitspeichers mit Hilfe einer Display–Technik in einem reinen Kellermechanismus zu steuern („Deletion–Strategie“) [Gr/Hi/La67]. Im Gegensatz zu

ALGOL60–artigen Sprachen, sind in *funktionalen*, bzw. *applikativen* Sprachen (wie z.B. LISP) folgende Konstruktionen erlaubt:

Beispiel 2.1

```
g = λ y . { y }
f = λ x . { x }
f(g)(A)
```

Der Aufruf der NSF f erfolgt mit der aktuellen Parameterliste „(g)“ und einer pending Parameterliste „(A)“. In f wird als Ergebnis lediglich der Funktionsidentifikator der NSF g *ohne* aktuelle Parameterliste geliefert. Das LZS vervollständigt diesen Funktionsausdruck mit der Parameterliste (A), und die anschließende Auswertung des Aufrufs g((A)) liefert dann das Atom A als Endergebnis.

Um zur Unterstützung solcher „höheren Funktionale“ nicht auf „Retention–Strategien“, verbunden mit einer aufwendigen Heap–Verwaltung (und der dann häufig notwendigen „Garbage Collection“), zurückgreifen zu müssen, wurde von W.-M. Lippe und F. Simon die applikative Programmiersprache LISP/N für ein *kellerartiges* Laufzeitsystem vorgestellt [Li/Si79], [Ho/Li/Si83]. Durch neuartige Verweise in den Laufzeitkeller, verbunden mit einer neuen Auffasung darüber, *wann* der Speicher für die Abarbeitung eines Aufrufs freigegeben werden kann, ist ein effizientes System zur Unterstützung höherer Funktionale entstanden. Auf dieser Grundlage gab Honschopp eine Implementation von Compiler und Laufzeitsystem für LISP/N an [Ho83].

In den folgenden Abschnitten stellen wir dieses neue Laufzeitsystem vor (in enger Anlehnung an [Ho83]). Dabei betrachten wir zunächst den generellen Aufbau eines Activation Record:

2.1 Der Aufbau eines Activation Record

Die Informationen, die bei der Ausführung einer NSF benötigt werden, werden durch einen zusammenhängenden Speicherbereich verwaltet, der *Activation Record* (kurz: AR), Display oder Festspeicherblock genannt wird.

Jedesmal wenn eine NSF aufgerufen wird, werden hier Speicherzellen für die aktuelle– und evtl. pending–Parameterliste(n), sowie für Hilfsvariablen reserviert. (Hilfsvariablen nehmen die Zwischenergebnisse der Auswertung der ersten $n-1$ Parameter einer n–stelligen SF auf.)

Es wird jedoch nicht nur Platz für die lokalen Identifikatoren der NSF reserviert, sondern es muß auch für organisatorische Zwecke Speicher zur Verfügung gestellt werden. Dieser Teil eines AR wird „Linkage“ genannt.

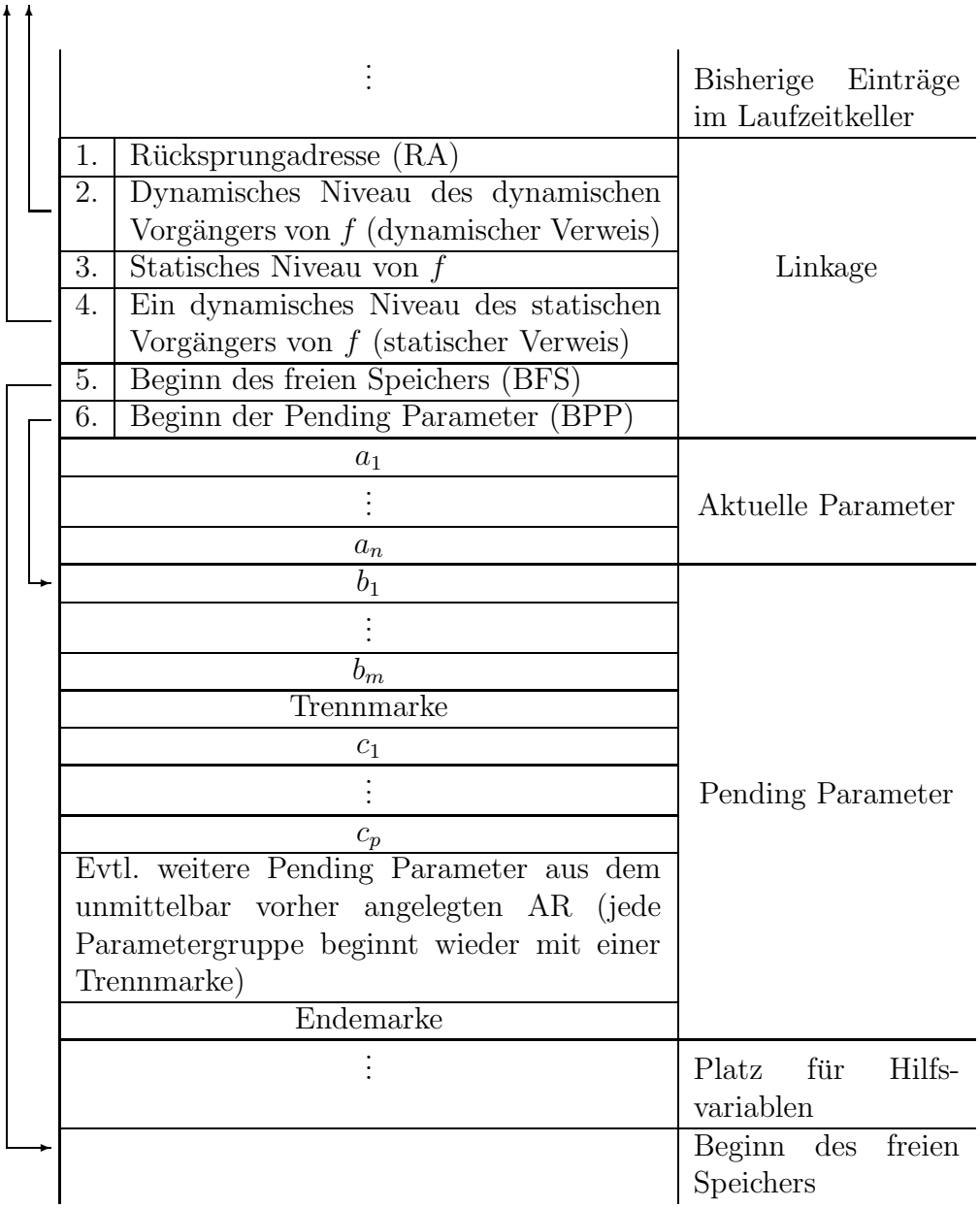
Ist ein Funktionsaufruf abgearbeitet, so kann der Speicherplatz des betreffenden AR wieder freigegeben werden.

Betrachten wir nun folgenden NSF–Aufruf c mit einer aktuellen– und zwei pending–Parameterlisten:

$$c \;=\; f\;(a_1,\ldots,a_n)\;(b_1,\ldots,b_m)\;(c_1,\ldots,c_p) \qquad \text{mit } n,m,p \geq 0$$

Er führt zum Eintrag des in Abbildung 2.1 angegebenen AR an die Spitze des Laufzeitkellers.

Abbildung 2.1: Aufbau eines Activation Records



2.1.1 Bemerkungen zu den einzelnen Einträgen im Activation Record

- Die Linkage besteht aus sechs Einträgen:

  1. Die Rücksprungadresse (kurz: RA) verweist auf die Adresse im Code–Speicher, an der die Verarbeitung nach Beendigung des Aufrufs fortgesetzt werden soll.
  2. Ein Verweis auf den Anfang des unmittelbar zuvor angelegten AR (siehe Definition 1.13).
  3. Das statische Niveau gibt die Schachtelungstiefe der NSF an, in der die Funktion definiert wurde. Es ist zum Ansteuern und Umladen der Indexregister notwendig (siehe Definition 1.11).
  4. Ein Verweis auf den Anfang des AR vom zum Zeitpunkt des Aufrufs gültigen statischen Vorgänger (siehe Definitionen 1.14 und Bemerkung 1.5).
  5. Ein Verweis direkt hinter das Ende des jeweiligen AR (dient zum schnellen ermitteln der ersten freien Speicherzelle für ein neu anzulegendes AR).
  6. Ein Verweis an den Anfang der pending Parameter (da die Anzahl der aktuellen Parameter nicht explizit im Keller gespeichert wird und der Suchvorgang nach einer möglichen Trennmarke ineffektiv wäre, erfolgt der schnelle Zugriff über diesen Verweis).

- Im Anschluß an die Linkage wird der Parameterblock angelegt. Zunächst die aktuellen Werte der formalen Parameter $a_1, \ldots, a_n$ und anschließend die einzelnen pending Parameterlisten (hier $b_1, \ldots, b_m$ und $c_1, \ldots, c_p$), jeweils durch eine Trennmarke getrennt. Falls vorhanden, so werden die pending Parameter aus dem unmittelbar zuvor angelegtem AR an das neu anzulegende AR angehängt.

  Bei der betrachteten Implemetierung [Ho83] werden je Parameter die in Tabelle 2.1 dargestellten Informationen eingetragen:

Tabelle 2.1: Eintrag für jeden Parameter in einem Activation Record

Typ des Parameters	Typ-Nr.	Adresse	statischer Verweis
Konstante (S-Ausdruck)	0	Heap-Adresse	statisches Niveau[1]
Funktionsidentifikator	1	Startadresse	ein dyn. Niveau des stat. Vorgäng.[3]
Funktionsidentifikator für einen „dicken Parameter“[2]	2	Startadresse	ein dyn. Niveau des stat. Vorgäng.[3]
Formaler Identifikator	(3)	aktueller Wert[4]	

Hinweise zu Tabelle 2.1:

(1) Das statische Niveau konstanter S–Ausdrücke ist unnötig. Der Einheitlichkeit halber ist dieser Eintrag vorhanden und kann mit 0 belegt werden.

(2) Dicke Parameter müssen als Parametertyp gesondert gehandhabt werden, damit alle ggf. hinzukopierten pending Parameter vom dynamischen Vorgänger auch solche bleiben und nicht zu vermeintlich aktuellen Parametern werden, wie dies bei Typ 1–Parametern, d.h. Funktionsausdrücken, der Fall wäre. (In Abschnitt 2.2 wird die Handhabung dicker Parameter verdeutlicht.)

(3) Dieser Verweis entspricht dem Verweis in der Linkage (siehe Abbildung 2.1) und zeigt auf „ein dynamisches Niveau des statischen Vorgängers“ vom jeweiligen funktionalen Parameter.

(4) Formale Identifikatoren werden zunächst als Typ–3 Parameter übergeben. Das Laufzeitsystem ermittelt dann unmittelbar den aktuellen Wert aus der dem formalen Identifikator entsprechenden Zelle des Laufzeitkellers und dieser wird dann kopiert. D.h., es wird immer nur ein Parameter vom Typ 0, 1 oder 2 in eine AR–Zelle eingetragen!

- Das Ende vom Parameterblock markiert eine Endemarke. Dahinter ist Platz für eventuell anfallende Hilfsvariablen. Hilfsvariablen können nur bei der Ausführung mehrstelliger Standardfunktionen anfallen. In LISP/N kommen deshalb nur die zweistelligen Standardfunktionen CONS und EQ in Frage. Die Hilfsvariablen werden dann zur Zwischenspeicherung des ausgewerteten ersten Arguments gebraucht.

2.2 Die Handhabung dicker Parameter

Ein dicker Parameter ist gemäß Definition 1.8 eine Applikation oder ein Konditional als aktueller Parameter eines NSF–Aufrufs. Damit solch ein umfangreicher Ausdruck gemäß der Call By Name–Strategie unausgewertet, d.h. textuell an die gerufene Funktion übergeben werden kann, wird ein dicker Parameter vom Compiler so übersetzt, als komme er nicht innerhalb des betreffenden aktuellen Parameters, sondern als Anweisungsteil einer vom Compiler eingeführten NSF vor:

Sei δ ein dicker Parameter als aktueller Parameter eines Aufrufs c der NSF f im Anweisungsteil einer NSF g (wobei g auch das Hauptprogramm sein kann), also

$$\mathtt{g} = \lambda \ldots . \quad \{ \ldots \underbrace{\mathtt{f}(\ldots,\delta,\ldots)}_{c} \ldots \}$$

Dann wird c vom Compiler übersetzt wie der Aufruf

$$c' = \texttt{f(...,h(),...)}$$

mit dem Aufruf der parameterlosen Nichtstandard–„Hilfsfunktion“ h anstatt von δ als aktuellen Parameter. Im Anweisungsteil der NSF h wird dann der dicke Parameter δ ausgeführt, d.h.

$$\texttt{h} = \lambda \ . \ \{ \ \delta \ \}$$

Dabei wird die NSF h so übersetzt, als komme sie im Funktions–Deklarationsteil der kleinsten den Aufruf c umfassenden NSF g deklarierend vor — die NSF g wird somit zum statischen Vorgänger der Hilfsfunktion h. Auf diese Weise wird sichergestellt, daß zur Laufzeit alle Informationen für eine mögliche Auswertung des dicken Parameters noch im AR–Keller vorhanden sind: Die Typ–2 Parameterzelle für einen dicken Parameter (siehe Tabelle 2.1) enthält einen Verweis auf einen statischen Vorgänger der zugehörigen Hilfsfunktion, d.h. in obiger Situation auf das AR_g, welches dann nicht frühzeitig freigegeben werden kann. Wir erhalten somit insgesamt:

$$\begin{array}{l} \texttt{g} = \lambda \ \ldots \ . \\ \qquad \{ \ \texttt{h} = \lambda \ . \ \{ \ \delta \ \} \\ \qquad\quad \ldots \ \underbrace{\texttt{f(...,h(),...)}}_{c'} \ \ldots \ \} \end{array}$$

Bemerkung 2.1 In Abschnitt 6.2 werden wir sehen, daß die Wahl der *kleinsten* das Vorkommen eines dicken Parameters umfassenden NSF zum statischen Vorgänger für die notwendige Hilfsfunktion eine häufig erhebliche Innefizienz bedeutet.

Es besteht somit ein Unterschied zwischen einem echten, d.h. gemäß Definition 1.27 übersetzbaren LISP/N–Programm Π, welches vom Programmierer erstellt wurde, und einem vom Compiler *übersetzten* Programm Π'. Dazu folgende

Definition 2.1 Sei Π ein echtes LISP/N–Programm. Wir sprechen von dem *übersetzten* LISP/N–Programm Π', falls Π gemäß der Handhabung dicker Parameter (s.o.) modifiziert wurde.

Bemerkung 2.2 In einem übersetzten LISP/N–Programm können somit nur noch die Applikationen $h()$ der von vom Compiler eingeführten Hilfsfunktionen h als aktuelle Parameter von NSF–Aufrufen vorkommen. Die vom Programmierer vorgegebenen NSF–Aufrufe $c = f(\ldots)$ innerhalb aktueller Parameter von NSF–Aufrufen sind in den Anweisungsteilen der generierten Hilfsfunktionen h verlegt.

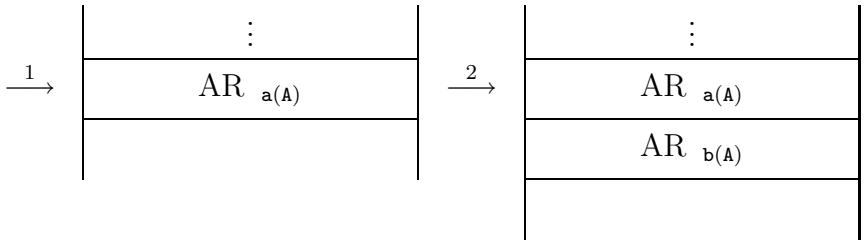
2.3 Der erste Optimierungsansatz

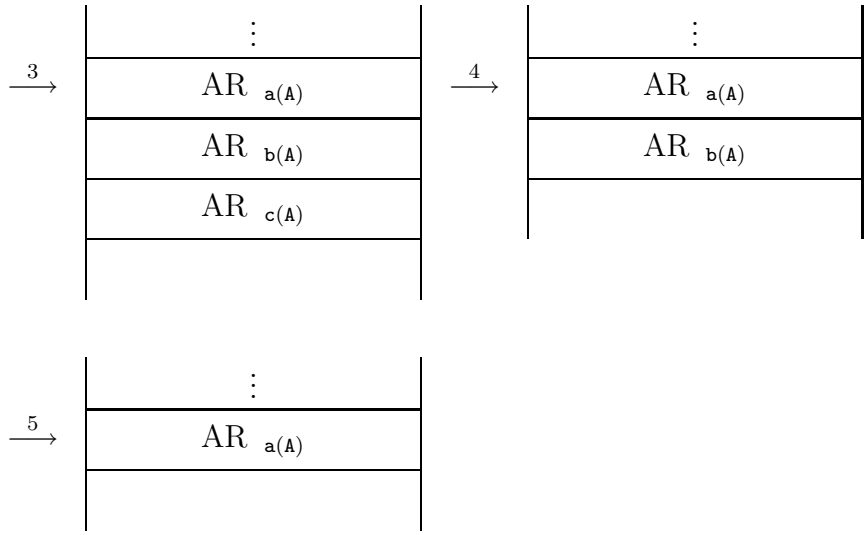
Betrachten wir die soeben dargestellte kellerartige Organisation des Laufzeitspeichers bei der Abarbeitung typischer funktionaler Programmstrukturen, so fällt eine erhebliche Ineffizienz im Umgang mit dem Speicherplatz auf: Wie für viele andere applikativen Programmiersprachen auch, ist es für den Anweisungsteil im Rumpf einer NSF typisch, daß er stets nur aus einem einzigen Ausdruck besteht (in dem wiederum weitere Ausdrücke vorkommen können). Zur Laufzeit, d.h. nach mehrmaliger Anwendung der Kopierregel, entsteht dann eine häufig tiefe Schachtlung von ineinander kopierten modifizierten Funktionsrümpfen. Dies hat zur Folge, daß der Laufzeitkeller solange mit AR's vollgeschrieben wird, bis das Ergebnis (durch Ausführung einer SF) anfällt. Der danach erfolgende aufwendige Verwaltungsvorgang hat mit dem eigentlichen Ergebnis (das ja schon vorliegt!) nichts mehr zu tun: Es erfolgt eine lange Reihe von Rücksprüngen, d.h. das jeweils letzte AR muß wieder gelöscht und die Indexregister (zur Verwaltung der statischen Verweiskette [La73]) umgeladen werden.

Beispiel 2.2 Das folgende sehr einfache Beispiel wurde ausgewählt, um mehrere geschachtelte NSF–Aufrufe zu bekommen, um gleichzeitig auf NSF–Aufrufe mit dicken Parametern zu verzichten und um eine kompakte Darstellung des Kellerablaufs zu ermöglichen. (Eine LISP/N–Version der bekannten Funktion Member, welche in [Ho83] (Beispiel IV.3.0) zur Motivation der anschließenden Optimierung dient, wird dort nicht entsprechend dem tatsächlichen Kellerablauf wiedergegeben: Es wied eine Call By Value–Semantik induziert, und die Aufrufe der für dicke Parameter notwendigen Hilfsfunktionen übergangen.):

```
a = λ x . { b(x) }
b = λ x . { c(x) }
c = λ x . { CONS(x,(B C)) }
a(A)
```

Wir betrachten nun die sukzessiven Veränderungen im Laufzeitkeller beim Aufruf von a(A):





Durch den NSF–Aufruf a(A) wird vom LZS das $AR_{a(A)}$ an der Kellerspitze angelegt. Zur Auswertung dieses Aufrufs findet der Aufruf b(A) statt und infolgedessen noch der Aufruf c(A). Daß heißt, ohne daß über ein Funktionsende gelaufen wird, findet bereits ein neuer NSF–Aufruf statt und deshalb werden solange AR's im Keller „gestapelt", bis kein neuer Aufruf mehr erfolgt. Bei der Auswertung von c(A) wird bereits die Liste (A B C) als Endergebnis geliefert und ein Funktionsende erreicht. In der Situation Funktionsende wird nun das letzte AR gelöscht, die Indexregister umgeladen, und mit der Abarbeitung an der Rücksprungadresse des Aufrufs fortgefahren. Dieser Vorgang wiederholt sich noch zweimal, bis der Aufruf a(A) komplett abgeschlossen ist.

Die Ineffizienz besteht nun darin, daß AR's unnötig lange im Speicher gehalten werden, die für die Berechnung des eigentlichen Ergebnisses nicht mehr benötigt werden. So wird für die Auswertung von b(A) das $AR_{a(A)}$ und für die Auswertung von c(A) weder das $AR_{a(A)}$ noch das $AR_{b(A)}$ benötigt. Nur beim Funktionsende wird noch auf die Rücksprungadresse in der Linkage zugegriffen, um an der alten Stelle im Code fortfahren zu können.

Für eine effizientere Speicherplatznutzung, d.h. für eine möglichst frühe Freigabe nicht mehr benötigter AR's, müssen wir uns zunächst überlegen, welche Informationen aus den einzelnen AR's überhaupt noch benötigt werden könnten: Im obigen Beispiel war lediglich noch die Rücksprungadresse nötig. Im allgemeinen wird aus *jedem* AR neben der Rücksprungadresse nur noch die gegebenenfalls vorhandenen Hilfsvariablen für Zwischenresultate, die bei der Auswertung von Termen mit mehrstelligen Standardfunktionen anfallen, benötigt. Nur wegen dieser beiden Informationen braucht ein ganzes AR nicht im Laufzeitkeller gehalten werden. Das ist offensichtlich unökonomisch!

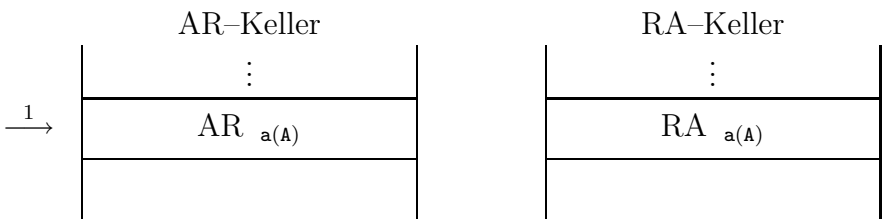
Nach [Ho/Li/Si83] werden daher zunächst die beiden folgenden Veränderungen im Laufzeitsystem eingeführt, um eine Speicherplatz–Optimierung zu erkennen und durchführen zu können:

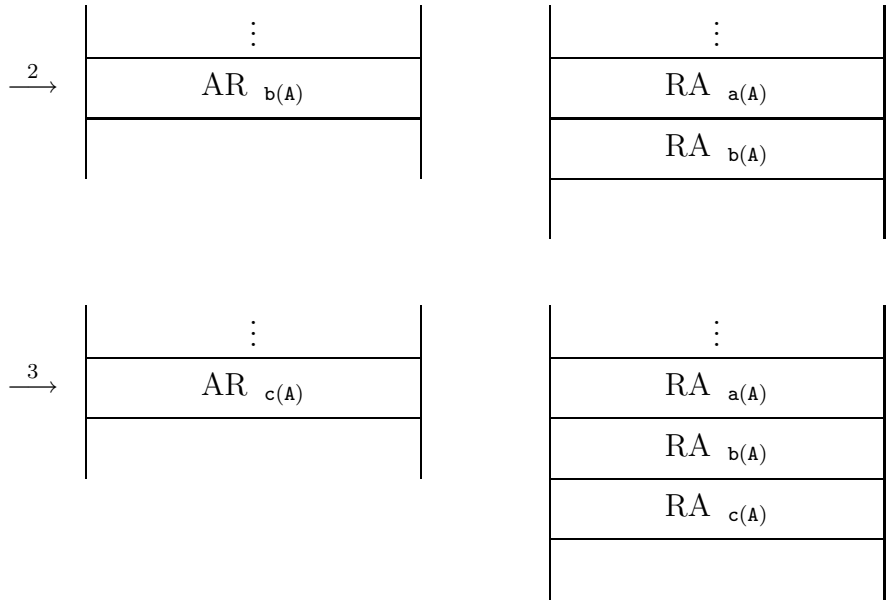
1. Speicherung von Rücksprungadressen (RA) und Hilfsvariablen (HV) in zwei zusätzlichen seperaten Kellern, in denen nur Push– und Pop–Operationen zulässig sind.
2. Verlagerung des AR–Löschvorgangs von der Situation „Funktionsende“ nach „Funktionsaufruf“:
   (a) Ein neues AR wird zunächst provisorisch angelegt.
   (b) Anhand der von diesem AR ausgehenden Verweise (siehe Bemerkung 2.3) in den Laufzeitkeller wird nach dem AR mit der geringsten Distanz zur Spitze des Kellers (aktueller Inhalt von BFS) gesucht.
   (c) Der Laufzeitkeller–Inhalt über dem soeben gefundenen AR wird bis zur Spitze gelöscht (siehe Bemerkung 2.4).
   (d) Das zuvor provisorisch angelegte AR wird im Keller abgespeichert.
   (e) Bei einem formalen Funktionsaufruf werden die Indexregister umgeladen.

Bemerkung 2.3 Bei den zu betrachtenden Verweisen handelt es sich in der Linkage (zunächst) ausschließlich um den Verweis „ein dynamisches Niveau des statischen Vorgängers“. Der Verweis „dynamisches Niveau des dynamischen Vorgängers“ zeigt immer auf das direkt zuvor angelegte AR und würde die beabsichtigte Optimierung stets verhindern. Außerdem sind die Verweise aus dem gesamten Parameterblock (siehe Tabelle 2.1) zu beachten: Falls es sich in den Parameterzellen um den Parametertyp „Funktionsidentifikator“ (Typ 1) oder „Funktionsidentifikator für dicke Parameter“ (Typ 2) handelt, müssen auch die statischen Verweise in diesen Zellen geprüft werden.

Bemerkung 2.4 Das provisorische Anlegen eines AR findet bereits an der Kellerspitze statt, d.h. falls der dynamische Vorgänger (das unmittelbar zuvor angelegte AR) nicht gelöscht werden darf, kann das provisorisch angelegte AR stehen bleiben, und es ist nichts weiter zu tun. Falls aber alte AR's gelöscht werden können, so werden sie nicht physisch gelöscht, sondern durch aktualisieren der globalen Variablen „Beginn des freien Speichers“ (kurz: BFS) freigegeben.

Wenden wir diese neue Strategie auf das Beispiel 2.2 an, so erhalten wir folgende Belegung der Keller (Hinweise: RA steht für Rücksprungadresse — Hilfsvariablen werden in diesem Beispiel nicht benötigt und der Hilfsvariablen–Keller wird deshalb nicht gezeigt):





Durch Ausführung der im RA–Keller eingetragenen Rücksprünge gelangt wird an das Ende des Hauptprogramms gelangt.

Da weniger AR's gleichzeitig im Speicher gehalten werden müssen und Rücksprungadressen wenig Speicherplatz in Anspruch nehmen, ist der geringere Speicherplatzbedarf offensichtlich. Allerdings führen die soeben vorgestellten Veränderungen alleine zu einer fehlerhaften Implementierung:

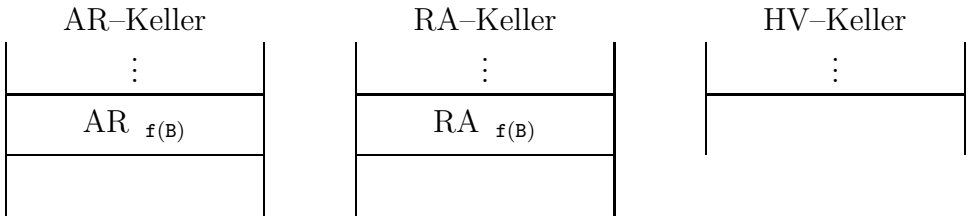
Beim Aufruf einer mehrstelligen SF $f_s(a_1, \ldots, a_n)$ mit den aktuellen Parametern $a_1, \ldots, a_n$ und $n \geq 2$ kann es vorkommen, daß durch die Auswertung der Parameter $a_1, \ldots, a_i$ mit $1 \leq i \leq n-1$ und somit eventuell bewirkter weiterer NSF–Aufrufe, auch das AR_c desjenigen Aufrufs $c = f(\ldots)$ der NSF f gelöscht wird, in derem Anweisungsteil der betrachtete SF–Aufruf von f_s steht. Obwohl Rücksprungadresse und Hilfsvariablen in seperaten Kellern ausgelagert sind, können für die Auswertung der Parameter $a_{i+1}, \ldots, a_n$ noch die aktuellen Parameter von c benötigt, durch Löschen des AR_c zum Aufruf c aber nicht mehr erreicht werden. Dazu folgendes

Beispiel 2.3 Gegeben seien die beiden Funktionen:

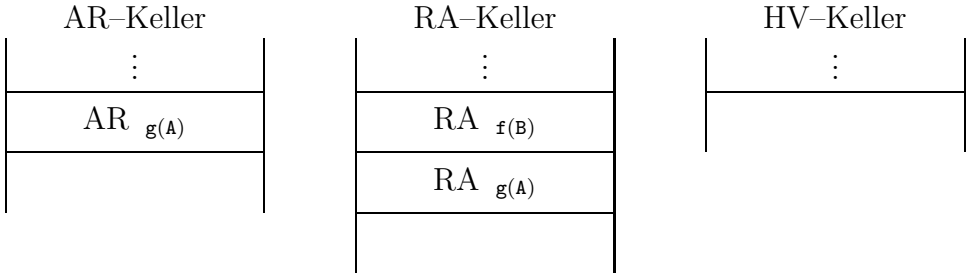
```
g = λ y . { y }
f = λ x . { CONS(g(A),g(x)) }
f(B)
```

Für den Aufruf f(B) werden wir der Vollständigkeit halber alle drei Keller angeben, wobei mit HV–Keller der Hilfsvariablen–Keller gemeint ist.

Zunächst wird das $\mathrm{AR}_{f(B)}$ provisorisch angelegt und nach gegebenenfalls frühzeitig freizugebener alter AR's im AR–Keller abgelegt:



Als nächstes wird der erste Parameter der SF CONS ausgewertet, d.h. ein AR für den Aufruf g(A) provisorisch angelegt. Da SN(g)=SN(f) und der aktuelle Parameter ein S–Ausdruck ist, zeigt von diesem AR kein Verweis auf das $\mathrm{AR}_{f(B)}$. Deshalb wird gemäß der Optimierung das $\mathrm{AR}_{f(B)}$ gelöscht, und das $\mathrm{AR}_{g(A)}$ im Keller eingetragen. Wir erhalten folgende Laufzeitkeller:



Danach muß der zweite Parameter von CONS ausgewertet und somit das $\mathrm{AR}_{g(B)}$ angelegt werden. Der aktuelle Parameter für x, d.h. das Atom B, war im $\mathrm{AR}_{f(B)}$ abgelegt. Dieses AR wurde jedoch gelöscht und der aktuelle Wert von x kann nicht mehr bestimmt werden.

Wir sehen, daß die vorgestellte Version eines optimierten Laufzeitsystems noch nicht korrekt arbeitet. Im nächsten Abschnitt wird gezeigt, wie das Löschen von eventuell noch benötigten AR's durch einen neuen Verweis in der Linkage verhindert werden kann.

2.4 Die Einführung des GDV

Frühstens bei der Auswertung des *letzten* Arguments einer mehrstelligen SF darf das AR gelöscht werden, das durch den Aufruf derjenigen Funktion angelegt wurde, in deren Anweisungsteil der Aufruf der SF erfolgte. Dies wird durch die Einführung eines neuen Verweises in der Linkage eines AR, dem sogenannten „generalisierten dynamischen Vorgänger“ (kurz: GDV), erreicht:

Definition 2.2 (GDV) Wird der Aufruf $c = f_s(a_1, \ldots, a_n)$ einer n–stelligen SF f_s mit den aktuellen Parametern $a_1, \ldots, a_n$ mit $n \geq 2$ im Anweisungsteil einer NSF f ausgeführt, dann ist der *GDV* im AR eines möglichen NSF–Aufrufs bei der Auswertung von

- a_i mit $1 \leq i \leq n-1$: ein Verweis auf das AR_h zum Aufruf der NSF f oder
- a_n: ein Verweis auf dasjenige AR, auf welches der GDV–Verweis vom AR_f zum Aufruf der NSF f zeigt (d.h. wiederherstellen des zuvor gültigen GDV–Verweises).

Tritt *kein neuer* GDV gemäß der beiden obigen Situationen in Kraft, so wird bei jedem weiteren NSF–Aufruf der GDV–Verweis vom dynamischen Vorgänger übernommen. Ist *kein* GDV gemäß der beiden obigen Situationen in Kraft, so ist der GDV das AR zum Hauptprogramm, d.h. das erste AR im AR–Keller.

Bemerkung 2.5 Der GDV kann also wie ein globaler Verweis (genau wie der Verweis BFS) in den AR–Keller verstanden werden, der mit dem ersten AR im Laufzeitkeller vorbesetzt ist und nur in den beiden Situationen gemäß Definition 2.2 verändert wird.

Bemerkung 2.6 Ein GDV–Verweis stellt somit sicher, daß zu Beginn der Auswertung *jedes* aktuellen Parameters eines mehrstelligen SF–Aufrufs die *selben* AR's im AR–Keller vorhanden sind.

Den auf Seite 23 angegebenen Veränderungen des Laufzeitsystems wird also hinzugefügt:

3. Ersetzen des Verweis auf ein „dynamisches Niveau des dynamischen Vorgängers“ durch einen Verweises auf den „generalisierten dynamischen Vorgänger“ (GDV) in der Linkage eines AR.
4. Wird ein Aufruf beendet, für den ein Verweis auf einen GDV eingerichtet wurde, so werden alle AR's nach dem GDV bis zur Spitze des Laufzeitkellers gelöscht und die Indexregister umgeladen.

Bemerkung 2.7 Durch 4. wird der zuvor gültige GDV–Verweis wiederhergestellt, da das AR des GDV nun oberstes AR ist. Entsprechend Bemerkung 2.4 werden AR's nicht physisch gelöscht, sondern durch aktualisieren des globalen Verweises BFS erreicht.

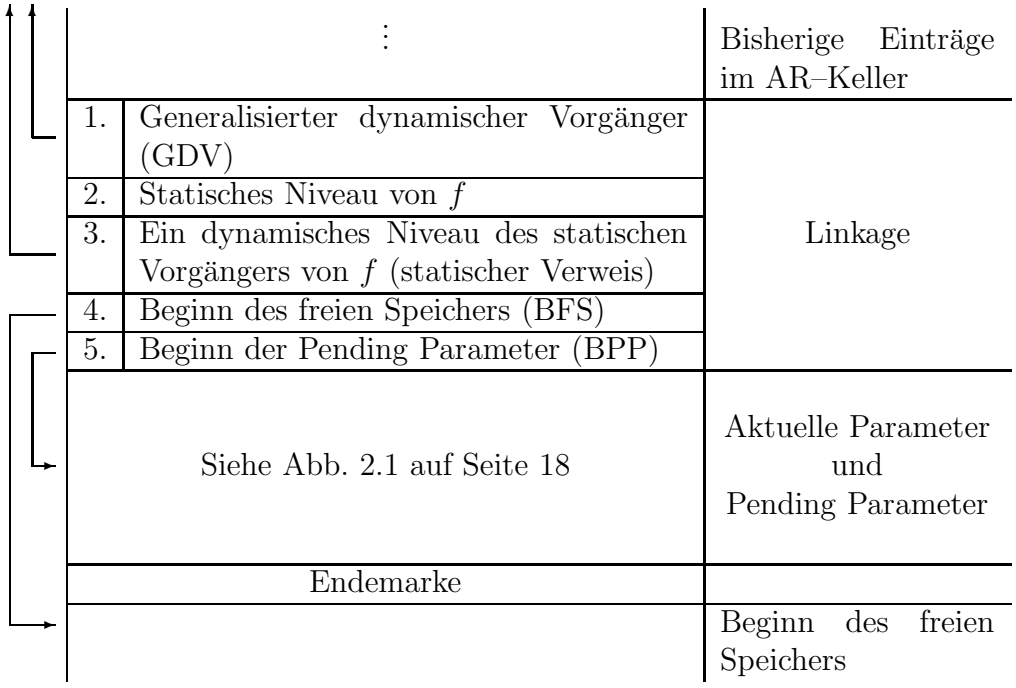
Bemerkung 2.8 In Anlehnung an Bemerkung 2.3 ergeben sich nun die folgenden Verweise in einem (zunächst) provisorisch angelegten AR, die hinsichtlich der möglichen frühzeitgen Freigabe alter AR's überprüft werden müssen:

1. GDV–Verweis (1. Linkagezelle),
2. der Verweis auf „ein dynamisches Niveau des statischen Vorgängers“ (3. Linkagezelle) und
3. die statischen Verweise in allen Parameterzellen vom Typ 1 oder Typ 2 (siehe Tabelle 2.1).

Bemerkung 2.9 Hinsichtlich der Definition 2.2 wird ein Konditional wie eine zweistellige Standardfunktion f_s gehandhabt: Der erste Parameter ist gerade der boolesche Ausdruck im if–Teil, der zweite Parameter entweder der then– oder der else–Teil. In LISP/N kommen somit generalisierte dynamische Vorgänger nur bei der Auswertung von linken Ästen (siehe Definition 1.9), d.h. Aufrufen von CONS, EQ und bei Auswertung eines if–Teils zum Tragen.

Bemerkung 2.10 Die Definition 2.2 des GDV kann alternativ auch so formuliert werden: Der dynamische Vorgänger des *ersten* NSF–Aufrufs innerhalb eines linken Astes ist der GDV für alle weiteren NSF–Aufrufe während der Auswertung dieses linken Astes (solange zwischenzeitlich kein neuer GDV auftritt).

Abbildung 2.2: Activation Record nach Honschopp



Weggefallen in der Linkage sind die Rücksprungadresse, die ja nun im RA–Keller „gekellert“ wird und der Verweis auf das „dynamische Niveau des dynamischen Vorgängers“, welcher durch den GDV–Verweis ersetzt wird. Die Hilfsvariablen werden nicht mehr am Ende eines AR, sondern im eigenen HV–Keller zwischengespeichert.
Noch eine Bemerkung zu den in den meisten LISP–Dialekten üblichen COND–Konditionalen:

Bemerkung 2.11 In LISP/N werden COND–Konditionals nicht unmittelbar unterstützt. Dafür steht die Pascal–ähnliche IF–THEN–ELSE–FI

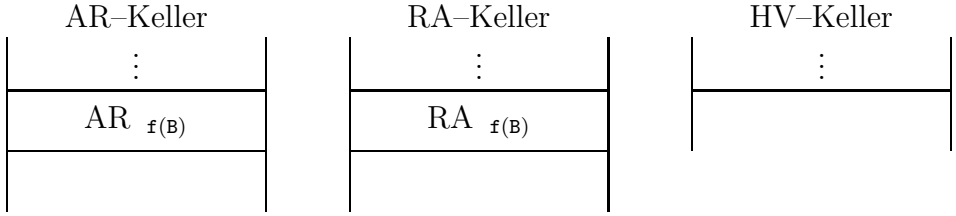
Kontroll–Struktur zur Verfügung. Dies bedeutet für LISP/N aber keine Einschränkung, da die COND–Struktur einer Folge von geschachtelten IF–Anweisungen entspricht:

```
(COND (Bed1 → Folg1)          IF Bed1 THEN Folg1
      (Bed2 → Folg2)                 ELSE IF Bed2 THEN Folg2
            ⋮             ⟺                          ELSE ...
      (Bedn−1 → Folgn−1)                               IF Bedn−1 THEN Folgn−1
      (T      → Folgn))                                        ELSE Folgn
                                                                    FI...FI FI
```

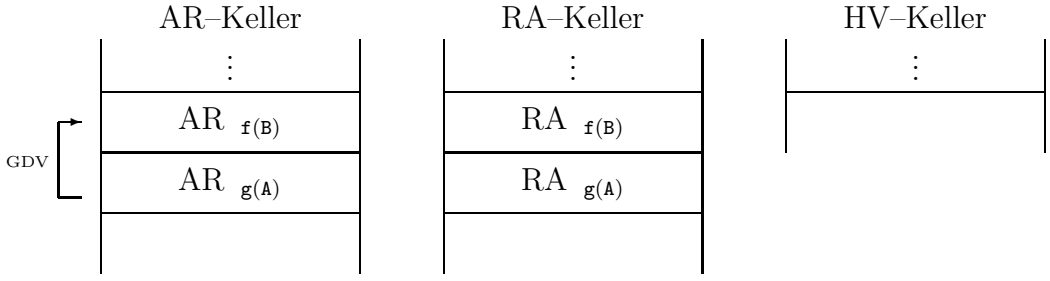
Das heißt, die Bedingungen bei COND entsprechen gerade den if–Teilen bei LISP/N.

Beispiel 2.3 (Fortsetzung): Betrachten wir nun für den Aufruf f(B) den korrigierten Ablauf in den Laufzeitkellern:

Wieder wird zunächst das $AR_{f(B)}$ provisorisch angelegt und, nach gegebenenfalls nach (neuer) Vorschrift zu löschender AR's, im AR–Keller abgelegt. Der GDV zeigt auf das erste AR im AR–Keller:

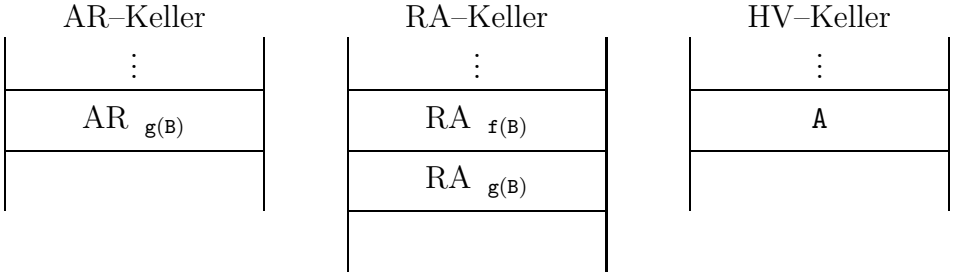


Danach wird wieder $AR_{g(A)}$ provisorisch angelegt. Da der Aufruf von g(A) aber im ersten Argument der zweistelligen Standardfunktion CONS erfolgte, zeigt der GDV auf das $AR_{f(B)}$. Diese AR darf deshalb nicht gelöscht werden, und wir erhalten:



Der linke Ast–Aufruf g(A) wird ausgeführt, und das Atom A als Zwischenergebnis der Auswertung von CONS im Hilfsvariablen–Keller abgelegt. Nach Beendigung von g(A) ist auch der linke Ast von CONS ganz abgearbeitet, und der GDV–Verweis wird wieder auf den alten Wert zurückgenommen (d.h. den GDV–Verweis im $AR_{f(B)}$). Das $AR_{f(B)}$ wird mittels dem BFS–Verweis wieder zum obersten AR erklärt, um für die Auswertung des rechten Astes die gleiche Umgebung wie zu Beginn des linken Astes zu haben.

Es folgt der Aufruf g(B) im rechten Ast von CONS und der aktuelle Wert von x, d.h. das Atom B, ist jetzt im $AR_{f(B)}$ erreicht worden. Das $AR_{g(B)}$ wird provisorisch angelegt, um die Verweise zu überprüfen: Im rechten Ast von CONS wird kein neuer GDV–Verweis gesetzt, und es verweist kein statischer Verweis auf das $AR_{f(B)}$, welches deshalb freigegeben wird:



Der Aufruf g(B) liefert das Atom B im Accumulator (AC) als Ergebnis zurück. Das Atom A wird vom HV–Keller „gepopt“ und die Standardfunktion CONS liefert die Liste (A . B) als Endergebnis. Nach Ausführung der Rücksprünge ist das Beispiel nun erfolgreich abgearbeitet worden.

Bemerkung 2.12 Honschopp hat dieses neue Laufzeitsystem an der Universität Kiel auf einer SIEMENS 7.760 unter BS2000 in SIEMENS–Pascal implementiert [Ho83] und die Korrektheit der Implementierung wurde von Kindler bewiesen [Ki87].

Kapitel 3

SKGI–Aufrufe nach Felgentreu

In der Dissertation von K.–U. Felgentreu [Fe87] wird eine „Optimierung von Funktionsaufrufen unter Shallow Binding für Static Scope LISP“ entwickelt. Diese Technik wird im Gegensatz zu der Optimierung in Honschopps Arbeit [Ho83], in der ein Compiler-Laufzeitsystem betrachtet wurde, an einem Interpreter vorgestellt. „Das verfolgte Ziel liegt darin, eine möglichst große entscheidbare Klasse von Aufrufen mit Hilfe eines einfachen Kriteriums zu erfassen. Die Technik umfaßt die aus der Literatur bekannten Optimierungen von Postrekursionen, wechselseitigen und verdeckten Postrekursionen und optimiert darüber hinaus eine Vielzahl nicht–rekursiver und sogar formaler Aufrufe. Die durch diese Technik erreichten Einsparungen — insbesondere an Speicherplatz — sind beträchtlich.“ ([Fe87], S.8)

Eine effiziente Implementierung und anschließende „Benchmarktests“ wurden von R. Beckmann vorgenommen [Be88].

Zunächst betrachten wir die Technik des Shallow Binding mit statischer Variablenbindung (Static Scoping). Anschließend betrachten wir die von Felgentreu eingeführte Standardisierung von LISP–Programmen, um dann den Kern der Arbeit, die „standardisiert kostenkünstig interpretierbaren“ Aufrufe (kurz: SKGI–Aufrufe oder auch LCC, d.h. Low Cost Calls) zu beschreiben. Ergebnis ist ein Markierungsalgorithmus, der bereits *vor* Beginn der eigentlichen Interpretation optimierbare Aufrufe als solche kennzeichnet.

3.1 Shallow Binding und Static Scoping

In den klassischen Implementierungen (wie die von Mc Carthy) wurde die Bindung von Werten an Namen durch eine sogenannte Assoziationsliste (kurz: A–Liste) realisiert. Dieses Verfahren ist aber sehr zeitaufwendig, da die Werte zu einem Namen nur durch einen Suchvorgang (oft tief unten) im Keller gefunden werden. Man spricht daher beim A–Listen Konzept auch von „Deep–Binding“ (tiefes Binden). Es wurden später eine Reihe effizienterer Methoden vorgestellt, wobei die bekannteste wohl die des „Shallow Binding“ (seichtes Binden) ist:

Die Grundidee des Shallow Binding ist, daß jedem Nichtstandardidentifikator–Vorkommen stets eine feste Adresse im Wertespeicher zugeordnet wird. Insbesondere wird verschiedenen definierenden Vorkommen desselben Nichtstandardidentifikators dieselbe Wertezelle zugeordnet. Der Bindungsalgorithmus muß also dafür Sorge tragen, daß zu jedem Zeitpunkt in der jeweiligen Umgebung der gültige Wert in der entsprechenden Wertezelle steht. Neben dem Wertespeicher steht dafür noch ein Datenkeller und ein Keller für Rücksprungadressen bereit.

Die Zuordnung von Wertezellen zu Nichtstandardidentifikatoren wird beim Einlesen des Programms vorgenommen. Erfolgt bei der Ausführung des Programms ein Funktionsaufruf, so werden zunächst die für die Abarbeitung des Funktionsrumpfes relevanten Identifikatoren festgestellt. Die „alten“ Werte dieser Identifikatoren werden in den Datenkeller gerettet und danach die „neuen“, d.h. gültigen Werte in die entsprechende Wertezelle eingetragen. Nach Abarbeitung des Funktionsrumpfes werden die „neuen“ Werte im Speicher wieder durch die „alten“ Werte, die stets als oberster Block an der Datenkeller–Spitze stehen, ersetzt.

Wir werden im Abschnitt 2.4 sehen, inwieweit dieses Retten und Wiederherstellen der „alten“ Umgebung, welches einen beträchtlichen Aufwand darstellt (also sehr kostenintensiv ist), unterbleiben kann, ohne daß sich das Interpretationsergebnis ändert.

Die betrachtete Implementierung basiert auf einer sogenannten statischen Variablenbindung (Static Scoping). Im Gegensatz zur dynamischen Variablenbindung (Dynamic Scoping), wie sie früher oft benutzt wurde (sie führt bei großen Programmen schnell zur Intransparenz), wird bei Static Scoping beim Zugriff auf eine freie Variable in einem Rumpf einer Funktion f stets derjenige Wert ermittelt, der bei der ursprünglichen Definition von f gültig war. Die Werte der freien Variablen einer Funktion werden in der sogenannten „Closure“ der Funktion festgehalten.

Für weitere Informationen und Beispiele wird auf [Fe87] verwiesen.

3.2 Die Standardisierung von LISP–Programmen

Die Standardisierung eines zu interpretierenden LISP–Eingabeprogramms stellt eine für die im nächsten Abschnitt dargestellte Optimierung *notwendige* Maßnahme dar. Zugleich ist sie für sich alleine betrachtet schon eine eigenständige Optimierung für Shallow–Binding, da der Bindungsalgorithmus vereinfacht und somit die Allokation von Werten günstiger wird. Die Standardisierung kann ohne großen Zeitaufwand *vor* der eigentlichen Interpretation, also z.B. in der Einlesephase des Programms, durchgeführt werden.

Die Standardisierung s ist eine Programmtransformation, die jedem echten

LISP–Programm p ein semantisch äquivalentes Programm p^s zuordnet, in dem nur noch die Nichtstandardidentifikatoren G_i (G steht für gebunden) und F_j (F steht für frei) vorkommen, wobei die Indizes i, j eine Art „Durchnummerierung“ der Identifikatoren darstellen:

Definition 3.1 Die *Standardisierung* s eines beliebigen echten LISP–Programmes p in Datensprache ist durch folgende 3 Schritte definiert:

1. Jede Lambda–Funktion f, die nicht die Lambda–Funktion einer Label–Funktion ist, wird ersetzt durch die Label–Funktion `(LABEL` id f`)`, wobei id ein Nichtstandardidentifikator ist, der in p nicht auftritt.

2. Falls das Programm Label–Funktionen mit freien Nichtstandardidentifikatoren enthält, wird wie folgt verfahren:

   (a) Zunächst werden alle Identifikatoren der Gestalt F_j zulässig umbenannt in Nichtstandardidentifikatoren, die nicht von dieser Gestalt sind.

   (b) Danach werden alle Label–Funktionen mit freien Nichtstandardidentifikatoren sukzessiv von innen nach außen wie folgt ersetzt: Sei f eine Label–Funktion mit $\text{frei}(f) = \{v_1^f, \ldots, v_k^f\}$, $k > 0$. Dann wird f ersetzt durch

   `((LAMBDA (`$F_1 \ldots F_k$`)` f'`)` $v_1^f \ldots v_k^f$`)`,

   wobei f' aus f hervorgeht, indem für $j = 1, \ldots, k$ jedes freie Vorkommen von v_j^f in f durch F_j ersetzt wird.

3. Schließlich geht jede Label–Funktion

   `(LABEL` id_f `(LAMBDA` $v_1 \ldots v_n$`)` r `)`

   durch zulässige Umbennenung über in

   `(LABEL` G_{n+1} `(LAMBDA` $G_1 \ldots G_n$`)` $\hat{r}$ `)`.

Bemerkung 3.1 Die semantische Äquivalenz zwischen einem echten LISP–Programm p und der standardisierten Fassung p^s wurde in [Fe87] bewiesen.

Beispiel 3.1 Die Wirkung der Standardisierung verdeutlichen wir am Beispiel der Funktion List3 (vgl. [Fe87], S.174):

```
f=(LABEL LIST3 (LAMBDA (X)
      (LAMBDA (Y Z)
         (LABEL CONSTLIST (LAMBDA () (CONS Z (CONS X (CONS Y NIL)))))
      )
   ))
```

Durch den Schritt 1 aus Definition 3.1 geht f über in

f^1=
```
(LABEL LIST3 (LAMBDA (X)
       (LABEL EGAL (LAMBDA (Y Z)
          (LABEL CONSTLIST (LAMBDA () (CONS Z (CONS X (CONS Y NIL)))))
       ))
    ))
```

Durch den Schritt 2b entstehen nun nacheinander die folgenden Funktionen:

f^2=
```
(LABEL LIST3 (LAMBDA (X)
       (LABEL EGAL (LAMBDA (Y Z)
          ((LAMBDA (F1 F2 F3)
              (LABEL CONSTLIST (LAMBDA () (CONS F1 (CONS F2 (CONS F3 NIL)))))
           )
           Z X Y)
       ))
    ))
```

f^3=
```
(LABEL LIST3 (LAMBDA (X)
       ((LAMBDA (F1)
           (LABEL EGAL (LAMBDA (Y Z)
              ((LAMBDA (F1 F2 F3)
                  (LABEL CONSTLIST (LAMBDA () (CONS F1 (CONS F2 (CONS F3 NIL)))))
               )
               Z F1 Y
           ))
        )
        X)
    ))
```

Damit ergibt sich schließlich gemäß Schritt 3 als standardisierte Fassung:

f^s=
```
(LABEL G2 (LAMBDA (G1)
       ((LAMBDA (F1)
           (LABEL G3 (LAMBDA (G1 G2)
              ((LAMBDA (F1 F2 F3)
                  (LABEL G1 (LAMBDA () (CONS F1 (CONS F2 (CONS F3 NIL)))))
               )
               G2 F1 G1
           ))
        )
        G1)
    ))
```

In einem standardisierten Programm p^s kommen unbenannte Lambda–Funktionen nur noch durch die Konstruktion in Schritt 2b der Definition 3.1 vor, d.h. nur dann, wenn eine Label–Funktion freie Nichtstandardidentifikatoren enthält.

Es ist aber offensichtlich unökonomisch, einer Label–Funktion durch die Einbettung in einer Lambda–Funktion die Identifikatoren ihrer freien Variablen zuzuführen.

Um diesen unnötigen Funktionsaufruf zu vermeiden, wird vereinbart, daß beim Standardisieren hinter jedem Rumpf einer Label–Funktion eine Liste der freien Nichtstandardidentifikatoren angehängt wird.

Erfolgt nun der Aufruf einer solchen Label–Funktion, so bildet der Interpreter anhand dieser Liste eine Closure mit den gerade aktuellen Werten dieser freien Variablen.

Alle durch den Schritt 2b in Definition 3.1 erzeugten Lambda–Funktionen der Gestalt

((LAMBDA ($F_1 \ldots F_k$) (LABEL G_{m+1} (LAMBDA (...) r))) $x_1 \ldots x_k$)

mit $k \geq 0$ werden deshalb noch beim Standardisieren ersetzt durch

(LABEL G_{m+1} (LAMBDA (...) r <$x_1 \ldots x_k$>)).

Wir sprechen dann von einem *standardisierten Programm in Kurznotation.*

Beispiel 3.2 Für die Funktion LIST3 aus Beispiel 3.1 ergibt sich dann folgende Kurznotation:

```
(LABEL G2 (LAMBDA (G1)
   (LABEL G3 (LAMBDA (G1 G2)
      (LABEL G1 (LAMBDA () (CONS F1 (CONS F2 (CONS F3 NIL))) <G2 F1 G1> ))
       <G1> ))
 <> ))
```

Ein in INTERLISP geschriebenes Programm RENAME, welches die Standardisierung mit Kurznotation durchführt, wird in [Fe87], S.193ff vorgestellt.

Wie eingangs erwähnt, stellt die Standardisierung eine eigenständige Optimierung dar. Dadurch, daß alle Identifikatoren G_i und F_i einer Funktion durch die Standardisierung durchnummeriert und somit auch bekannt sind, ist es möglich, sie in einem bestimmten jeweils zusammenhängenden Bereich im Datenspeicher abzulegen. Der Platzbedarf und der Zeitaufwand für die Allokation von Nichtstandardidentifikatoren wird somit minimal.

Erfolgt ein Aufruf einer m–stelligen Nichtstandardfunktion, so kann das Retten der „alten Umgebung“ durch einen schnellen Blocktransfer der Werte der Adressen für $G_1, \ldots, G_{m+1}$ in den Datenkeller erfolgen. Zudem braucht nur noch die Rücksprungadresse und der Zeiger auf die Werteliste der alten Closure (gemäß standardisiertem Shallow–Binding) mitgespeichert werden. Die nötigen Aktionen beschränken sich also auf $m + 3$ Zellen.

3.3 SKGI–Aufrufe

In den Abschnitten 2.2 und 2.3 haben wir die Technik des standardisierten Shallow Binding kurz vorgestellt. Wir sahen, daß die Effizienz gegenüber älteren Methoden durch die Identifizierung eines Nichtstandardidentifikators mit einer festen Adresse im Wertespeicher begündet ist. Um zu gewährleisten,

daß bei einem Zugriff auf diese Wertezelle stets der gültige Wert vorhanden ist, sind vom Bindungsalgorithmus bei der Situation „Funktionseintritt“ die folgenden allgemeinen Maßnahmen

(M1) Retten der alten Umgebung,

(M2) Herstellen der neuen Umgebung

und bei der Situation „Funktionsende“ die Maßnahme

(M3) Wiederherstellen der alten Umgebung

nötig. Dabei sind M1 und M3 am kostenintensivsten, d.h. sie benötigen die meiste Rechenzeit und den meisten Speicherplatz.

In [Fe87] ist untersucht worden, unter welchen Umständen M1 und M3 unnötig sind, d.h. in welchen Fällen sich durch die Einsparung von M1 und M3 das Interpretationsergebnis nicht ändern kann.

Im Gegensatz zu früheren Arbeiten, in denen solche Überlegungen nur für bestimmte Klassen von Funktionen, wie zum Beispiel die der postrekursiven, verdeckt postrekursiven und wechselseitig (verdeckt) postrekursiven, werden in [Fe87] *alle* Fälle untersucht, in denen M1 und M3 ausgelassen, d.h. „*kostengünstig interpretiert*“ (kurz: KGI) werden dürfen. Diese Optimierung liefert einen Algorithmus, der in einem standardisierten LISP–Programm alle Aufrufe c mit einer Marke versieht, die angibt, ob und wenn in welchem Umfang die Aktionen M1 und M3 bei der Ausführung von c eingespart werden können. Entscheidend am Erfolg dieser Optimierung ist, daß der weiter unten vorgestellte Algorithmus als Eingabe lediglich das standardisierte Programm p^s benötigt, nicht jedoch dessen Eingabedaten. Dies bedeutet, daß die Kennzeichnung aller optimierbaren Aufrufe genauso wie die Standardisierung bereits statisch, also *vor* Interpretationsbeginn, und zwar mit vernachlässigbarem Zeitaufwand beim Einlesen des Programms erfolgen kann.

Bevor wir nun den Entscheidungsalgorithmus für die SKGI–Eigenschaft (SKGI steht für „standardisiert kostengünstig interpretierbar“) vorstellen, folgende zentrale Definition:

Definition 3.2 Sei p ein echtes LISP–Programm und c ein Aufruf in p.

- Der *lokale Kontext von* c in p (kurz: „$\mathrm{LK}_{Fe}(c)$“) ist der Rumpf der kleinsten c umfassenden Funktion.
- Der *relevante lokale Kontext von* c in p (kurz: „$\mathrm{RLK}_{Fe}(c)$“) ist der lokale Kontext von c
  (a) ohne den Aufruf c,
  (b) ohne die syntaktischen Zeichenreihen links von c und

(c) falls c in der Folgerung einer Klausel einer bedingten Form vorkommt, ohne die übrigen Klauseln dieser bedingten Form.

In [Fe87] wurde gezeigt, daß der $\mathrm{RLK}_{Fe}(c)$ derjenige Teil von $\mathrm{LK}_{Fe}(c)$ ist, der bei der Fortsetzung von c im $\mathrm{LK}_{Fe}(c)$ noch (ganz oder teilweise) abgearbeitet werden kann. Das Retten und Wiederherstellen gemäß der Maßnahmen M1 und M3 kann sich somit auf diejenigen Informationen beschränken, die im $\mathrm{RLK}_{Fe}(c)$ auch vorkommen, also evtl. noch benötigt werden. Zunächst verdeutlichen wir den syntaktischen Umfang des RLK_{Fe} an folgendem

Beispiel 3.3 An der standardisierten Fassung der bekannten Funktion Append geben wir für die Aufrufe

(a) c_1 = `(null`^{s} `G1)`

(b) c_2 = `(CAR G1)`

(c) c_3 = `(G3 (CDR G1) G2)`

den jeweiligen relevanten lokalen Kontext an. *null*^{s} stellt die standardisierte Fassung dieser gängigen Funktion dar. Der jeweilig betrachtete Aufruf c ist dabei kursiv und der zugehörige $\mathrm{RLK}_{Fe}(c)$ unterstrichen dargestellt.

(a) append^s =
```
(LABEL G3
   (LAMBDA (G1 G2)
      (COND (null^s G1)    G2)
            ( T          (CONS (CAR G1)
                               (G3 (CDR G1) G2))))
            <> ))
```

(b) append^s =
```
(LABEL G3
   (LAMBDA (G1 G2)
      (COND ((null^s G1) G2)
            ( T         (CONS (CAR G1)
                              (G3 (CDR G1) G2))))
            <> ))
```

(c) append^s =
```
(LABEL G3
   (LAMBDA (G1 G2)
      (COND ((null^s G1) G2)
            ( T         (CONS (CAR G1)
                              (G3 (CDR G1) G2))))
            <> ))
```

Bevor der Entscheidungsalgorithmus für die SKGI–Eigenschaft eines Aufrufs c vorgestellt wird, wird die Klasse der „SSB–relevanten“ Aufrufe eingeführt, denn nur diese Aufrufe können Ziel der SKGI–Optimierung sein:

Definition 3.3 Sei $p^s = (p\ e_1 \ldots e_n)$ ein standardisiertes LISP–Programm mit echten Eingabedaten, und sei $c = (f\ a_1 \ldots a_m)$ ein Aufruf in p. Dann heißt c *Standardisiert–Shallow–Binding–relevant* (kurz: SSB–relevant) genau dann, wenn es mindestens eine m–stellige Nichtstandardfunktion in p gibt, die durch c aufgerufen wird.

Bemerkung 3.2 In [Fe87] ist gezeigt worden, daß die SKGI–Eigenschaft für einen Aufruf c ohne Beschränkung auf die SSB–relevanten Aufrufe im allgemeinen statisch unentscheidbar ist. Diese Beschränkung bedeutet aber praktisch keine Einschränkung, weil die betrachtete Optimierung natürlich nur Sinn macht, wenn der Aufruf c

- keine Standardfunktion ist,
- eine korrekte Anzahl m von Parametern hat,
- überhaupt zur Ausführung gelangt, d.h. erreicht wird.

Bemerkung 3.3 Ebenfalls wurde in [Fe87] gezeigt, daß es unentscheidbar ist, ob ein Aufruf c SSB–relevant ist. Auch dies ist praktisch unerheblich, denn der unten folgende Algorithmus terminiert auch bei *nicht* SSB–relevanten Aufrufen und kann deshalb gefahrlos auf jeden Aufruf angewandt werden.

Unter diesen Vorraussetzungen erhalten wir nun den entscheidenden, in [Fe87] bewiesen Satz:

Satz 3.1 Die SKGI–Eigenschaft ist entscheidbar für SSB–relevante Aufrufe.

Dieser Satz ist die Grundlage für den folgenden Algorithmus, der für jeden Aufruf c in einem standardisierten Eingabeprogramm entscheidet, inwieweit c zur Interpretationszeit optimiert werden kann. Dabei wird unterschieden zwischen:

- *Nicht SKGI*: Die Maßnahmen M1 und M3 müssen gemäß dem Shallow–Binding Konzept in vollem Umfang ausgeführt werden.
- *Schwach SKGI*: Da freie Variablen im RLK_{Fe} vorkommen, müssen für M1 lediglich der Verweis auf die zuvor gültige Closure–Werteliste in den Datenkeller und die Rückkehradresse in den Adresskeller gerettet werden. In M3 werden nur diese beiden Werte wieder zurückgeladen.
- *Stark SKGI*: M1 und M3 können komplett entfallen.

Entscheidungsalgorithmus für die SKGI–Eigenschaft:

Gegeben sei ein Aufruf c in einem standardisierten LISP–Programm p^s.

1. Bestimme die Anzahl m der Argumente von c.
2. Bestimme die Stelligkeit n der kleinsten c umfassenden NSF.
3. Erstelle frei($\mathrm{RLK}_{Fe}(c)$) und überprüfe
   (a) $m \leq n$
   (b) $\forall\ G_i \in$ frei($\mathrm{RLK}_{Fe}(c)$): $i > m+1$
4. – Wenn (3.a) und (3.b) erfüllt sind, und
   - falls frei($\mathrm{RLK}_{Fe}(c)$) kein F_j enthält, dann gib aus: „c ist *stark* SKGI“,
   - falls frei($\mathrm{RLK}_{Fe}(c)$) mindestens ein F_j enthält, dann gib aus: „c ist *schwach* SKGI“,
   – In allen anderen Fällen gib aus: „c ist *nicht* SKGI“.

Diese Informationen, d.h. ob ein Aufruf $c = (f\ \ldots)$ stark, schwach oder nicht kostengünstig interpretiert werden kann, wird durch eine entsprechende Marke im Programmtext festgehalten: Aus dem Aufruf $c = (f\ \ldots)$ wird dann $c' = (f$ mark $\ldots)$ mit *mark*$\in$\{StarkSKGI,SchwachSKGI,NichtSKGI\}.

In [Fe87] (S.222ff) stellt Felgentreu ein in INTERLISP geschriebenes Programm MARK vor, welches als Eingabe ein durch RENAME standardisiertes Programm p^s in Kurznotation erwartet und als Ausgabe gemäß obigen Algorithmus ein mit entsprechenden Marken versehenes Programm p^{sm} liefert.

Wir wollen die Wirkung von MARK an dem standardisierten LISP–Programm Append in Kurznotation aus Beispiel 3.3 verdeutlichen:

```
(LABEL G3
  (LAMBDA (G1 G2)
     (COND ((null^S NichtSKGI G1) G2)
           ( T                    (CONS StarkSKGI
                                        (CAR NichtSKGI G1)
                                        (G3 StarkSKGI
                                            (CDR NichtSKGI G1)
                                            G2))))
     <> ))
```

Zur Laufzeit, also dynamisch, kann dann der Interpreter durch einfachen Zugriff auf diese im wesentlichen statisch erzeugten Marken entscheiden, in welchem Umfang die Maßnahmen M1 und M3 ausgelassen werden dürfen.

3.4 Die Klasse der SKGI–Aufrufe

Mit dem Static Scope LISP–Interpreter mit der Technik des Standardisierten Shallow Binding und der LCC–Optimierungstechnik wird eine weitreichende Optimierung von Funktionsaufrufen erreicht. Mit den vorgestellten einfachen Kriterien wird dabei eine sehr große Menge der in der Literatur bisher behandelten Klassen von Aufrufen echt umfaßt und optimiert. Bei den gängigsten dieser Klassen, wie z.B. die der „postrekursiven“, „verdeckt postrekursiven“ und „wechselseitig (verdeckt) postrekursiven“, handelt es sich stets um rekursive Aufrufe auf (verdeckten) Postpositionen. Die Klasse der SKGI–Aufrufe ist an diese Einschränkungen jedoch nicht gebunden.

Ohne auf die genauen Definitionen der jeweiligen Klassen und Beispiele explizit einzugehen (dazu sei auf [Fe87] verwiesen), wollen wier hier die Hierachie der nach Felgentreu optimierten Klassen von Aufrufen [Fe87] (S.149) kurz wiedergeben:

1. Aufrufe auf Postpositionen („goto type calls“)
   - 1.1. postrekursive Aufrufe
   - 1.2. wechselseitig postrekursive SKGI–Aufrufe
   - 1.3. PZ–Aufrufe („Parallele Zuweisung“)
   - 1.4. formale „goto type“ Aufrufe mit SKGI–Eigenschaft
     - 1.4.1. formale postrekursive Aufrufe
2. Aufrufe auf verdeckten Postpositionen
   - 2.1. verdeckt postrekursive Aufrufe
   - 2.2. wechselseitig verdeckt postrekursive SKGI–Aufrufe
   - 2.3. „last call optimization“ nach Warren
   - 2.4. formale Aufrufe auf verdeckten Postpositionen mit SKGI–Eigenschaft
     - 2.4.1. formale verdeckt postrekursive Aufrufe
3. Aufrufe auf total verdeckten Postpositionen
   - 3.1. total verdeckt postrekursive Aufrufe
   - 3.2. nicht rekursive SKGI–Aufrufe auf total verdeckten Postpositionen
4. sonstige Aufrufe
   - 4.1. SKGI–Aufrufe in Bedingungen
   - 4.2. formale Aufrufe mit SKGI–Eigenschaft auf sonstigen Positionen

Bemerkung 3.4 Alle Aufruf–Klassen in der obigen Aufzählung, in der die SKGI–Eigenschaft nicht explizit angegeben ist, werden uneingeschränkt von der Klasse der SKGI–Aufrufe umfaßt. So gehören z.B. alle rekursiven Aufrufe

c, die sich auf einer Postposition befinden, der Klasse der SKGI–Aufrufe an, weil zum einen der $\mathrm{RLK}_{Fe}(c)$ aufgrund der Postposition immer leer ist und zum anderen aufgrund der Rekursivität von c die Anzahl der Argumente von c mit der Stelligkeit der kleinsten c umfassenden NSF übereinstimmt, d.h. die beiden Bedingungen (3.a) und (3.b) vom SKGI–Entscheidungsalgorithmus sind immer erfüllt.

3.5 Die Relevanz der Optimierung

Die Effizienz eines Static Scope Shallow–Binding Interpreters, der seine Eingabeprogramme zunächst standardisiert und markiert, wurde von R. Beckmann untersucht [Be88]: Dabei wurde der Interpreter als C–Programm auf diversen Maschinen implementiert und anhand einiger typischer Programme und Eingabedaten getestet. Die Testprogramme sind dabei zu einem Großteil aus dem „Gabriel-Set“ (siehe [Ga85]) entnommen.

Diese „Benchmark-Tests“ zeigten dann die Relevanz der Optimierungen: Es wurde festgestellt, daß (bei den ausgewählten Beispielen) im Durchschnitt etwa 75% des Kellerspeichers eingespart werden können, wobei rund 1/3 der Einsparungen alleine durch die Standardisierung bewirkt werden. Die Einsparungen an Speicherplatz können also beachtlich sein!

Es hat sich aber auch gezeigt, daß dies in dem Umfang für den zeitlichen Gewinn *nicht* gilt: Unter Berücksichtigung des zeitlichen Aufwands für die Standardisierung und der Markierung wurde nur in ganz speziellen Anwendungen ein zeitlicher Gewinn von 10% erzielt. Ist das Eingabeprogramm relativ groß im Vergleich zu den auszuführenden Instruktionen, kann auch mal eine längere Ausführungszeit entstehen. Im Durchschnitt sind Laufzeitgewinne von knapp unter 5% zu erwarten.

Zusammenfassend können wir sagen, daß die von Felgentreu eingeführte Klasse der SKGI–Aufrufe im Zusammenhang mit der Standardisierung zu erheblichen Einsparungen an Speicherplatz führen kann, ohne daß sich dies zeitmäßig besonders auswirkt. Bei geeigneter Implementierung könnte ein sehr konkurenzfähiges Interpreter–System entstehen.

Im nächsten Kapitel wenden wir uns wieder dem Laufzeitsystem für den LISP/N–Compiler von Honschopp zu:

Kapitel 4

Nach Honschopp optimierte Aufrufe

Wie wir in Kapitel 2 gesehen haben, ist eine wichtige Neuerung im Laufzeitsystem nach Honschopp, daß Speicherplatz nicht erst nach der Abarbeitung von Aufrufen, sondern schon vorher freigegeben werden kann. Dies eröffnet grundsätzlich neue Möglichkeiten, um nicht mehr benötigten Kellerspeicher zu erkennen und freizugeben.

Ziel dieses Kapitels ist es, formale Aussagen darüber zu machen, wie die Klasse der nach Honschopp optimierten Funktionen aussieht.

Dabei orientieren wir uns hier stark an der Implementation von Honschopp [Ho83] und versuchen zunächst die Kriterien, die zur Freigabe von Kellerspeicherplatz führen, formal zu fassen.

Unter Berücksichtigung der Maßnahmen, die der Compiler zur Handhabung dicker Parameter durchführt (Abschnitt 2.2), können wir bereits *statisch*, d.h. vor der eigentlichen Programmausführung durch das Laufzeitsystem, entscheiden, ob ein NSF–Aufruf dynamisch (d.h. falls er ausgeführt wird!) eine frühzeitige Speicherplatzfreigabe bewirkt.

Aufgrund der Call By Name–Semantik von LISP/N und der Tatsache, daß auch Verweise aus dem Parameterteil eines AR über eine frühzeitige Speicherplatzfreigabe bestimmen (siehe Bemerkung 2.8), können wir dann mit gewissen „Auflagen“ an die aktuellen Parameter eines NSF–Aufrufs Vergleiche zu bekannten Klassen von Funktionsaufrufen ziehen.

Den Abschluß dieses Kapitels bildet ein Vergleich zu der im Kapitel 3 vorgestellen LCC–Optimierungstechnik nach Felgentreu.

4.1 Honschopp–Optimierung auslösende Aufrufe

Zunächst einmal müssen wir festlegen, *wann* wir überhaupt von einer optimierten Abarbeitung eines NSF–Aufrufs in Honschopps Implementation sprechen können. Dazu folgende

Definition 4.1 Sei das AR_c zu dem Aufruf $c = f(\ldots)$ einer NSF f zunächst provisorisch angelegt worden. Falls aufgrund der von dem AR_c ausgehenden Verweise (siehe Bemerkung 2.8) noch mindestens ein AR von der AR–Kellerspitze aus freigegeben werden kann, so bewirkt der Aufruf c eine *Honschopp–Optimierung* (kurz: *HOpt*).

Bemerkung 4.1 Man beachte, daß ein Aufruf c, der eine HOpt bewirkt, nicht selber optimiert wird, sondern eine nachträgliche Optimierung von zuvor erfolgten Aufrufen darstellt, d.h. derern AR's ggf. frühzeitig durch c freigegeben werden.

Nun betrachten wir eine formale Erfassung von Situationen, die eine HOpt *verhindern*. Im folgenden Lemma verhindert der GDV–Verweis eine HOpt:

Lemma 4.1 Sei $c' = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s mit $f_s \in \{CONS, EQ\}$, bzw. die Auswertung eines Konditionals, d.h. $c' =$ IF a_1 THEN a_2 ELSE a_3 FI, und sei $c'' = g(\ldots)$ ein Aufruf der kleinsten c' umfassenden NSF g (wobei g auch das Hauptprogramm bzw. eine vom Compiler erzeugte (siehe Abschnitt 2.2) sein kann). Befindet sich das $AR_{c''}$ an der AR–Kellerspitze, so bewirkt jeder Aufruf $c = f(\ldots)$ einer NSF f, welcher für die Auswertung von a_1 ausgeführt werden muß, *keine* HOpt.

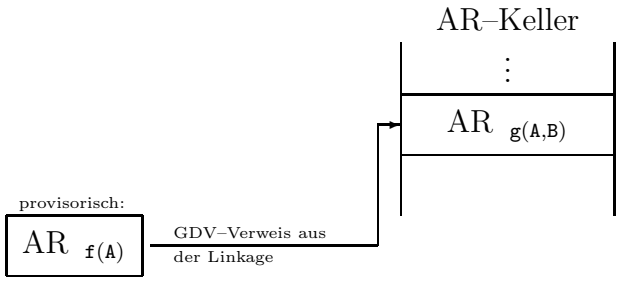
Beweis (Lemma 4.1): Klar, denn wenn c' im Anweisungsteil der NSF g ausgeführt wird, ist per Definition 2.2 das AR_g der GDV für NSF–Aufrufe bei der Auswertung des linken Astes a_1 und darf somit nicht gelöscht werden. Ist das AR_g das oberste AR, d.h der dynamische Vorgänger, wird somit eine HOpt verhindert. (Tritt ein neuer GDV während der Auswertung von a_1 in Kraft, so ist wieder ein linker Ast a_1' erreicht worden, und das Lemma gilt entsprechend für ein $AR_{g'}$, welches *nach* dem zuvor gültigen GDV (AR_g) im AR–Keller stehen muß.) □

Beispiel 4.1

$$\texttt{f} = \lambda\ \texttt{z}\ .\ \{\ \texttt{z}\ \}$$
$$\texttt{g} = \lambda\ \texttt{x}\ \texttt{y}\ .\ \{\ \underbrace{\texttt{CONS}(\underbrace{\texttt{f(x)}}_{c},\texttt{y})}_{c'}\ \}$$

$$\underbrace{\texttt{g(A,B)}}_{c''}$$

Zur Veranschaulichung soll folgende Abbildung des AR–Kellers beim zunächst provisorischen Anlegen des AR_f zum Aufruf c dienen:



Bemerkung 4.2 In dem Lemma 4.1 wurde ein GDV gemäß Definition 2.2 zugrundegelegt (siehe auch Bemerkung 2.10). In Kapitel 5 werden wir jedoch sehen, daß in Honschopp's Implementation [Ho83] *jeder* NSF–Aufruf auf einem linken Ast eine HOpt verhindert! Modifikationen an der Implementierung bringen dann aber einen definitionsgemäßen GDV–Verweis zurück.

Im nächsten Lemma verhindert der Verweis in der Linkage „ein dynamisches Niveau des statischen Vorgängers“ eine HOpt:

Lemma 4.2 Sei $c = f(\ldots)$ der Aufruf einer NSF f, und sei $c' = g(\ldots)$ ein Aufruf vom statischen Vorgänger von f. Befindet sich das AR_g des Aufrufs c' an der AR–Kellerspitze, so bewirkt c *keine* HOpt.

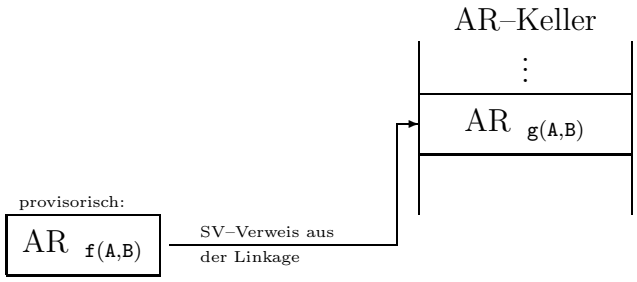
Beweis (Lemma 4.2): Klar, da durch einen NSF–Aufruf der zugehörige statische Vorgänger nicht gelöscht werden darf (Erhaltung der statischen Verweiskette). □

Beispiel 4.2

```
g = λ v w . { f = λ x y . { CONS(x,y) }
              f(v,w) }
g(A,B)
```

($f(v,w)$: c; $g(A,B)$: c')

Zur Veranschaulichung soll folgende Abbildung des AR–Keller beim zunächst provisorischen Anlegen des AR_f zum Aufruf c dienen:



In Lemma 4.3 verhindert der Verweis aus einer Parameterzelle eine HOpt:

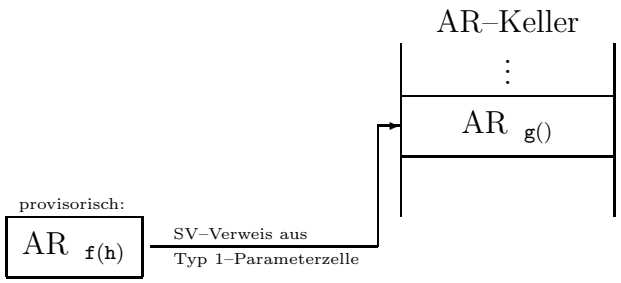
Lemma 4.3 Sei $c = f(\ldots)$ der Aufruf einer NSF f. Gibt es in c mindestens einen aktuellen Parameter mit funktionalem Wert h, d.h. vom Typ 1 oder Typ 2 gemäß Tabelle 2.1, und sei $c' = g(\ldots)$ ein Aufruf vom statischen Vorgänger von h und befindet sich ein AR_g vom Aufruf c' an der AR–Kellerspitze, so bewirkt c *keine* HOpt.

Beweis (Lemma 4.3): Klar, da das AR_g bei einem möglichen späteren Aufruf von h „ein dynamisches Niveau des statischen Vorgängers“ wäre und somit im Keller stehen bleiben muß, um eine gültige statische Verweiskette für die Auswertung von h zu sichern. □

Beispiel 4.3 Es verhindert ein Parameter vom Typ–1 beim Aufruf c eine HOpt:

```
f = λ x . { x(A) }
g = λ .   { h = λ y . { y }
            f(h) }        <- c
g()                       <- c'
```

Zur Veranschaulichung soll folgende Abbildung des AR–Kellers beim zunächst provisorischen Anlegen des AR_f zum Aufruf c dienen:



Beispiel 4.4 Es verhindert ein Parameter vom Typ–2 beim Aufruf c eine HOpt:

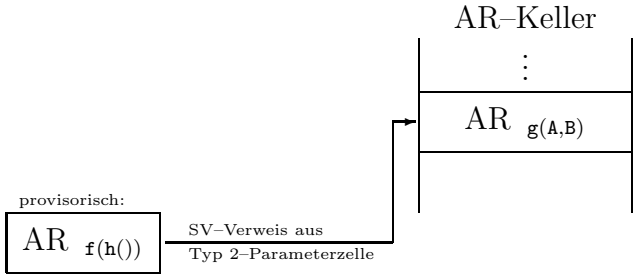
```
f = λ x .     { x }
g = λ v w . { f(CONS(v,w)) }   <- c

g(A,B)                         <- c'
```

Gemäß der Handhabung dicker Parameter (siehe Abschnitt 2.2) wird dieses Programm vom Compiler übersetzt wie:

```
f = λ x .    { x }
g = λ v w . { h = λ . { CONS(v,w) }
              f(h()) }
              ─────
                c
g(A,B)
──────
  c'
```

Zur Veranschaulichung soll folgende Abbildung des AR–Kellers beim zunächst provisorischen Anlegen des AR_f zum Aufruf c dienen:



Entsprechend der Definition 1.7 werden im Lemma 4.3 nur pending Parameter eingeschlossen, die in einer aktuellen– bzw. pending–Parameterliste des Aufrufs c vorkommen, d.h. unmittelbar dem Aufruf c „mitgegeben“ werden, wie z.B. bei $c = f(A)(B)$. Wie wir in Abschnitt 2.1 aber gesehen haben, werden bei der Neuanlage eines AR im AR–Keller die im unmittelbar zuvor angelegtem AR („dynamischer Vorgänger“) evtl. vorhandenen pending Parameter an das neuanzulegende AR weitergereicht. Der obige Aufruf $c = f(A)(B)$ könnte also z.B. zur Laufzeit zu einem Aufruf $c = f(A)(B)(C)$ vervollständigt werden.

Aus diesen „hinzukopierten Parametern“ können ebenfalls Verweise in den AR–Keller stattfinden. Hierzu folgender Hilfssatz:

Hilfssatz 4.1 Werden in der Situation in Lemma 4.3 in dem AR_f vom Aufruf c noch eventuell vorhandene pending Parameter vom dynamischen Vorgänger übernommen, so können die statischen Verweise aus diesen „hinzukopierten pending Parametern“ eine mögliche HOpt für c nicht verhindern.

Beweis (Hilfssatz 4.1): Wenn ein AR (zunächst provisorisch) angelegt wird, werden zunächst die aktuellen und die pending Parameter in die entsprechenden Zellen eingetragen. Sind im unmittelbar zuvor angelegtem AR pending Parameter eingetragen, so werden diese in das neu anzulegende AR hinter

die evtl. bereits vorhandenen Parameter kopiert (zwischen zwei pending–Parameterlisten wird dabei eine Trennmarke gesetzt).
⇒ Sollte gemäß der Situation in Lemma 4.3 ein hinzukopierter pending Parameter mit funktionalem Wert, d.h. vom Typ 1 oder Typ 2 gemäß Tabelle 2.1, einen statischen Verweis auf den dynamischen Vorgänger haben, so müßte der Verweis dieses Parameters im obersten AR selbst auf das eigene AR zeigen.

Da dies aber aufgrund der Definition 1.10 des statischen Vorgängers nicht möglich ist, und der Parameter lediglich in das neu anzulegende AR *kopiert* wird, ist der Hilfssatz bewiesen. □

Die obigen drei Lemmata und der Hilfssatz sind die Grundlage für den nun folgenden Satz:

Satz 4.1 Abgesehen von den drei Situationen in Lemma 4.1, Lemma 4.2 und Lemma 4.3 bewirkt jeder Aufruf $c = f(\ldots)$ einer NSF f eine HOpt.

Beweis (Satz 4.1) Sei o.B.d.A. das AR_f vom Aufruf $c = f(\ldots)$ einer NSF f zunächst provisorisch angelegt worden. Gemäß der Optimierung in [Ho83] ist die einzige Möglichkeit, daß das oberste AR im AR–Keller durch den Aufruf c nicht freigegeben wird, daß mindestens ein Verweis (siehe Bemerkung 2.8) aus dem AR_f auf das oberste AR im AR–Keller zeigt. Im folgenden sind gemäß der Implementation *alle* Speicherbereiche eines AR beschrieben (siehe auch Abbildung 2.2), aus denen ein Verweis auf bereits im AR–Keller vorhandene AR's stattfinden kann:

1. In der Situation in Lemma 4.1 zeigt der Verweis „GDV“ in der 1. Linkagezelle vom AR_f auf das oberste AR im AR–Keller.

2. In der Situation in Lemma 4.2 zeigt der Verweis „ein dynamisches Niveau des statischen Vorgängers“ in der 3. Linkagezelle vom AR_f auf das oberste AR im AR–Keller.

3. In der Situation in Lemma 4.3 zeigt mindestens ein Verweis in einer aktuellen oder pending Parameterzelle vom AR_f auf das oberste AR im AR–Keller.

4. Aus möglichen hinzukopierten pending Parameterzellen kann gemäß Hilfssatz 4.1 kein Verweis auf das oberste AR im AR–Keller erfolgen.

Zusammen mit dem Hilfssatz 4.1 folgt also: Ist keine der drei Situationen von Lemmata 4.1 bis 4.3 gegeben, so zeigt kein Verweis vom AR_f auf das oberste AR im AR–Keller, und mindestens dieses wird bei der Anlage des zunächst provisorisch angelegten AR_f im AR–Keller freigegeben, d.h. eine HOpt wird durchgeführt. □

Definition 4.2 Die Menge aller NSF–Aufrufe, die durch das neue Laufzeitsystem nach Honschopp [Ho83] eine HOpt bewirken, wird als *HOpt–Klasse* bezeichnet.

In den obigen Lemmata haben wir uns darauf beschränkt, diejenigen *dynamischen* Aufruf–Situationen zu beschreiben, in denen keine HOpt stattfindet, wobei dies entsprechend Satz 4.1 alle Möglichkeiten sind. Entsprechend der dynamischen Natur der Optimierung ist diese Vorgehensweise auch gegeben.

Im folgenden wollen wir für *Vorkommen* von NSF–Aufrufen $c = f(\ldots)$ untersuchen, ob c dynamisch eine HOpt bewirkt. Unter Berücksichtigung der Handhabung dicker Parameter vom Compiler (siehe Abschnitt 2.2) werden wir dann sehen, daß für *jedes* Vorkommen eines NSF–Aufrufs $c = f(\ldots)$ in einem echten LISP/N–Programm Π (d.h. Π sei gemäß Definition 1.27 übersetzbar, aber gemäß Definition 2.1 noch nicht übersetzt) *statisch* entschieden werden kann, ob c *dynamisch* im Falle einer Ausführung eine HOpt bewirkt.

In einem vom Compiler übersetzten LISP/N–Programm Π' kann es nur noch die NSF–Aufrufe $h()$ der vom Compiler eingeführten Hilfsfunktionen h für dicke Parameter als aktuelle Parameter eines NSF–Aufrufs geben (siehe Bemerkung 2.2). Aufgrund der Call By Name–Semantik des Laufzeitsystems kann für diese Aufrufe $h()$ im allgemeinen statisch *nicht* über eine HOpt zur Laufzeit entschieden werden:

Lemma 4.4 Sei $c = h()$ der Aufruf einer vom Compiler für einen dicken Parameter eingerichteten Nichtstandard–Hilfsfunktion h in einem übesetzten Programm Π', d.h. c ist ein aktueller Parameter eines NSF–Aufrufs c', also $c' = f(\ldots, c, \ldots)$. Dann ist im allgemeinen *statisch* nicht entscheidbar, ob c eine HOpt bewirkt.

Beweis (Lemma 4.4): Wir geben ein Beispielprogramm an, in welchem es von der Eingabe abhängt, ob c auf einem linken Ast ausgeführt werden muß oder nicht. Dabei nehmen wir der Einfachheit halber den Aufruf der SF CDR als dicken Parameter, anstatt eines NSF–Aufrufs. Der Parameter IN veranlaßt das Laufzeitsystem einen S–Ausdruck über die Standardeingabe (z.B. Tastatur) einzugeben:

```
f = λ x y . { IF ATOM(x) THEN CONS(y,D)
                         ELSE y
              FI }
f(IN,CDR((B C)))
```

wird vom Compiler übersetzt wie (siehe Abschnitt 2.2):

```
h = λ . { CDR((B C)) }
f = λ x y . { IF ATOM(x) THEN CONS(y,D)
                         ELSE y
              FI }
f(IN,h())
```

Der NSF–Aufruf $c = h()$ wird bei Eingabe eines Atoms auf dem linken Ast von CONS ausgeführt und das AR_f durch einen GDV–Verweis aus dem AR_h nicht freigegeben. Wird jedoch eine Liste eingegeben, so wird der else–Teil des Konditionals abgearbeitet und c ist ein rechter Ast–Aufruf — das AR_f kann frühzeitig freigegeben werden. Wir sehen also, daß statisch nicht über eine HOpt des Hilfsfunktionsaufrufs c entschieden werden kann. □

Aufgrund Lemma 4.4 beschränken wir uns im folgenden auf Vorkommen von NSF–Aufrufen $c = f(\ldots)$ in einem *echten* und somit insbesondere *nicht übersetzten* LISP/N–Programmen Π. Die folgenden Aussagen zu statischen Aufrufsituationen berücksichtigen dann die notwendige Tatsache, daß das Laufzeitsystem nur übersetzte Programme ausführt.

Zunächst wenden wir uns der Situation in Lemma 4.1 zu:

Satz 4.2 Sei $c = f(\ldots)$ der Aufruf einer NSF f in einem echten LISP/N–Programm Π. Beim Aufruf c wird durch einen GDV–Verweis eine HOpt verhindert $\Longleftrightarrow$ einer der beiden folgenden Punkte ist erfüllt:

1. c kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* c kommt innerhalb eines linken Astes vor, *oder*

2. c kommt innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* c kommt innerhalb des kleinsten c umfassenden aktuellen Parameters eines NSF–Aufrufs auf einem linken Ast vor.

Beweis (Satz 4.2): „$\Longleftarrow$“: Sei Π' das übersetzte LISP/N–Programm zu Π. In Π' kann c nicht mehr innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen: Bei 1. ist dies schon in Π gegeben, und bei 2. ist gemäß Abschnitt 2.2 der dicke Parameter, in dem c in Π vorkommt, vom Compiler so übersetzt worden, als komme er in Π' im Anweisungsteil einer Hilfsfunktion vor.

Sei o.B.d.A. g die kleinste c umfassende NSF in Π' (wobei g auch das Hauptprogramm bzw. im Falle 2. eine Hilfsfunktion sein kann). Da c also im Anweisungsteil von g vorkommt, ist g vor dem Aufruf c schon mindestens einmal aufgerufen worden, wobei $c' = g(\ldots)$ der *letzte* Aufruf von g sei. Dann reicht zu zeigen, daß

I. das $AR_{c'}$ der GDV des Aufrufs c ist und

II. das $AR_{c'}$ beim Aufruf c an der AR–Kellerspitze steht.

Zu I.: Klar, da gemäß Definition 2.2 das $AR_{c'}$ zum Aufruf der NSF g der GDV für den linken Ast–Aufruf c ist.

Zu II.: Da c in dem übersetzten LISP/N–Programm Π' nicht innerhalb eines aktuellen Parameters eines anderen NSF–Aufrufs vorkommt, besteht der Anweisungsteil von g lediglich aus (ggf. geschachtelten) SF–Aufrufen oder/und

Konditionalen. Falls c der erste NSF–Aufruf im Anweisungsteil von g ist, ist klar, daß das $AR_{c'}$ noch an der AR–Kellerspitze stehen muß. Ist c nicht der erste NSF–Aufruf im Anweisungsteil von g, so müssen die zuvor ausgeführten NSF–Aufrufe $\tilde{c}_1, \ldots, \tilde{c}_n$ mit $n \geq 1$ ebenfalls auf linken Ästen vorkommen und das $AR_{c'}$ als GDV ausweisen. Da c nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs und somit insbesondere nicht als aktueller Parameter von $\tilde{c}_1, \ldots, \tilde{c}_n$ vorkommt, sind die $\tilde{c}_1, \ldots, \tilde{c}_n$ *vor* dem Aufruf c beendet, und das $AR_{c'}$ ist aufgrund der GDV–Handhabung wieder zum aktuell obersten AR erklärt worden.

„$\Longrightarrow$“: Zu zeigen ist, daß 1. und 2. *alle* Möglichkeiten sind, durch einen GDV–Verweis eine HOpt zu verhindern. Dies ist aber klar, da c zum einen auf einem linken Ast vorkommen muß, damit die GDV–Situation überhaupt gegeben ist, und zum anderen sichergestellt werden muß, daß das $AR_{c'}$ (siehe „$\Longleftarrow$“) noch an der AR–Kellerspitze steht. Falls c in Π im Anweisungsteil von g innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen würde, so käme c im übersetzten Programm Π' im Anweisungsteil einer Hilfsfunktion h vor und das $AR_{c'}$ wäre beim Aufruf c nicht mehr das oberste AR im AR–Keller. □

Die möglichen Situationen von Lemma 4.2 grenzen wir wie folgt ein:

Satz 4.3 Sei $c = f(\ldots)$ der Aufruf einer NSF f in einem echten LISP/N–Programm Π, und g sei die kleinste, den Aufruf c umfassende NSF (wobei g auch das Hauptprogramm sein kann). Beim Aufruf c wird durch einen Verweis auf einen statischen Vorgänger von f eine HOpt verhindert $\Longleftrightarrow$ die beiden folgenden Punkte sind erfüllt:

1. c kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, und
2. es gilt $\mathrm{SN}(f)=\mathrm{SN}(g)+1$.

Beweis (Satz 4.3): „$\Longleftarrow$“: Da c im Anweisungsteil von g vorkommt, ist g vor dem Aufruf c schon mindestens einmal aufgerufen worden, wobei $c' = g(\ldots)$ der *letzte* Aufruf von g sei. Dann reicht zu zeigen, daß

I. aus dem AR_c auf das $AR_{c'}$ verwiesen wird und

II. das $AR_{c'}$ beim Aufruf c an der AR–Kellerspitze steht.

Zu I.: Dies ist klar, da wegen $\mathrm{SN}(f)=\mathrm{SN}(g)+1$ die NSF g der statische Vorgänger von f ist, und deshalb f im Funktions–Deklarationsteil von g deklarierend vorkommen muß — andernfalls wäre f über die statische Verweiskette nicht erreichbar und ein Fehler läge vor. Aus der Linkage des AR_c wird somit auf das $AR_{c'}$ zum Aufruf von g als statischer Vorgänger verwiesen.

Zu II.: Da c nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommt, besteht der Anweisungsteil von g entweder

i. aus c selbst, oder

ii. c kommt in (ggf. geschachtelten) SF–Aufrufen oder/und Konditionalen vor.

Im Falle i. ist c der erste (und einzige) NSF–Aufruf im Anweisungsteil von g, und das $AR_{c'}$ muß noch an der AR–Kellerspitze stehen. Im Falle ii. folgt dies aus dem gleichen Grund wie bei „Zu II.“ im Beweis von Satz 4.2.

„$\Longrightarrow$“: Es ist zu zeigen, daß die Gültigkeit von 1. und 2. die *einzige* Möglichkeit ist, beim Aufruf c durch einen Verweis auf den statischen Vorgänger von f eine HOpt zu verhindern:

Dies ist aber klar, da zum einen aufgrund Definition 1.11 die NSF g nur der statische Vorgänger von f sein kann, falls $SN(f)=SN(g)+1$ gilt und zum anderen sichergestellt werden muß, daß das $AR_{c'}$ (siehe „$\Longleftarrow$“) noch an der AR–Kellerspitze steht. Falls c in Π im Anweisungsteil von g innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen würde, so käme c im übersetzten Programm Π' im Anweisungsteil einer Hilfsfunktion h vor und das $AR_{c'}$ wäre beim Aufruf c nicht mehr das oberste AR im AR–Keller. □

Im folgenden Satz grenzen wir die Möglichkeiten ein, in denen durch die Situation in Lemma 4.3 eine HOpt verhindert werden kann:

Satz 4.4 Sei $c = f(\ldots)$ der Aufruf einer NSF f mit $n \geq 1$ aktuellen Parametern $a_1, \ldots, a_n$ (wobei nach Definition 1.7 und für $1 \leq i \leq n$ die a_i auch in einer pending Parameterliste von c vorkommen können) in einem echten LISP/N–Programm Π, und sei g die kleinste c umfassende NSF (wobei g auch das Hauptprogramm sein kann).

Beim Aufruf c wird durch einen aktuellen Parameter a_i von c eine HOpt verhindert $\Longleftrightarrow$ c kommt nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* einer der beiden folgenden Punkte ist erfüllt:

1. a_i ist ein gewöhnlicher Funktionsausdruck, d.h. lediglich der Funktionsidentifikator einer NSF h mit $SN(h)=SN(g)+1$, *oder*

2. a_i ist ein dicker Parameter.

Beweis (Satz 4.4): „$\Longleftarrow$“: Da c im Anweisungsteil von g vorkommt, ist g vor dem Aufruf c schon mindestens einmal aufgerufen worden, wobei $c' = g(\ldots)$ der *letzte* Aufruf von g sei. Dann reicht zu zeigen, daß

I. aus dem AR_c auf das $AR_{c'}$ verwiesen wird, und

II. das $AR_{c'}$ beim Aufruf c an der AR–Kellerspitze steht.

Zu I.: Im Falle 1. ist wegen $SN(h)=SN(g)+1$ die NSF g der statische Vorgänger von h, und deshalb muß f im Funktions–Deklarationsteil von g

deklarierend vorkommen — andernfalls wäre h über die statische Verweiskette nicht erreichbar und ein Fehler läge vor. Aus der Typ 1–Parameterzelle zu a_i wird somit auf das $AR_{c'}$ verwiesen.

Im Falle 2. wird gemäß der Handhabung dicker Parameter (siehe Abschnitt 2.2) für a_i eine Hilfsfunktion h eingeführt, die im Funktions–Deklarationsteil von g deklarierend vorkommt. Somit ist g der statische Vorgänger zur Hilfsfunktion h, und der Typ 2–Parameter a_i enthält einen statischen Verweis auf das $AR_{c'}$.

Zu II.: Da c nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommt, besteht der Anweisungsteil von g entweder

i. aus c selbst, oder

ii. c kommt in (ggf. geschachtelten) SF–Aufrufen oder/und Konditionalen vor.

Im Falle i. ist c der erste (und einzige) NSF–Aufruf im Anweisungsteil von g und das $AR_{c'}$ muß noch an der AR–Kellerspitze stehen. Im Falle ii. folgt dies aus dem gleichen Grund wie bei „Zu II.“ im Beweis von Satz 4.2.

Aus I. und II. folgt, daß ein statischer Verweis aus der für a_i angelegten Parameterzelle eine HOpt verhindert.

„$\Longrightarrow$“: Es ist zu zeigen, daß dies *alle* Möglichkeiten sind, durch einen aktuellen Parameter a_i eine HOpt zu verhindern:

Falls c in Π im Anweisungsteil von g innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen würde, so käme c im übersetzten Programm Π' im Anweisungsteil einer Hilfsfunktion h vor und das $AR_{c'}$ wäre beim Aufruf c nicht mehr das oberste AR im AR–Keller.
$\Rightarrow$ c kann nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen.

Bleibt zu zeigen, daß 1. oder 2. gelten muß: Dazu gehen wir gemäß dem Produktionensystem P alle Möglichkeiten für einen aktuellen Parameter a_i durch:

- Ein aktueller Parameter a_i, der sich gemäß P auf <S-Ausdruck> oder IN reduzieren läßt, kann eine HOpt nicht verhindern, da aus der entsprechenden Typ 0–Parameterzelle (siehe Tabelle 2.1) kein statischer Verweis berücksichtigt werden braucht.

- Läßt sich a_i auf <Standardfunktion> reduzieren, so liegt der Funktionsidentifikator einer SF vor und der statische Verweis aus der zugehörigen Typ 1–Parameterzelle enthält immer einen Verweis auf das erste AR im AR–Keller — eine HOpt kann durch a_i nicht verhindert werden.

– Läßt sich a_i jedoch auf <Applikation> oder <Konditional> reduzieren, so ist a_i per definitionem ein dicker Parameter und die Vorraussetzungen für 2. sind gegeben. $\surd$

– Die letzte Möglichkeit ist, daß sich a_i auf <Nichtstandardidentifikator> reduzieren läßt. Dann kann a_i entweder

  (a) der Identifikator eines formalen Parameters oder

  (b) der Funktionsidentifikator einer NSF sein.

  Im Fall (a) ist der Wert von a_i in einem AR im AR–Keller gebunden und wird in das AR_c für den Aufruf c kopiert. Falls ein Parameter vom Typ 1 oder Typ 2 geschrieben wird, so kann der übernommene statische Verweis dieses Parameters eine HOpt nicht verhindern: Im „Extremfall“ ist der formale Parameter im $AR_{c'}$ gebunden, aber da ein Verweis auf einen statischen Vorgänger nicht auf sich selbst zeigen kann und lediglich kopiert wird (vgl. Beweis zu Hilfssatz 4.1), kann eine HOpt nicht verhindert werden.

  Im Fall (b) ist a_i der Funktionsidentifikator einer NSF h und somit vom Typ 1. Damit g der statische Vorgänger zur NSF h sein kann, muß $\mathrm{SN}(h)=\mathrm{SN}(g)+1$ gelten. $\surd$ □

Wir fassen unsere bisherigen Ergebnisse in folgendem Satz zusammen:

Satz 4.5 Sei $c = f(\ldots)$ der Aufruf einer NSF f mit $n \geq 1$ aktuellen Parametern $a_1, \ldots, a_n$ (wobei nach Definition 1.7 und für $1 \leq i \leq n$ die a_i auch in einer pending Parameterliste von c vorkommen können) in einem echten LISP/N–Programm Π, und g sei die kleinste den Aufruf c umfassende NSF (wobei g auch das Hauptprogramm sein kann). Der Aufruf c bewirkt *keine* HOpt $\Longleftrightarrow$ mindestens eine der folgenden drei Aussagen ist erfüllt:

1. einer der beiden folgenden Punkte ist erfüllt:

   (a) c kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* c kommt innerhalb eines linken Astes vor, *oder*

   (b) c kommt innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* c kommt innerhalb des kleinsten c umfassenden aktuellen Parameters eines NSF–Aufrufs auf einem linken Ast vor.

2. die beiden folgenden Punkte sind erfüllt:

   (a) c kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, und

   (b) es gilt $\mathrm{SN}(f)=\mathrm{SN}(g)+1$.

3. c kommt nicht innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor, *und* für mindestens einen aktuellen Parameter a_i von c ist einer der beiden folgenden Punkte erfüllt:

   (a) a_i ist ein gewöhnlicher Funktionsausdruck, d.h. lediglich der Funktionsidentifikator einer NSF h mit SN(h)=SN(g) + 1, *oder*

   (b) a_i ist ein dicker Parameter.

Beweis (Satz 4.5): „$\Longleftarrow$“: 1. bis 3. folgen direkt den Sätzen 4.2 bis 4.4.

„$\Longrightarrow$“: Entsprechend Satz 4.1 beschreiben die Lemmata 4.1 bis 4.3 *alle* Möglichkeiten, eine HOpt zu verhindern. Da 1. bis 3. zur Laufzeit den Situationen in Lemmata 4.1 bis 4.3 entspricht, muß mindestens einer der Punkte 1 bis 3 erfüllt sein, um eine HOpt zu verhindern. □

Bemerkung 4.3 Falls in Satz 4.5 $c = x(\ldots)$ der *formale* Aufruf einer NSF f ist, so kann Punkt 2 weggelassen werden, da f nicht deklarierend im Funktions–Deklarationsteil im Rumpf von g vorkommen kann.

Bemerkung 4.4 In Lemma 4.4 haben wir gesehen, daß für die Aufrufe $h()$ der vom Compiler erzeugten Hilfsfunktionen h statisch über eine HOpt zur Laufzeit nicht entschieden werden kann. In den Voraussetzungen zu Satz 4.5 werden aber die Aufrufe von Hilfsfunktionen ausgeschlossen, da in einem echten und insbesondere nicht übersetzten LISP/N–Programm noch keine Hilfsfunktionen vorkommen können.

Nach Satz 4.5 ist daher für *jedes* Vorkommen eines NSF–Aufrufs $c = f(\ldots)$ in einem echten LISP/N–Programm Π statisch entscheidbar, ob c dynamisch (natürlich nur im Falle einer Ausführung!) eine HOpt bewirkt.

Bemerkung 4.5 Die Aussage in Bemerkung 4.4 können wir auch wie folgt formulieren: Die Menge aller NSF–Aufrufe in einem echten LISP/N–Programm Π, die (im Falle der Ausführung) der HOpt–Klasse angehören, läßt sich *statisch* bestimmen!

Bemerkung 4.6 Ob ein NSF–Aufruf c zur Laufzeit ausgeführt werden wird, läßt sich im allgemeinen statisch nicht entscheiden (siehe auch Bemerkung 3.3 zur „SSB–relevanz“).

Bisher haben wir nur solche Funktionsaufrufe beschrieben, durch die eine HOpt *verhindert* wird. Unter gewissen Voraussetzungen können wir jedoch Klassen von Funktionsaufrufen angeben, in denen jeder Aufruf eine HOpt *bewirkt*: Abgesehen von Punkt 3 enthält der Satz 4.5 die syntaktischen Voraussetzungen, durch die einige aus der Literartur bereits bekannte Klassen von Funktionsaufrufen charakterisiert werden:

Definition 4.3 Sei Π ein syntaktisches Programm und $c = f(\ldots)$ ein Aufruf der NSF f in Π, wobei (f, k) das Vorkommen des Callers von c ist, und sei g die kleinste den Aufruf c umfassende NSF. Dann heißt c

1. *rekursiv* $\Longleftrightarrow_{def}$ (f, k) ist ein angewandtes Vorkommen des Funktionsidentifikators von g.

2. *postrekursiv* $\Longleftrightarrow_{def}$ c ist rekursiv und befindet sich auf einer *Postposition*, d.h. folgende beiden Bedingungen sind erfüllt:

   (a) es gibt keinen Aufruf im Anweisungsteil von g, der c umfaßt, und

   (b) es gibt keinen if–Teil im Anweisungsteil von g, der c umfaßt.

3. *verdeckt postrekursiv* $\Longleftrightarrow_{def}$ c ist rekursiv und befindet sich auf einer *verdeckten Postposition*, d.h. die folgenden beiden Bedingungen sind erfüllt:

   (a) es gibt mindestens einen Aufruf in Anweisungsteil von g, der c umfaßt, und für jeden derartigen Aufruf $c' = f(a_1, \ldots, a_n)$ gilt:

      i. f ist eine SF f_s, und

      ii. c ist enthalten in a_n.

   (b) es gibt keinen if–Teil im Anweisungsteil von g, der c umfaßt.

4. *total verdeckt postrekursiv* $\Longleftrightarrow_{def}$ c ist rekursiv und befindet sich auf einer *total verdeckten Postposition*, d.h. die folgenden beiden Bedingungen sind erfüllt:

   (a) es gibt mindestens einen Aufruf in Anweisungsteil von g, der c umfaßt, und für jeden derartigen Aufruf $c' = f(a_1, \ldots, a_n)$ gilt:

      i. f ist eine NSF, und

      ii. c ist enthalten in a_n.

   (b) es gibt keinen if–Teil im Anweisungsteil von g, der c umfaßt.

Bemerkung 4.7 Wir sehen also, daß jeder NSF–Aufruf $c = f(\ldots)$, der auf einer (verdeckten) Postposition vorkommt, *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommen kann. Aufgrund Bemerkung 2.2 können in einem *übersetzten* Programm auf total verdeckten Postpositionen nur noch Aufrufe der vom Compiler eingeführten Hilfsfunktionen vorkommen (siehe Abschnitt 2.2).

Wenn wir im folgenden die Schreibweise c ist ((total) verdeckt) postrekursiv benutzen, so ist c entweder postrekursiv, verdeckt postrekursiv oder total verdeckt postrekursiv. Das gleiche gilt für ((total) verdeckte) Postpositionen.

Satz 4.6 Sei $c = f(\ldots)$ ein ((total) verdeckt) postrekursiver Aufruf der NSF f mit $n \geq 0$ aktuellen Parametern in einem echten LISP/N–Programm Π. Falls für keinen aktuellen Parameter a_i von c mit $0 \leq i \leq n$ die Punkte (3.a) oder (3.b) aus Satz 4.5 zutreffen, so bewirkt c eine HOpt.

Beweis (Satz 4.6): Da c in Π auf einer ((total) verdeckten) Postposition vorkommt, kann c nicht innerhalb eines linken Astes vorkommen, d.h. die Situation gemäß Punkt 1 aus Satz 4.5 ist nicht gegeben.

Da c ein rekursiver Aufruf ist, also insbesondere nicht der Aufruf einer im Funktions–Deklarationsteil von f deklarierten NSF ist, ist auch die Situation gemäß Punkt 2 aus Satz 4.5 nicht gegeben.

Nach Voraussetzung sind die Situationen der Punkte (3.a) und (3.b) nicht gegeben, und somit bewirkt c nach Satz 4.5 eine HOpt. □

Definition 4.4 Sei Π ein syntaktisches Programm, und für $i \in \{1, 2\}$ sei $c_i = f_i(\ldots)$ ein NSF–Aufruf in g_i, wobei g_i die kleinste c_i umfassende NSF ist und $g_1 \neq g_2$ gilt. (f_i, k_i) seien die angewandten Vorkommen des Callers von c_i. Dann heißen c_1 und c_2

1. *wechselseitig rekursiv* $\Longleftrightarrow_{def}$
   - (f_1, k_1) ist ein angewandtes Vorkommen des Funktionsidentifikators von g_2 und
   - (f_2, k_2) ist ein angewandtes Vorkommen des Funktionsidentifikators von g_1.
2. *wechselseitig (verdeckt) postrekursiv* $\Longleftrightarrow_{def}$ c_1 und c_2 sind wechselseitig rekursiv und befinden sich auf (verdeckten) Postpositionen.

Bemerkung 4.8 In einem gebundenen Programm Π können zwei NSF–Aufrufe c_1 und c_2 nur dann wechselseitig (verdeckt) rekursiv sein, wenn die kleinsten c_i umfassenden NSF'en g_i den *gleichen* statischen Vorgänger haben.

Satz 4.7 Seien $c_j = f_j(\ldots)$ mit $j \in \{1, 2\}$ wechselseitig (verdeckt) postrekursive Aufrufe der NSF f_j mit jeweils $n_j \geq 0$ aktuellen Parametern $a_{1_j}, \ldots, a_{n_j}$ in einem echten LISP/N–Programm. Falls für keinen aktuellen Parameter a_{i_j} von c_j mit $0 \leq i \leq n_j$ die Punkte (3.a) oder (3.b) aus Satz 4.5 zutreffen, so bewirkt c_j eine HOpt.

Beweis (Satz 4.7): Entsprechend dem Beweis zu Satz 4.6, nur das die c_j aufgrund der wechselseitigen Rekursivität und Bemerkung 4.8 nicht die Aufrufe von im Funktions–Deklarationsteil von f_j deklarierten NSF sein können. □

Ab Kapitel 6 werden wir sehen, daß neben den Aufrufen, die der HOpt–Klasse angehören, noch eine Fülle weiterer NSF–Aufrufe optimiert werden können. Insbesondere werden die Situationen der Lemmata 4.1 und 4.3 sowie die Punkte (1) und (3.b) aus Satz 4.5 nicht mehr notwendig eine frühzeitige Speicherplazfreigabe verhindern.

4.2 Ein Vergleich zur SKGI–Optimierung

In diesem Abschnitt wollen wir kurz die Optimierungstechniken von Felgentreu und Honschopp vergleichen. Dabei werden wir zunächst einige Unterschiede und Gemeinsamkeiten der beiden Methoden aufführen und anschließend die Klassen der optimierten Aufrufe gegenüberstellen. Dabei werden wir sehen, daß die jeweilige Optimierungsmethode sehr eng mit der verwendeten Implementierungstechnik verbunden ist und durch die zusätzlich noch

verschiedenen Auswertungs–Strategien (Call By Name / Call By Value) ein Vergleich beider Klassen in dieser Form schwierig ist.

Wir führen die jeweiligen Aspekte in den folgenden Punkten vor:

- Die Optimierungsmethode nach Felgentreu beruht auf dem Standardisieren und anschließenden Markieren des Eingabeprogramms. Diese Maßnahm-en können bereits beim Einlesen des Eingabeprogramms durchgeführt werden. Zur Laufzeit stehen die notwendigen Optimierungsmaßnahmen dann bereits fest und können direkt ausgeführt werden. Sehen wir von der Einlesephase des zu interpretierenden Programms ab, so ist die Optimierung *statischer* Natur.

  Im Gegensatz dazu ist die Optimierungsmethode im Laufzeitsystem nach Honschopp rein *dynamischer* Natur: Über eine frühzeitige Speicherplatzfreigabe wird erst beim Anlegen eines AR in dem AR–Keller entschieden, indem alle von diesem AR ausgehenden Verweise auf den maximalen Wert hin untersucht werden. Ist kein Verweis auf den dynamischen Vorgänger, d.h. das unmittelbar zuvor angelegte AR, vorhanden, so wird eine HOpt durchgeführt. In diesem Fall wird dann das zunächst provisorisch angelegte AR an einen günstigereren (tieferen) Platz im AR–Keller kopiert.

- Wir haben im Abschnitt 3.5 gesehen, daß die Ausführung standardisierter und markierter LISP–Programme in der Regel zu geringen Laufzeitgewinnen führt.

  Wegen der rein dynamischen Struktur der Honschopp–Optimierung und trotz der im Laufe dieser Arbeit verringerten Kosten der Maßnahmen für eine HOpt (siehe Kapitel 8), kann durch eine HOpt *prinzipiell* keine Laufzeit eingespart werden!

- Was beide Optimierungstechniken jedoch gemeinsam auszeichnet, ist die oft frühzeitige Freigabe nicht mehr benötigten Speichers und damit verbunden der häufig erhebliche Speicherplatzgewinn. Die Gewinne beim Felgentreu–Interpretierer beruhen dabei auf dem vollständigen (stark SKGI) bzw. teilweisen (schwach SKGI) Verzicht auf die Maßnahmen M1 und M3, d.h. Retten und Wiederherstellen der alten Umgebung gemäß dem Shallow–Binding Konzept.

  Dabei ist die Maßnahme M1 vergleichbar mit der Anlage eines neuen AR an der AR–Kellerspitze: Dies impliziert nämlich ein „Retten“ der alten Umgebung, d.h. die zuvor angelegten AR's bleiben unverändert im Keller stehen. Falls die Maßnahme M1 beim Interpreter eingespart werden kann, so wird nur auf das Retten der Umgebung des unmittelbar zuvor ausgeführten NSF–Aufrufs verzichtet. Beim Laufzeitsystem jedoch wird im Falle einer HOpt *mindestens* das AR (d.h. die Umgebung) vom dynamischen Vorgänger freigegeben. Im günstigsten Falle

kann somit auf ein Retten *aller* zuvor angelegten AR's verzichtet werden, und das neu anzulegende AR würde am Kelleranfang direkt hinter dem AR vom Hauptprogramm angelegt werden.

Die Aktion M3, d.h. das Wiederherstellen der alten Umgebung, ist beim Laufzeitsystem in der Form nicht gegeben, da eine wieder benötigte Umgebung im Keller an unveränderter Position stehen geblieben ist.

- In Abschnitt 3.4 konnten einige von zum Teil aus der Literatur bereits bekannten Klassen von Aufrufen wiedergegeben werden, die durch die SKGI–Optimierung echt umfaßt werden. Alle diese Klassen haben gemeinsam, daß die Zugehörigkeit eines Aufrufs zu einer Klasse unabhängig von den möglichen aktuellen Parametern dieses Aufrufs ist — es ist lediglich die syntaktische Position des Aufrufs relevant: Wie wir gesehen haben, gehören z.B. alle Aufrufe auf Postpositionen, für die die Bedingung (3.a) des SKGI–Entscheidungsalgorithmus erfüllt ist, in trivialer Weise der Klasse der SKGI–Aufrufe an, weil der rlk_{Fe} dieser Aufrufe immer leer ist. Rekursive Aufrufe auf Postpositionen gehören *immer* der SKGI–Klasse an, da der rlk_{Fe} immer leer ist und die *Anzahl* der aktuellen Parameter immer mit der Stelligkeit der gerufenen Funktion übereinstimmt, also insbesondere nicht größer ist.

  Dagegen ist die vorbehaltlose Einordnung der (wechselseitig) (verdeckt) postrekursiven bzw. der ((total) vedeckt) postrekursiven Aufrufe in die HOpt–Klasse, wie wir in Abschnitt 4.1 gesehen haben, nicht möglich, da gemäß der Honschopp–Optimierung ein Verweis aus dem Parameterteil eines AR's eine HOpt verhindern kann. Der „Inhalt“ der aktuellen Parameter eines NSF–Aufrufs ist also für eine Optimierung relevant. Wir konnten diese Aufrufklassen der HOpt–Klasse daher nur mit gewissen „Auflagen“ an die aktuellen Parameter unterordnen (Ausschluß der Punkte (3.a) und (3.b) in Satz 4.5).

  Dafür ist die *Anzahl* der aktuellen Parameter völlig uninteressant für die Entscheidung über eine HOpt!

- Ein weiterer Unterschied der beiden Optimierungstechniken ist in der unterschiedlichen *Semantik* begründet: Aufgrund der Call By Value–Semantik im Interpreter liegt der zur Optimierung nötige RLK_{Fe} für jeden Aufruf in einem syntaktischem Programm bereits statisch fest. Auch wenn ein Aufruf innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommt, so wird er noch an gleicher syntaktischer Position ausgeführt und nur das Ergebnis an die gerufene Funktion übergeben. Es läßt sich also für *jeden* Aufruf bereits statisch entscheiden, ob zur Laufzeit optimiert werden kann.

  Im Call By Name–Laufzeitsystem läßt sich nur für jeden NSF–Aufruf in einem *echten* LISP/N–Programm statisch entscheiden, ob dynamisch eine HOpt stattfindet. Für die *übersetzten* LISP/N–Programme, die das

Laufzeitsystem dann aber tatsächlich ausführen muß, ist dies im allgemeinen statisch nicht mehr entscheidbar: Die Ausführung der möglichen Hilfsfunktions–Aufrufe kann an syntaktisch nicht mehr vorhersehbarer Position stattfinden und dadurch evtl. eine Optimierung verhindern (siehe Lemma 4.4).

Wir sehen also, daß neben den sehr unterschiedlichen Implementierungen (Interpreter mit Shallow Binding / Compiler und kellerartiges Laufzeitsystem) auch noch die verschiedenen Auswertungsstrategien (Call By Value / Call By Name) eine vergleichende Gegenüberstellung der optimierten Klassen von Funktionsaufrufen erschweren. Auch wenn die aus funktionaler Sicht vorzuziehende Strategie Call By Name (entspricht der Semantik des λ–Kalküls!) durch eine Call By Value–Semantik ersetzt würde, könnte weiterhin ein aktueller Parameter eines NSF–Aufrufs eine frühzeitige Freigabe des dynamischen Vorgängers verhindern: In Satz 4.5 würde zwar der Punkt (3.b) entfallen (weil dicke Parameter gemäß Call By Value *vor* Ausführung des eigentlichen Aufrufs ausgeführt würden und somit keine Typ 2–Parameter mehr als aktuelle Parameter an die gerufene NSF weitergereicht werden könnten), aber ein Funktionsausdruck wie in Punkt (3.a) könnte durch einen statischen Verweis weiterhin eine Optimierung verhindern.

Ohne „Auflagen“ an die aktuellen Parameter würde daher auch ein Call By Value–Laufzeitsystem (wechselseitig) (verdeckt) postrekursive bzw. ((total) verdeckt) postrekursive Aufrufe *nicht grundsätzlich* optimieren, wie dies beim Interpreter der Fall ist.

Wir werden in Kapitel 9 auf die Frage der Auswertungsstrategie noch näher eingehen.

Trotz der sehr unterschiedlichen Vorraussetzungen bei beiden Optimierungstechniken werden wir ab Kapitel 6 sehen, daß sich die bis dato rein dynamischen Optimierungskriterien nach Honschopp um statische Analysetechniken, ähnlich die der in Kapitel 3 vorgestellten LCC–Optimierung nach Felgentreu, ergänzen läßt, und dadurch viele weitere NSF–Aufrufe optimiert werden können, die der HOpt–Klasse nicht angehören.

Kapitel 5

Die unökonomische Auslegung der GDV–Definition

Gemäß der Definition 2.2 des „Generalisierten Dynamischen Vorgängers“ (siehe auch [HO83], S.53) soll durch den GDV–Linkage–Verweis von NSF–Aufrufen innerhalb eines linken Astes verhindert werden, daß dasjenige AR nicht frühzeitig freigegeben werden kann, welches zu dem Aufruf einer NSF gehört, in deren Anweisungsteil die betrachteten Aufrufe des linken Astes vorkommen. Dadurch soll sichergestellt werden, daß zu Beginn des rechten Astes die selben AR's im AR–Keller stehen wie zu Beginn des linken Astes und nicht durch die mögliche frühzeitige Speicherplatzfreigabe durch NSF–Aufrufe im linken Ast zu viele AR's gelöscht werden.

Untersuchungen an der Implementation von Honschopp [Ho83] haben jedoch ergeben, daß die Handhabung des GDV nicht definitionsgemäß erfolgt und dadurch oft unnötig viel Speicherplatz im AR–Keller beansprucht wird: *Jeder* NSF–Aufruf $c = f(\ldots)$ und *jeder* formale Nichtstandardidentifikator x, die innerhalb eines linken Astes vorkommen, werden vom Compiler mit einem „L“ markiert. Wird zur Laufzeit ein mit „L“ markierter NSF–Aufruf aufgerufen, oder ergibt die Auswertung eines mit „L“ markierten formalen Identifikators einen NSF–Aufruf, so wird jedesmal ein neuer GDV–Verweis auf den gerade aktuellen dynamischen Vorgänger eingeführt.

Dadurch existieren oft unnötig viele AR's als vermeintliche GDV's im Keller, und entsprechend der Maßnahme (4.) auf Seite 27 muß ein unnötiger zusätzlicher Verwaltungsaufwand betrieben werden.

Wir wollen diese unökonomische Implementation an folgendem Beispiel verdeutlichen:

Beispiel 5.1

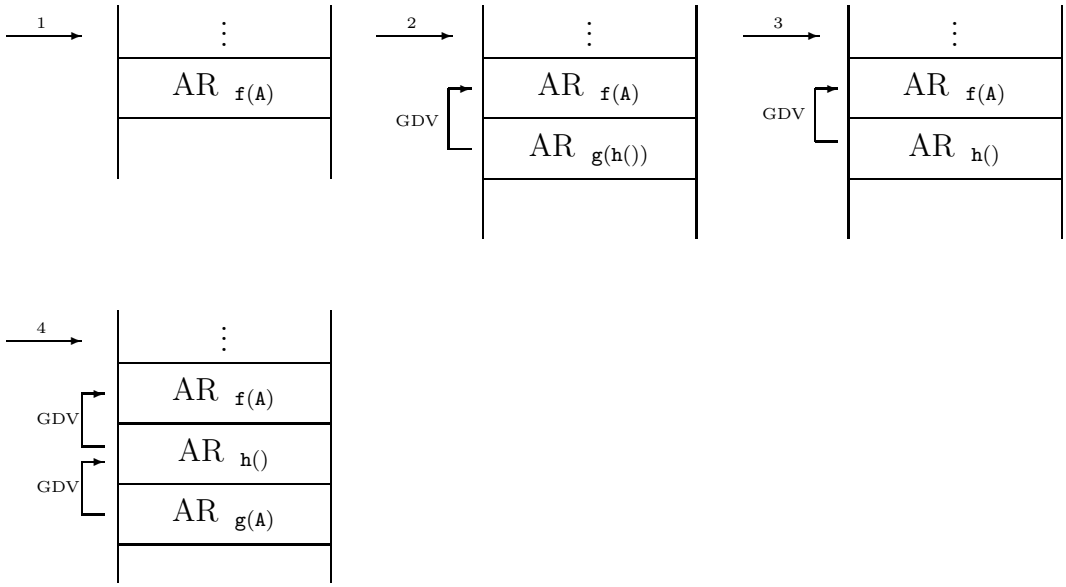
```
g = λ x . { x }
f = λ y . { CONS ( g„L“(g„L“(y„L“)) , y ) }
                  \_____linker Ast_____/
```

Aufgrund der Handhabung dicker Parameter (siehe Abschnitt 2.2) wird das Programm vom Compiler übersetzt wie:

```
g = λ x . { x }
f = λ y . { h = λ . { g„L“(y„L“) }
            CONS ( g„L“(h()) , y ) }
                   linker Ast
```

Dabei wird der Aufruf h() der Hilfsfunktion h zum dicken Parameter g(y) korrekterweise nicht mit einem „L“ versehen, weil durch den zuvor erfolgten NSF–Aufruf g(h()) schon der GDV–Verweis gesetzt wurde.

Der Ablauf im AR–Keller für den Aufruf f(A) würde damit wie folgt aussehen:



Per definitionem ist das $AR_{f(A)}$ der GDV für *alle* Aufrufe im linken Ast von CONS. Beim Aufruf g(h()) wird das $AR_{f(A)}$ zum GDV. Bei der Auswertung des formalen Identifikators x im Anweisungsteil von g wird die Hilfsfunktion h aufgerufen. Aus dem $AR_{h()}$ wird lediglich auf das $AR_{f(A)}$ als statischer Vorgänger verwiesen (siehe Abschnitt 2.2) und somit nach Übernahme des zuvor gültigen GDV–Verweises das $AR_{g(h())}$ frühzeitig gelöscht. Im 4. NSF–Aufruf wird der Anweisungsteil der Hilfsfunktion h, d.h. der ursprüngliche dicke Parameter g(y), ausgeführt. Da dieser Aufruf auch mit einem „L“ markiert wurde, wird der dynamische Vorgänger, d.h. das $AR_{h()}$, zum neuen GDV, obwohl das $AR_{f(A)}$ weiterhin der definitionsgemäße GDV wäre. Das $AR_{h()}$ wird somit unnötig lange im AR–Keller gehalten, was offensichtlich unökonomisch ist.

Bemerkung 5.1 Würde obiges Beispiel 5.1 nicht mit f(A), sondern z.B. mit f(CAR((A B))) aufgerufen, so würde für die Auswertung des mit „L“ markierten formalen Nichtstandardidentifikators y eine Hilfsfunktion für den

dicken Parameter CAR((A B)) aufgerufen werden und dafür ein weiterer GDV–Verweis erzeugt werden! In diesem Fall würden also schon zwei AR's gleichzeitig unnötig lange im Keller stehen bleiben.

Bemerkung 5.2 Die Implementation Honschopps stellt „auf jeden Fall" sicher, daß rechte Äste ein vollständiges Enviroment vorfinden und somit korrekt abgearbeitet werden können.

Im Korrektheitsbeweis von Kindler [Ki87] wird nur die Korrektheit der Implementation von Honschopp bewiesen und daher auch dort nicht der definitionsgemäße GDV benutzt.

5.1 Ein GDV gemäß Definition

In diesem Abschnitt wollen wir eine Methode beschreiben, die es erlaubt, mit relativ geringem Aufwand zu einer definitionsgemäßen Handhabung der GDV–Situation zu gelangen.

Dazu müssen wir dafür sorgen, daß innerhalb eines linken Astes nur der dynamische Vorgänger des *ersten* NSF–Aufrufs der GDV für die Auswertung des linken Astes sein darf. Alle weiteren Aufrufe, die ausgeführt werden, ohne daß gemäß Definition ein neuer GDV in Kraft tritt und ohne daß der Aufruf beendet ist, der zum Setzen des GDV führte, müssen so gehandhabt werden, als stünden sie nicht in einem linken Ast.

Insbesondere bedeutet dies, daß bei einer Schachtelung von NSF–Aufrufen in einem linken Ast (wie im obigen Beispiel) nur jeweils der äußerste Aufruf das Setzen des GDV auf den dynamischen Vorgänger bewirken darf. Die „inneren" Aufrufe werden dann genau wie rechter Ast–Aufrufe gehandhabt.

Da linke Äste selber inneinander verschachtelt vorkommen können und da durch die Call By Name–Semantik von LISP/N aktuelle Parameter von NSF–Aufrufen unausgewertet an die gerufene Funktion übergeben werden, wird der Begriff des „relevanten linken Astes" für die unten angegebene neue Handhabung der GDV–Situation eingeführt:

Definition 5.1 Sei Π ein syntaktisches Programm und (ℓ, k) ein linker Ast in einem Anweisungsteil. Der linke Ast ℓ heißt *relevanter linker Ast* (kurz: RLA), falls einer der beiden folgenden Punkte für ℓ erfüllt ist:

1. (a) (ℓ, k) kommt *nicht* in einem aktuellen Parameter eines NSF–Aufrufes vor und
   (b) (ℓ, k) ist bei einer möglichen Schachtelung von linken Ästen der äußerste linke Ast,

   oder

2. (a) (ℓ, k) kommt in einem aktuellen Parameter a_i eines NSF–Aufrufs vor, wobei a_i der kleinste (ℓ, k) echt umfassende aktuelle Parameter ist und

(b) (ℓ, k) ist bei einer möglichen Schachtelung von linken Ästen innerhalb von a_i der äußerste linke Ast.

Eine definitionsgemäße Handhabung des GDV wird nun wie folgt erreicht:

1. Beim Einritt in einen RLA wird eine globale Variable RLACALL (steht für „relevanter linker Ast Aufruf“) mit „nicht aktiv“ initialisiert, d.h. RLACALL:=false gesetzt. Beim erreichen von linken Ästen, die im Inneren eines RLA vorkommen, wird somit nichts gemacht.

2. Findet ein mit „L“ markierter NSF–Aufruf statt, bzw. führt die Auswertung eines mit „L“ markierten formalen Identifikators zu einem NSF–Aufruf, so wird unterschieden:

   (a) Gilt RLACALL=false, d.h. es ist noch kein linker Ast–Aufruf aktiv, so wird folgendes gemacht:

      i. Setze RLACALL:=true , sowie

      ii. in der 1. Linkagezelle „GDV–Verweis“ wird ein Verweis auf den dynamischen Vorgänger (wie bisher) *zusammen* mit der Rücksprungadresse (kurz: RA) des betreffenden Aufrufs geschrieben. Die RA protokolliert somit den für das Setzen des GDV verantwortlichen Aufruf.
      Also: 1. Linkagezelle GDV := RA | GDV–Verweis.

   (b) Gilt RLACALL=true, d.h. es ist noch ein linker Ast–Aufruf aktiv, so wird die 1. Linkage–Zelle (d.h. RA– und GDV–Verweis) vom dynamischen Vorgänger übernommen und somit der GDV weitergereicht. Es wird also genauso verfahren, als ob ein mit „R“ markierter Aufruf, d.h. ein Aufruf, der *nicht* auf einem linken Ast steht, stattfindet.

3. Wird ein mit „L“ markierter NSF–Aufruf beendet, bzw. derjenige NSF–Aufruf beendet, der für die Auswertung eines mit „L“ markierten formalen Identifikators aufgerufen wurde, und gilt:

   (a) Die RA vom Aufruf ist *gleich* der RA vom GDV, so ist derjenige Aufruf beendet, der für das Setzen des GDV verantwortlich war, und der zuvor gültige GDV wird wieder gültig. Ferner wird RLACALL:=false gesetzt, damit ein erneuter NSF–Aufruf auf dem linken Ast wieder den GDV setzen kann.

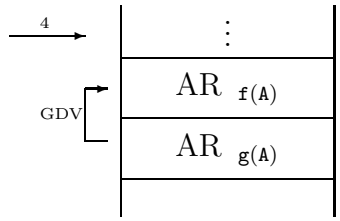
   (b) Die RA vom Aufruf ist *ungleich* der RA vom GDV, so bleiben der GDV sowie die globale Variable RLACALL unverändert.

4. Beim Austritt aus einem RLA wird RLACALL:=true gesetzt, damit bei der möglichen Rückkehr in einen linken Ast dort der GDV nicht erneut gesetzt werden kann. Beim Austritt aus linken Ästen, die innerhalb eines RLA vorkommen, wird also nichts gemacht.

Bemerkung 5.3 Die eingeführte ökonomische Handhabung der GDV–Definition, insbesondere die Benutzung einer *globalen* Variablen RLACALL, ist möglich, weil

- linke Äste sich nicht überlappen können,
- bei geschachtelten linken Ästen nur der RLA bei Ein– und Austritt besonders behandelt wird und
- die Rücksprungadresse für jedes Vorkommen eines NSF–Aufrufs in einem Programm *eindeutig* ist.

Bevor wir zur Relevanz der Korrektur kommen, betrachten wir den nun kostengünstigeren Ablauf von Beispiel 5.1 (die ersten drei Aufrufe bleiben unverändert und werden daher nicht erneut dargestellt):



Wir sehen, daß aus dem $AR_{g(A)}$ des „inneren“ Aufrufs von g nun kein Verweis mehr auf das $AR_{h()}$ des Aufrufs der dicker Parameter–Hilfsfunktion vorhanden ist, welches somit frühzeitig freigegeben werden kann. Der Speicherplatzgewinn ist offensichtlich.

5.2 Die Relevanz der Verbesserung

In dem obigen Beispiel haben wir gesehen, daß für die Abarbeitung des Aufrufs f(A) die „maximale Kellertiefe“ um die Länge des $AR_{h()}$ verringert wurde. Was dabei unter der maximalen Kellertiefe zu verstehen ist, zeigt folgende

Definition 5.2 Die *maximale Kellertiefe* für einen NSF–Aufruf c gibt die größte Anzahl von Speicherzellen im AR–Laufzeitkeller, d.h. den maximal angenommen Wert des Verweises BFS an, der bei der Abarbeitung von c erreicht wird. Der allgemeine Aufbau eines AR ist dabei der Abbildung 2.2 auf Seite 28 zu entnehmen.

Bemerkung 5.4 Die Angabe einer *durchschnittlichen Kellertiefe* hat so gut wie keine praktische Relevanz. Sie ist eine im wesentlichen zur Laufzeit einer Programmausführung redundante Information. Ab Kapitel 8 werden wir laufzeitrelevante Optimierungen vorstellen und deshalb auch laufzeitintensivere Beispiele verwenden. Dann werden die gemessenen Laufzeiten mit angegeben.

Die Einsparungen durch die effizientere GDV–Implementation vervielfachen sich natürlich, falls auf linken Ästen rekursive Aufrufe stehen. In der applikativen Programmierung kommen solche Konstellationen durchaus öfters vor. Hierzu zwei bekannte Beispiele und die dazu festgestellten Einsparungen:

Beispiel 5.2 Invertieren einer Liste durch Reverse:

```
reverse = λ x . { IF ATOM(x)
                    THEN x
                    ELSE CONS(reverse(CDR(x)),reverse(CAR(x)))
                  FI }
```

Für den Aufruf: reverse((A B C D E F G H)) ergibt sich:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	116
Definitonsgemäßer GDV	74

⇒ ca. 36% weniger AR–Keller–Bedarf.

Die Einsparungen im Beispiel 5.2 ergeben sich dadurch, daß für die Auswertungen des formalen Identifikators x im linken Ast von CONS die jeweilige erneute Einführung eines GDV–Verweises vermieden werden kann: Ohne Optimierung würde die Auswertung von x immer einen GDV–Verweis auf das $AR_{h()}$ der vom Compiler eingeführten Hilfsfunktion h zum dicken Parameter CDR(x) setzen, obwohl der definitionsgemäße GDV schon durch den umfassenden Aufruf von reverse(CDR(x)) gesetzt worden ist. Da diese Auswertungen rekursiv erfolgen, potenziert sich die Einsparung entsprechend.

Beispiel 5.3 Die Fibonacci–Rekursion auf Listenbasis:

```
fib = λ x . { IF ATOM(x)
                THEN x
                ELSE IF ATOM(CDR(x))
                       THEN x
                       ELSE CONS(fib(CDR(x)),fib(CDR(CDR(x))))
                     FI
              FI }
```

Für den Aufruf fib((I I I I I I I I I I I I)) ergibt sich:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	168
Definitonsgemäßer GDV	102

⇒ ca. 39% weniger AR–Keller–Bedarf.

Die Einsparungen im Beispiel 5.3 ergeben sich aus gleichem Grund wie im Beispiel zuvor, d.h. durch die Auswertungen von x im linken Ast von CONS.

Bemerkung 5.5 Die Optimierung benötigt dynamisch lediglich die Verwaltung der globalen Variable RLACALL sowie die zusätzliche Information der RA–Adresse in der GDV–Linkagezelle. Der zeitliche Aufwand zur Verwaltung dieser beiden Informationen kann vernachlässigt werden.

Dadurch jedoch, daß nun öfters zunächst provisorisch angelegte AR's verschoben werden müssen, bedeutet die Speicherplatz–Optimierung einen geringen zeitlichen Mehraufwand (wie übrigens jede frühzeitige Speicherplatzfreigabe im Rahmen der Honschopp–Optimierung): Für den obigen Aufruf von fib ist die Laufzeit etwa 4,7% langsamer (siehe Anhang A.1). Im allgemeinen stehen die Laufzeiteinbußen jedoch einem etwa fünfmal so hohem Speicherplatzgewinn gegenüber!

Später werden wir jedoch sehen, daß sich die Kosten für das Verschieben von AR's, d.h. für eine HOpt, stark verringern lassen (Kapitel 8) und durch weitere Maßnahmen auch enorme Laufzeitgewinne erzielt werden können (Kapitel 9).

Bemerkung 5.6 Die Auswertung weiterer Beispiele und die Dokumentation ihrer Einsparungen sind im Anhang A.1 aufgeführt.

Kapitel 6

Optimierung durch statische Programmanalyse

Der GDV–Verweis entstand aus dem Problem heraus, daß bei der Auswertung eines linken Astes evtl. Informationen gelöscht werden können, die für die weiteren Ausführungen nach Beendigung des linken Astes noch benötigt werden (siehe Kapitel 2).

Sehen wir einmal von der im vorherigen Abschnitt beseitigten Ineffizienz des GDV–Verweises ab, so kann während der Ausführungen im linken Ast das AR vom Aufruf der kleinsten, den linken Ast umfassenden Nichtstandardfunktion, *nicht* gelöscht werden. Für die möglichen Ausführungen im rechten Ast kann also keine Information gelöscht worden sein (vgl. Bemerkung 2.6).

Das rigorose Setzen des GDV–Verweises auf den dynamischen Vorgänger ist jedoch in vielen Fällen nicht nötig und somit ineffizient!

Unterstützt durch die Ergebnisse aus der Dissertation von Felgentreu ([Fe87]; siehe Kapitel 3) werden wir in diesem Kapitel eine Technik beschreiben, die es uns erlaubt, *statisch*, also bereits beim Übersetzen des Programmtextes, zu entscheiden, ob für einen NSF–Aufruf auf einem linken Ast überhaupt ein GDV nötig ist, und wenn er nötig ist, welche Informationen (d.h. welche AR's) im Keller durch einen GDV–Verweis geschützt werden müssen. Der hier vorgestelle GMARK–Markierungsalgorithmus wird die dazu nötigen Informationen während der Compilationsphase ermitteln und durch in das zu übersetzende Programm eingefügte Markierungen dem Compiler mitteilen, mit welchen Informationen das Programm weiter übersetzt werden soll. *Dynamisch*, d.h. zur Laufzeit, kann die Optimierung dann sehr effizient durchgeführt werden.

Damit wir durch unseren neuen, oft kostengünstigereren GDV–Verweis eine korrekte Implementation beibehalten, werden wir in diesem Kapitel noch zwei weitere Veränderungen am Honschopp–Compiler, bzw. –Laufzeitsystem vorstellen:

- Zum einen eine oft effizientere Handhabung von dicken Parametern: Durch eine *statische* Analyse des Eingabeprogramms, welche ebenfalls vom GMARK–Algorithmus durchgeführt werden wird, werden dicke Parameter mit einem „notwendigen statischen Niveau“ vom Compiler übersetzt und so dynamisch oft Speicherplatz gewonnen. Wir weden sehen, daß die hier beschriebene Technik schon eine eigenständige Optimierung darstellt, im Zusammenhang mit der GDV–Optimierung jedoch eine notwendige Maßnahme ist.
- Zum anderen wird für die GDV–Optimierung die Notwendigkeit eines eigenen Laufzeitkellers für die GDV–Verweise verdeutlicht.

6.1 GDV–Optimierung durch statische Programmanalyse

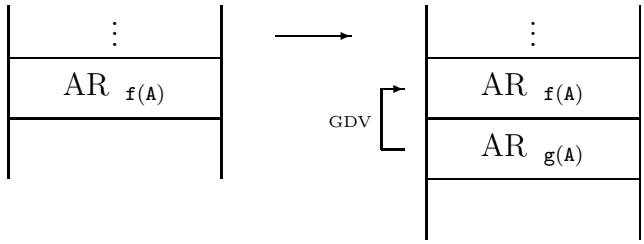
6.1.1 Ein einführendes Beispiel

Zunächst betrachten wir einmal ein motivierendes Beispiel, welches die häufige Ineffizienz des GDV–Verweises aufzeigt:

Beispiel 6.1 (vgl. Beispiel „gdv1.lsp“ im Anhang A.1)

```
g = λ x . { x }
f = λ y . { CONS ( g(y) , B ) }
f(A)
```

Für den NSF–Aufruf f(A) entsteht folgender Ablauf im AR–Keller:



Da der Aufruf g(A) auf dem linken Ast von CONS stattfindet, wird das $AR_{f(A)}$ durch einen GDV–Verweis im Keller gehalten. Der rechte Ast von CONS ist jedoch ein konstanter S–Ausdruck und benötigt keine Informationen aus dem $AR_{f(A)}$. Der GDV–Verweis auf das $AR_{f(A)}$ ist somit zur korrekten Abarbeitung des Aufrufs unnötig und verhindert eine frühzeitige Speicherplatzfreigabe!

6.1.2 Die LCC–Optimierung von Felgentreu

In Kapitel 3 haben wir die Techniken der Low Cost Call–Optimierung in Felgentreus Dissertation [Fe87] kurz beschrieben: Durch Analyse des syntaktischen Kontextes eines Aufrufs kann bereits statisch entschieden werden, ob dynamisch effiziente Maßnahmen zur Optimierung durchgeführt werden können:

Haben wir einen Aufruf c, so enthält nach Felgentreu gerade der *Relevante Lokale Kontext* von c (kurz: $\mathrm{RLK}_{Fe}(c)$, siehe Definition 3.2) alle Identifikatoren im *Lokalen Kontext* von c (kurz: $\mathrm{LK}_{Fe}(c)$, siehe Definition 3.2), auf die für die weitere Abarbeitung des $\mathrm{LK}_{Fe}(c)$ *nach* der Ausführung von c (die sogenannte „lokale Fortsetzung von c“ — vgl. Definition 6.4) noch zurückgegriffen werden könnte. Mit diesem Wissen können die entsprechend der Implementation von Static Scope Shallow Binding anfallenden Maßnahmen Retten und Wiederherstellen der alten Umgebeung (M1 und M3) häufig ganz oder zumindest teilweise (stark oder schwach SKGI) eingespart werden.

Diese statische LCC–Optimierungstechnik wollen wir nun auf das kellerartige Laufzeitsystem nach Honschopp übertragen, um zu einer effizienteren Handhabung der GDV– und dicker Parameter–Situation zu gelangen (letztere in Abschnitt 6.2).

6.1.3 Übertragung auf die GDV–Situation

Der LISP–Interpreter [Fe87] basiert aus Effizienzgründen auf einer Call By Value–Semantik. Für einen NSF–Aufruf bedeutet dies, daß die aktuellen Parameter ausgewertet der gerufenen Funktion übergeben werden. Sei $c = f(a_1, \ldots, a_n)$ ein Aufruf einer n–stelligen NSF f, dann umfaßt beispielsweise der RLK_{Fe} eines Aufrufs innerhalb von a_1 auch die restlichen aktuellen Parameter $a_2 \ldots a_n$, die gemäß Call By Value noch innerhalb des $\mathrm{LK}_{Fe}(c)$ ausgeführt werden müssen.

Das LISP/N–Laufzeitsystem nach Honschopp basiert aber auf der dem λ–Kalkül entsprechenden Semantik Call By Name und übergibt die aktuellen Parameter unausgewertet, d.h. textuell, der gerufenen Funktion. Für die obige Situation bedeutet dies, daß die aktuellen Parameter $a_1 \ldots a_n$ frühstens innerhalb des Anweisungsteils der gerufenen Funktion ausgewertet werden. In dem Anweisungsteil der gerufenen Funktion liegt dann, beispielweise für die Position der Auswertung des ursprünglichen Parameters a_1, ein anderer lokaler Kontext und somit auch ein anderer relevanter lokaler Kontext vor, in dem die weiteren Parameter $a_2, \ldots, a_n$ nicht mehr vorkommen.

In LISP können im Rumpf von Nichtstandardfunktionen, d.h. LAMBDA– oder LABEL–Funktionen, weitere Nichtstandardfunktionen deklariert werden (siehe Beispiel 3.1). Der RLK_{Fe} schließt die Rümpfe der darin enthaltenen Funktionsdeklarationen mit ein.

In LISP/N kommen Funktionsdeklarationen dagegen nicht im Anweisungsteil einer NSF, sondern im Funktions–Deklarationsteil des Rumpfes der NSF vor, d.h. der relevante lokale Kontext umfaßt die Identifikatoren in den „inneren“ Anweiungsteilen nicht. Dies ist aber auch für unsere Zwecke ausreichend, da für eine mögliche frühzeitige Speicherplatzfreigabe der GDV–Verweis immer im Zusammenspiel mit Verweisen auf statische Vorgänger aus der Linkage und dem Parameterteil eines AR betrachtet wird (siehe Bemerkung 2.3) und von daher keine Informationen verloren gehen können.

Aufgrund der Call By Name–Sematik von LISP/N reicht es nicht mehr aus, den (relevanten) lokalen Kontext nur für Applikationen, d.h. Aufrufe mit aktuellen Parametern, zu betrachten, sondern es wird allgemein der (relevante) lokale Kontext für Vorkommen von Nichtstandardidentifikatoren in Anweisungsteilen bestimmt:

Definition 6.1 Sei Π ein syntaktisches Programm und (I, k) das Vorkommen eines angewandten Nichtstandardidentifikators I in Π. Für den *lokalen Kontext* von (I, k) (kurz: $\mathcal{LK}(I, k)$) wird unterschieden:

1. (I, k) kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor. Dann ist der $\mathcal{LK}(I, k)$ der Anweisungsteil der kleinsten (I, k) echt umfassenden NSF f (wobei f auch das Hauptprogramm sein kann).

2. (I, k) kommt in einem aktuellen Parameter a_i eines NSF–Aufrufs vor, wobei a_i der kleinste (I, k) echt umfassende aktuelle Parameter ist. Dann ist der $\mathcal{LK}(I, k)$ der aktuelle Parameter a_i.

Der $\mathcal{LK}$ bezeichnet somit diejenigen syntaktischen Zeichenfolgen in den Anweisungsteilen von NSF'en in echten LISP/N–Programmen Π, die zur Laufzeit entweder im Anweiungsteil selber ausgewertet oder gemäß der Call By Name–Kopierregelsemantik als aktueller Parameter unausgewertet an eine andere NSF weitergereicht werden können.

In *übersetzten* LISP/N–Programmen Π' (siehe Definition 2.1) ist der $\mathcal{LK}(I, k)$ immer identisch mit demjenigen Anweisungsteil, der das Vorkommen (I, k) umfaßt!

Bemerkung 6.1

1. In Kapitel 4 haben wir einen $\mathcal{LK}$ schon im Satz 4.2 umschrieben. Wir können nun einfacher formulieren: „. . . c kommt innerhalb des $\mathcal{LK}(c, k)$ auf einem linken Ast vor. . .“

2. Die Punkte 1 und 2 der Definition 5.1 des relevanten linken Astes können nun wie folgt zusammengefasst werden: „Im $\mathcal{LK}(\ell, k)$ ist (ℓ, k) bei einer möglichen Schachtelung von linken Ästen der äußerste linke Ast.“

Interessiert man sich dafür, welche syntaktischen Zeichenfolgen *nach* einem NSF–Aufruf innerhalb eines Anweisungsteiles noch erreicht werden können, so ist dies im allgemeinen statisch nicht entscheidbar. Jedoch können bestimmte syntaktische Zeichenfolgen innerhalb eines Anweisungsteils angegeben werden, die für die „lokale Fortsetzung“ sicher nicht mehr in Frage kommen. Dadurch reduziert sich der zu untersuchende syntaktische Kontext vom lokalen auf den „relevanten lokalen Kontext“:

Definition 6.2 Sei Π ein syntaktisches Programm und (I, k) das Vorkommen eines angewandten Nichtstandardidentifikators I in Π. Der *relevante lokale Kontext* von (I, k) (kurz: $\mathcal{RLK}(I, k)$) ist der $\mathcal{LK}(I, k)$

1. ohne (I, k) selbst,
2. ohne die syntaktischen Zeichenfolgen links von (I, k),
3. falls (I, k) der Caller einer Applikation ist: ohne die zugehörige aktuelle und ggf. pending Parameterliste(n) und
4. falls (I, k) im then–Teil eines Konditionals vorkommt: ohne den zugehörigen else–Teil dieses Konditionals.

Bemerkung 6.2 Aufgrund der Definition 6.2 ist der $\mathcal{RLK}(I, k)$ immer leer, falls (I, k) *nicht* innerhalb eines linken Astes vorkommt!

Bemerkung 6.3 Im Laufe dieser Arbeit sprechen wir auch öfters kurz vom $\mathcal{LK}(S)$ bzw. $\mathcal{RLK}(S)$, wobei S eine syntaktische Zeichenfolge gemäß Definition 1.4 sei und in der Regel einen aktuellen Parameter oder eine Applikation darstellt. In den Definitionen 6.1 und 6.2 ist dann (I, k) durch das gemeinte Vorkommen (S, k) zu ersetzen.

Bemerkung 6.4 Die Wahl eines anderen Schrifttyps soll den (relevanten) lokalen Kontext nach Felgentreu (RLK_{Fe} bzw. LK_{Fe}) von den in obigen Definitionen neu definierten $\mathcal{RLK}$ bzw. $\mathcal{LK}$ zusätzlich unterscheiden.

Wir verdeutlichen den $\mathcal{RLK}$ entsprechend Beispiel 3.3:

Beispiel 6.2 Im Unterschied zu Beispiel 3.3 geben wir hier den $\mathcal{RLK}$ nicht nur für Aufrufe, sondern allgemeiner für angewandte Vorkommen (I, k) von Nichtstandardidentifikatoren I an:

(a) *null*,
(b) x (aktueller Parameter vom SF–Aufruf CAR(x)),
(c) *append*,
(d) x (aktueller Parameter vom NSF–Aufruf null(x)) und
(e) y (im then–Teil).

Das jeweilige Nichtstandardidentifikator–Vorkommen (I, k) ist dabei kursiv und der zugehörige $\mathcal{RLK}(I, k)$ unterstrichen dargestellt:

(a)
```
append = λ x y . { IF null(x) THEN y
                              ELSE CONS(CAR(x),
                                        append(CDR(x),y))
                   FI }
```

(b)
```
append = λ x y . { IF null(x) THEN y
                              ELSE CONS(CAR( x),
                                        append(CDR(x),y))
                   FI }
```

(c)
```
append = λ x y . { IF null(x) THEN y
                              ELSE CONS(CAR(x),
                                        append(CDR(x),y))
                   FI }
```

(d)
```
append = λ x y . { IF null( x) THEN y
                               ELSE CONS(CAR(x),
                                         append(CDR(x),y))
                   FI }
```

(e)
```
append = λ x y . { IF null(x) THEN y
                              ELSE CONS(CAR(x),
                                        append(CDR(x),y))
                   FI }
```

Aufgrund der Call By Name–Semantik finden Aufrufe nicht nur an den Stellen in einem Programm statt, an denen eine Applikation *vorkommt*, sondern auch bei der Auswertung formaler Identifikatoren–Vorkommen. Dazu folgende Sprechweise:

Definition 6.3 Sei (I, k) das angewandte Vorkommen eines Nichtstandardidentifikators I in einem syntaktischen LISP/N–Programm Π. Wir sprechen von dem *Aufruf* $c = f(\ldots)$ *der NSF* f *an der Position* (I, k) wenn

- (I, k) das Vorkommen eines Callers einer Applikation ist, und diese Applikation den gewöhnlichen oder formalen Aufruf c ergibt oder
- (I, k) das Vorkommen eines formalen Identifikators I ist, und die Auswertung von I den Aufruf c ergibt,

wobei c jeweils der *erste* solche Aufruf sei, den eine Auswertung von (I, k) ergibt.

Den schon oben erwähnten Begriff der „lokalen Fortsetzung“ wollen wir an dieser Stelle einführen:

Definition 6.4 Sei Π ein syntaktisches Programm und (I, k) das angewandte Vorkommen eines Nichtstandardidentifikators I in dem Anweisungsteil r einer NSF g (wobei g auch das Hauptprogramm sein kann). Sei $c = f(\ldots)$ der Aufruf einer NSF f an der Position (I, k). Dann sprechen wir von der *lokalen Fortsetzung von* c, wenn wir die weitere Abarbeitung von r *nach* der Ausführung von c meinen.

Bemerkung 6.5 Zur lokalen Fortsetzung eines NSF–Aufrufs c auf einem linken Ast gehört somit immer der zugehörige rechte Ast.

Im Zusammenhang mit der GDV–Situation können wir knapp formulieren: Der GDV–Verweis für einen NSF–Aufruf auf einem linken Ast sichert das (gesamte) Enviroment für die korrekte lokale Fortsetzung von $c \ldots$

Wie wir in Beispiel 6.1 schon gesehen haben, ist für die korrekte lokale Fortsetzung eines Aufrufs c auf einem linken Ast durch einen GDV–Verweis nicht notwendigerweise das gesamte Enviroment, d.h. *alle* AR's im Keller zu halten, sondern wir werden sehen, daß es genügt, gewisse zu Nichtstandardidentifikatoren „zugehörige AR's" im Keller zu sichern:

Definition 6.5 Sei (I, k) das Vorkommen eines angewandten Nichtstandardidentifikators I in einem echten LISP/N–Programm Π. Das zur Laufzeit zu (I, k) *zugehörige AR* ist das AR an der Anfangsadresse DN(SV(I, k) (siehe Definitionen 1.23 und 1.12).

Es folgt der entscheidende Satz, welcher die Relevanz des $\mathcal{RLK}$ für die GDV–Situation verdeutlicht:

Satz 6.1 Sei Π ein syntaktisches Programm und (I, k) ein angewandtes Vorkommen eines Nichtstandardidentifikators I auf einem linken Ast in Π. Sei $c = f(\ldots)$ der Aufruf einer NSF f an der Position (I, k), dann enthält der $\mathcal{RLK}(I, k)$ *alle* Nichtstandardidentifikator–Vorkommen, die bei der lokalen Fortsetzung von c erreicht werden könnten und somit deren zugehörige AR's durch einen GDV–Verweis vor der frühzeitigen Speicherplatzfreigabe durch c geschützt werden müssen.

Beweis (Satz 6.1) Sei r der zur lokalen Fortsetzung von c gehörige Anweisungsteil, und sei (id, i) ein beliebiges Vorkommen eines Nichtstandardidentifikators id in Π. Dann ist zu zeigen, daß falls $(id, i) \notin \mathcal{RLK}(I, k)$ gilt, (id, i) bei der lokalen Fortsetzung von c in r nicht erreicht werden kann. Insbesondere ist dann (id, i) ein für einen GDV–Verweis durch c irrelevantes Vorkommen:

Dazu betrachten wir alle Möglichkeiten, bei denen (id, i) *nicht* im $\mathcal{RLK}(I, k)$ vorkommt:

1. $(id, i) \notin \mathcal{LK}(I, k)$:

   (a) (I, k) kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vor $\Rightarrow$ (id, i) kommt nicht in r vor und ist somit bei der lokalen Fortsetzung von c in r nicht erreichbar.

   (b) (I, k) kommt innerhalb eines aktuellen Parameters a_i eines NSF–Aufrufs $c' = e(\ldots)$ vor und (id, i) kommt nicht in a_i vor $\Rightarrow$ gemäß der Call By Name–Kopierregelsemantik von LISP/N wird c frühstens im Anweisungsteil der gerufenen NSF e bei der Auswertung eines formalen Identifikator–Vorkommens (I', k') ausgeführt. Dann aber ist der Satz für die Position (I', k') anstatt (I, k) anzuwenden.

2. $(id, i) \in \mathcal{LK}(I, k) \wedge (id, i) \notin \mathcal{RLK}(I, k)$:

   (a) $(id, i) = (I, k)$, d.h. $id \equiv I$ und $i = k \Rightarrow (id, i)$ ergibt den Aufruf c selber und kann daher *nach* Ausführung von c in r nicht mehr erreicht werden.

   (b) (id, i) kommt *links* von (I, k) vor $\Rightarrow$ (id, i) kann *nach* Abarbeitung von c in r nicht erreicht werden, da Anweisungsteile von links nach rechts bearbeitet werden.

   (c) (I, k) ist das Vorkommen des Callers I einer Applikation c', und (id, i) kommt in einem aktuellen Parameter von c' vor, d.h. $c' = I(\ldots id \ldots) \Rightarrow (id, i)$ gehört zum Aufruf c selber und kann daher *nach* Ausführung von c in r nicht mehr erreicht werden.

   (d) (I, k) kommt im then–Teil eines Konditionals ξ vor, und (id, i) kommt innerhalb des else–Teils von ξ vor $\Rightarrow$ wird c ausgeführt, kann (id, i) *nach* c innerhalb von r nicht mehr erreicht werden.

$\stackrel{1,2}{\Longrightarrow}$ Kommt (id, i) nicht im $\mathcal{RLK}(I, k)$ vor, so kann (id, i) *nicht* bei der lokalen Fortsetzung von c erreicht werden. Da c innerhalb eines linken Astes ausgeführt wurde, kann (id, i) somit insbesondere nicht im zugehörigen rechten Ast vorkommen und ein GDV–Verweis zum Erhalten des zu (id, i) zugehörigen AR's ist unnötig. □

Wir nutzen im folgenden die statisch entscheidbare maximale Menge der erreichbaren Nichtstandardidentifikatoren in der lokalen Fortsetzung nach einem linker Ast–Aufruf aus, um zu einem häufig kostengünstigereren GDV–Verweis zu gelangen:

6.1.4 Der neue GDV–Verweis

Ähnlich wie Honschopp *dynamisch* für die frühzeitige Speicherplatzfreigabe in der Situation NSF–Aufruf den maximalen (also BFS–nächsten) Verweis aus einem zunächst provisorisch angelegten AR bestimmt, sammeln wir nun

statisch alle Nichtstandardidentifikatoren im $\mathcal{RLK}(I,k)$ eines angewandten Nichtstandardidentifikator–Vorkommens (I,k) und bestimmen daraus das maximale statische Niveau:

Definition 6.6 Sei (I,k) das Vorkommen eines angewandten Nichtstandardidentifikators I in einem syntaktischem LISP/N–Programm Π. Sei

$$\Omega := \{\ (I',k') \mid I' \in \text{NSIDF} \wedge (I',k') \in \mathcal{RLK}(I,k)\ \}$$

Dann bestimmen wir aus Ω das maximale statische Niveau der im $\mathcal{RLK}(I,k)$ enthaltenen Nichtstandardidentifikatoren wie folgt:

$$\text{GMNIV}(I,k) := \begin{cases} 0 \text{ , falls } \Omega = \emptyset \\ \max\{\ \text{SN}(I',k') \mid (I',k') \in \Omega\ \} \text{ , sonst.} \end{cases}$$

Bemerkung 6.6 Entsprechend Bemerkung 6.3 werden wir auch öfters kurz GMNIV(S) schreiben.

Mit der GMNIV–Information und der Aussage aus Satz 6.1 können wir nun *statisch* darüber entscheiden, ob ein (neuer) GDV–Verweis für einen Aufruf auf einem linken Ast überhaupt notwendig ist:

Satz 6.2 Sei (I,k) das Vorkommen eines angewandten Nichtstandardidentifikators I innerhalb eines linken Astes in einem syntaktischen LISP/N–Programm Π, und sei $c = f(\ldots)$ der Aufruf einer NSF f an der Position (I,k). Falls gilt: GMNIV$(I,k) = 0$, so ist für c *kein* GDV–Verweis notwendig und der Aufruf kann so behandelt werden, als käme er nicht auf einem linken Ast vor (der zuvor gültige GDV–Verweis kann dann für c unverändert übernommen werden).

Beweis (Satz 6.2): Aufgrund Satz 6.1 enthält der $\mathcal{RLK}(I,k)$ *alle* Vorkommen von Nichtstandardidentifikatoren, deren zugehörige AR's für die lokale Fortsetzung von c noch benötigt werden könnten und somit durch einen GDV–Verweis bei c vor der frühzeitgen Speicherplatzfreigabe geschützt werden müssen. Gilt nun GMNIV$(I,k) = 0$, so ist entweder

1. der $\mathcal{RLK}(I,k)$ leer oder
2. der $\mathcal{RLK}(I,k)$ enthält nur Nichtstandardidentifikatoren mit statischem Niveau 0.

Zu 1.: In der lokalen Fortsetzung von c wird kein Nichtstandardidentifikator erreicht und somit auch auf kein AR zugegriffen. Ein neuer GDV–Verweis für c ist nicht notwendig.

Zu 2.: Die im $\mathcal{RLK}(I,k)$ enthaltenen Nichtstandardidentifikatoren können nur die angewandten Vorkommen von Funktionsidentifikatoren von im Funktions–Deklarationsteil des Hauptprogramms vorkommenden NSF's sein. Der statische Vorgänger dieser Funktionen ist das Hauptprogramm und da das AR vom Hauptprogramm sowieso nie gelöscht wird, ist ein neuer GDV–Verweis für c nicht notwendig. □

Bemerkung 6.7 In dem einführenden Beispiel 6.1 gilt GMNIV($g(y)$) = 0, da im $\mathcal{RLK}(g(y))$ nur das Atom B als S–Ausdruck und somit insbesondere kein Nichtstandardidentifikator mit statischem Niveau größer 0 vorkommt — für den NSF–Aufruf g(A) ist kein GDV–Verweis mehr notwendig!

Der häufigere Fall dürfte aber sein, daß gemäß der Situation in Satz 6.2 GMNIV$(I,k) > 0$ gilt, d.h. im $\mathcal{RLK}(I,k)$ Nichtstandardidentifikatoren mit maximalem statischem Niveau echt größer als 0 vorkommen, und die zugehörigen AR's somit durch einen GDV–Verweis im Keller gehalten werden müssen. Für den Aufruf $c = f(\ldots)$ an der Position (I,k) kann dann zur Laufzeit mit Hilfe der Indexregister (IR) über den Zugriff

$$\text{IR [GMNIV}(I,k)-1 \text{]}$$

der für c *notwendige* GDV–Verweis bestimmt werden. Diese Information wird nun zusammen mit der Rücksprungadresse (RA) von c in die GDV–Linkagezelle geschrieben (vgl. Kapitel 5).

Da der neue GDV–Verweis jetzt nicht mehr notwendig auf ein dynamisches Niveau des dynamischen Vorgängers von c verweist, muß noch dafür Sorge getragen werden, daß zuvor gültige GDV–Verweise nicht übersprungen werden: Dafür ist vor einer möglichen Speicherplatzbereinigung das Maximum aller GDV–Verweise im AR–Keller zu ermitteln. Um diesen Aufwand zu reduzieren und ständig Zugriff auf den aktuellen maximalen GDV–Verweis im Keller zu haben, fügen wir jeder GDV–Linkagezelle noch eine dritte Information hinzu, und erhalten dann:

$$\text{GDV–Linkagezelle := RA | MaxGDV | IR[GMNIV}(I,k)-1]$$

Immer dann, wenn ein neuer GDV–Verweis anfällt, wird der neue Wert von MaxGDV wie folgt ermittelt:

$$\text{MaxGDV := max (MaxGDV vom dyn. Vorgänger , IR[GMNIV}(I,k)-1] \text{)}$$

und ebenfalls in die GDV–Linkagezelle eingetragen. In der GDV–Linkagezelle zum Hauptprogramm, d.h. dem ersten AR im AR–Keller, wird MaxGDV mit 1 initialisiert.

Insgesamt erhalten wir somit: Ist (I,k) das Vorkommen eines angewandten Nichtstandardidentifikators I auf einem linken Ast in einem syntaktischen LISP/N–Programm und muß für den Aufruf $c = f(\ldots)$ einer NSF f an der Position (I,k) ein neuer GDV bestimmt werden, d.h. gilt gemäß Abschnitt 5.1 LACALL=false, so wird die GDV–Linkagezelle für das AR_c zum Aufruf c wie folgt besetzt:

- Falls gilt: GMNIV(I,k)=0, dann wird gemäß Satz 6.2 kein neuer GDV gesetzt, sondern der alte übernommen:
  GDV–Linkagezelle := GDV–Linkagezelle vom dynamischen Vorgänger.

- Falls gilt: GMNIV(I,k)>0, dann bestimme
  MaxGDV := max (MaxGDV vom DN(DV(c)) , IR[GMNIV$(I,k)-1$])
  und setze dann
  GDV–Linkagezelle := RA$_c$ | MaxGDV | IR[GMNIV(I,k)-1].

Nach der zunächst provisorischen Anlage des AR$_c$ wird dann wie gehabt der maximale Verweis in den AR–Keller ermittelt und ggf. eine HOpt durchgeführt.

Damit das Laufzeitsystem zur Laufzeit über die GMNIV–Informationen verfügt, und somit der GDV–Verweis entweder gar nicht oder wenn dann gezielt gesetzt werden kann, ist der Markierungs–Algorithmus GMARK entwickelt worden. Er ist in den Compiler integriert worden und ermittelt für jedes angewandte Vorkommen (I,k) eines Nichtstandardidentifikators I innerhalb eines linken Astes den $\mathcal{RLK}(I,k)$. Die daraus resultierende GMNIV(I,k)–Information wird als Kommentar in Form von

(* GMNIV(I,k) *)

direkt *nach* dem Vorkommen (I,k) in den Programmtext eingfügt.

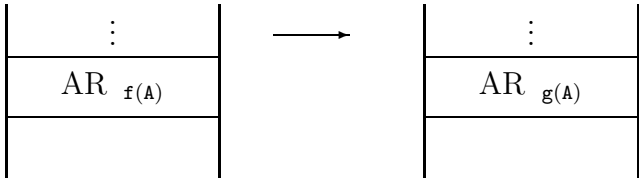
Bemerkung 6.8 Da im Anweisungsteil r des Hauptprogamms nur Nichtstandardidentifikatoren mit statischem Niveau 0 vorkommen können (d.h. die angewandten Vorkommen von Funktionsidentifikatoren), gilt für *alle* angewandten Vorkommen (I,k) von Nichtstandardidentifikatoren I in r: GMNIV$(I,k) = 0$. Aufgrund Satz 6.2 ist daher für NSF–Aufrufe innerhalb von r kein GDV–Verweis notwendig. Aus diesem Grunde wird auf die Markierung von r verzichtet, und linke Äste in r werden vom Compiler einfach wie rechte Äste übersetzt.

Das Beispiel 6.1 würde somit von GMARK wie folgt markiert:

Beispiel 6.1 (Fortsetzung)

```
g = λ x . { x }
f = λ y . { CONS ( g(*0*)(y(*0*)) , B )}
f(A)
```

Entsprechend Bemerkung 6.7 sind alle Marken „0“ und für die Ausführung des Programms braucht überhaupt kein neuer GDV–Verweis gesetzt zu werden. Der Ablauf im AR–Keller sieht damit wie folgt aus:



Das $AR_{f(A)}$ wird durch den Aufruf g(A) frühzeitig freigegeben und der Speicherplatzgewinn ist offensichtlich.

Bevor wir zu der Relevanz dieser Maßnahmen kommen und anschließend den GMARK–Markierungsalgorithmus vorstellen, werden in den nächsten beiden Abschnitten notwendige Maßnahmen für eine korrekte Implementierung der soeben vorgestellten GDV–Optimierung aufgezeigt, die aber zugleich auch eigenständige Optimierungen des Honschopp–Laufzeitsystems darstellen:

6.2 Dicke Parameter–Optimierung durch statische Programmanalyse

Wie wir in Abschnitt 2.2 gesehen haben, wird aus implementations–technischen Gründen ein dicker Parameter δ, d.h. eine Applikation oder ein Konditional als aktueller Parameter eines NSF–Aufrufs c, also $c = f(\ldots, \delta, \ldots)$, übersetzt wie ein Aufruf $c' = f(\ldots, h(), \ldots)$. Dabei ist $h()$ der Aufruf einer zusätzlichen, vom Compiler eingeführten parameterlosen Nichtstandard–Hilfsfunktion h.

Damit bei einer späteren Ausführung von h (wegen der Call By Name Strategie werden beim Aufruf c' die Argumente nicht sofort ausgewertet) noch alle notwendigen Informationen im Laufzeitkeller zugänglich sind, wird die NSF h vom Compiler so übersetzt, als komme sie im Funktions–Deklarationsteil der kleinsten den Aufruf c umfassenden NSF g vor. Dies wiederum bedingt, daß die NSF g zum statischen Vorgänger SV(h) von h wird und erst dann freigegeben werden kann, wenn h nicht ausgeführt oder nicht mehr als Parameter weitergereicht wird.

Um diese oft unökonomische Handhabung zu demonstrieren, folgendes (durch GMARK bereits markiertes) Programm:

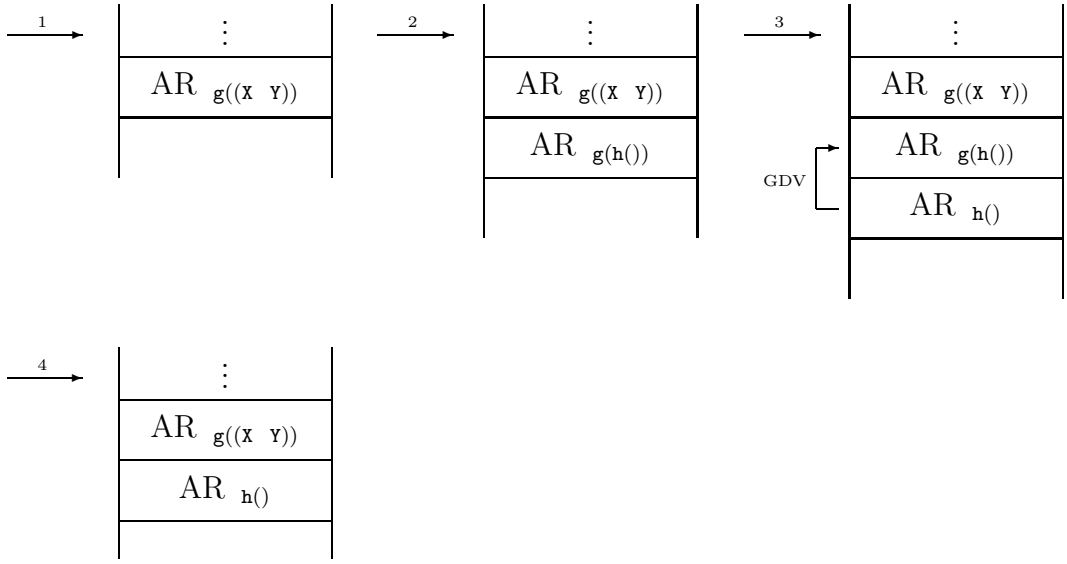
Beispiel 6.3 (vgl. Beispiel „dckop.lsp“ im Anhang A.1)

```
g = λ x . { IF ATOM(x(*1*)) THEN x
                            ELSE g(CAR((A B)))
                                   dicker Parameter
            FI }
g((X Y))
```

Das Programm wird gemäß der Handhabung dicker Parameter (siehe Abschnitt 2.2) vom Compiler übersetzt wie

```
g = λ x . { h = λ . { CAR((A B)) }
            IF ATOM(x(*1*)) THEN x
                            ELSE g(h())
            FI }
g((X Y))
```

Für den Aufruf g((X Y)) ergibt sich dann folgender AR–Kellerablauf:



Durch den Aufruf g(CAR((A B)) im else–Teil der NSF g wird das $\mathrm{AR}_{g((X\ Y))}$ nicht frühzeitig freigegeben, weil g der statische Vorgänger der Hilfsfunktion h zum dicken Parameter CAR((A B)) ist. Es ist jedoch offensichtlich, daß für die spätere Ausführung von CAR((A B)) das $\mathrm{AR}_{g((X\ Y))}$ nicht benötigt wird.

Im allgemeinen sollte das statische Niveau einer Hilfsfunktion h zu einem dicken Parameter so gewählt werden, daß zur Laufzeit durch den resultierenden Verweis auf den statischen Vorgänger der Hilfsfunktion $\mathrm{SV}(h)$ nur solche AR's nicht freigegeben werden können, die für die Auswertung des dicken Parameters nötige Informationen enthalten. Nicht die kleinste den dicken Parameter umfassende NSF, sondern ggf. eine äußere umfassende NSF kann als statischer Vorgänger für h genügen.

Ähnlich wie für die GDV–Optimierung können wir *statisch* darüber entscheiden, welches statische Niveau die Hilfsfunktion h bei der Übersetzung vom Compiler zugewiesen bekommen soll:

Definition 6.7 Sei δ ein dicker Parameter als aktueller Parameter eines Aufrufs c der NSF f, also $c = f(\ldots, \delta, \ldots)$, in einem syntaktischen LISP/N–Programm Π. Sei

$$\Theta := \{\ (I', k') \mid I' \in \mathrm{NSIDF} \wedge (I', k') \in \delta\ \}.$$

Dann bestimmen wir aus Θ das maximale statische Niveau der in δ enthaltenen Nichtstandardidentifikatoren wie folgt:

$$\mathrm{DPNIV}(\delta) := \begin{cases} 0\ , \text{ falls } \Theta = \emptyset \\ \max\ \{\ \mathrm{SN}(I', k') \mid (I', k') \in \Theta\ \}\ , \text{ sonst.} \end{cases}$$

Falls in δ keine Nichtstandardidentifikatoren vorkommen, so ergibt demnach $\mathrm{DPNIV}(\delta)$ eine 0.

Erreicht der Compiler bei der Übersetzung einen dicken Parameter δ, dann wird für die zu generierende Hilfsfunktion h das statische Niveau SN(h) nicht mehr mit der gerade aktuellen Schachtelungstiefe der Funktions–Rümpfe gleichgesetzt, sondern durch

$$\mathrm{SN}(h) \texttt{ := } \mathrm{DPNIV}(\delta)$$

bestimmt. Zur Laufzeit kann dann das Laufzeitsystem (wie üblich!) über den Zugriff

$$\mathrm{IR}\ [\ \mathrm{SN}(h)\ -1\ ]$$

einen Verweis auf das AR vom *notwendigen* statischen Vorgänger SV(h) von δ und somit im günstigsten Fall auf das AR vom Hauptprogramm ermitteln. Für die in δ enthaltenen Nichtstandardidentifikatoren werden die zugehörigen AR's durch den so ermittelten Verweis im Keller gehalten, und es gehen keine noch evtl. benötigten Informationen verloren.

Für die Berechnungen der DPNIV's und für die Markierung des Programmtextes konnte der GMARK–Algorithmus entsprechend erweitert werden. Dabei wird unmittelbar *vor* jedem dicken Parameter δ die DPNIV(δ)–Information ebenfalls in Form eines Kommentars

$$\texttt{(*}\ \mathrm{DPNIV}(\delta)\texttt{+100 *)}$$

in den Programmtext eingefügt.

Bemerkung 6.9 Zu der DPNIV–Information wird 100 addiert, um die Marke eindeutig von einer GMNIV–Information zu unterscheiden. Dadurch kann die Einbettung der Marken in Kommentarklammern einheitlich beibehalten werden.

Unser Beispiel würde, zusammen mit den Marken für die GDV–Optimierung, wie folgt aussehen:

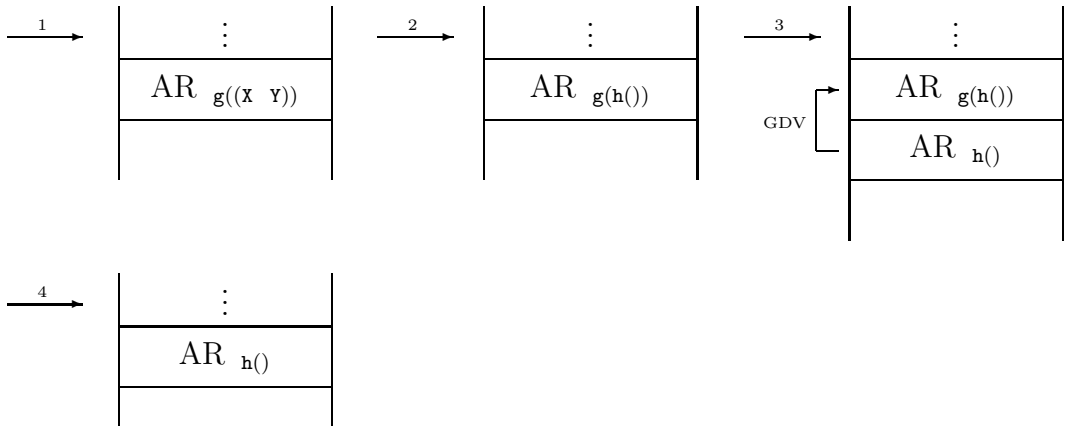
Beispiel 6.3 (Fortsetzung):

```
g = λ x . { IF ATOM(x(*1*)) THEN x
                            ELSE g((*100*)CAR((A B)))
          FI }
g((X Y))
```

und damit vom Compiler übersetzt werden wie

```
h = λ . { CAR((A B)) }
g = λ x . { IF ATOM(x(*1*)) THEN x
                            ELSE g(h())
          FI }
g((X Y))
```

Der dicke Parameter CAR((A B)) bekommt vom Compiler somit das statische Nivau 0 und hat als statischen Vorgänger das Hauptprogramm. Der Ablauf im AR–Keller für den Aufruf g((X Y)) sieht damit wie folgt aus:



Auf das $AR_{g((X\ Y))}$ wird nicht mehr verwiesen, und es kann frühzeitig freigegeben werden. Der Speicherplatzgewinn ist offensichtlich.

Wie durch Beispiel 6.3 verdeutlicht, stellt die in diesem Abschnitt vorgestellte dicke Parameter–Optimierung für sich alleine schon eine eigenständige Optimierung des Honschopp–Systems dar. Im Zusammenhang mit der im vorherigen Abschnitt vorgestellten Optimierung von GDV–Verweisen, ist die dicke Parameter–Optimierung jedoch eine *notwendige* Maßnahme für eine korrekte Implementierung:

Tritt ein dicker Parameter δ im $\mathcal{RLK}$ eines linker Ast–NSF–Aufrufs auf, so kann es sein, daß *ohne* die dicke Parameter–Optimierung durch einen optimierten GDV–Verweis der statische Vorgänger SV(h) der Hilfsfunktion h zum dicken Parameter δ gelöscht wird und eine fehlerhafte Konstellation im Laufzeitkeller, insbesondere in der statischen Verweiskette, entsteht:

Beispiel 6.4 Gegeben sei folgendes Programm (vgl. Beispiel „dckop1“ im Anhang A.1):

```
f = λ x.
      { g = λ y . { IF ATOM(x) THEN x
                               ELSE CONS(f(y),f(CDR(x)))
                  FI }
        g(A) }
f((A B))
```

wobei der Aufruf CDR(x) ein dicker Parameter ist und deshalb dieses Beispiel vom Compiler *ohne* dicke Parameter–Optimierung wie folgt übersetzt wird:

```
f = λ x.
       { g = λ y . { h = λ . { CDR(x) }
                     IF ATOM(x) THEN x
                                ELSE CONS(f(y),f(h()))
                     FI }
         g(A) }
f((A B))
```

Ohne die dicke Parameter–Optimierung würde durch den Aufruf f(A) im linken Ast von CONS das $\mathrm{AR}_{g(A)}$ durch den optimierten GDV gelöscht werden. Die NSF g ist aber der statische Vorgänger der Hilfsfunktion h zum dicken Parameter δ =CDR(x), und das $\mathrm{AR}_{g(A)}$ darf deshalb nicht gelöscht werden.

Durch die dicke Parameter–Optimierung wird jedoch vor δ die Marke `(*101*)` eingefügt und h somit mit $\mathrm{SN}(h) = 0$ übersetzt. Dadurch ist nicht mehr g, sondern f der statische Vorgänger von h, und das Beispiel wird korrekt abgearbeitet.

Bemerkung 6.10 Im folgenden setzen wir die dicke Parameter–Optimierung voraus, wenn wir die GDV–Optimierung nutzen.

Im nächsten Abschnitt werden wir sehen, daß die Ausnutzung der GMARK–Markierungen zur GDV–Optimierung eine wichtige Änderung des Laufzeitsystems nötig macht:

6.3 Die Einführung des GDV–Kellers

Das Laufzeitsystem von Honschopp basiert, wie beschrieben, auf dem Prinzip, daß die Speicherplatzbereinigung in der Situation „Funktionsaufruf“ stattfindet und nicht erst, wie sonst üblich, in der Situation „Funktionsende“. Daher wurde durch die Einführung zweier neuer Keller sichergestellt, daß die Informationen, die für die Abarbeitung der Funktionen *nach* der Speicherplatzfreigabe noch benötigt werden, auf keinen Fall verloren gehen: Der RA–Keller für die Rücksprungadressen und der HV–Keller für die Hilfsvariablen, d.h. für die Ergebnisse der Auswertung von linken Ästen von mehrstelligen Standardfunktionen (in LISP/N: CONS und EQ).

Die ursprüngliche (definitionsgemäße) Handhabung der GDV–Situation stellte immer sicher, daß ein duch einen GDV–Linkage–Verweis einmal geschütztes AR nicht eher wieder freigegeben werden konnte, als bis der Aufruf beendet wurde, der zum Setzen des GDV–Verweises führte. Durch das rigorose Setzen des GDV auf den jeweiligen dynamischen Vorgänger konnten alle zuvor gültigen GDV–Verweise nicht verlorengehen.

Die Effizienz des neuen GDV ist darin begründet, daß ein GDV (falls er überhaupt gesetzt werden braucht!) nun nicht mehr den gültigen Kellerzustand durch einen Verweis auf den aktuellen dynamischen Vorgänger „einfriert“,

sondern ein Verweis auch auf ein AR tief im Keller möglich ist. Dadurch kann es aber passieren, daß zuvor gültige GDV–Linkage–Verweise in denjenigen AR's verloren gehen, die durch die nun mögliche Speicherplatzfreigabe bei einem linker Ast–Aufruf gelöscht wurden. Soll ein GDV–Verweis auf den zuvor gültigen Wert zurückgesetzt werden, so ist dies dann nicht mehr immer möglich:

Betrachten wir dazu folgendes

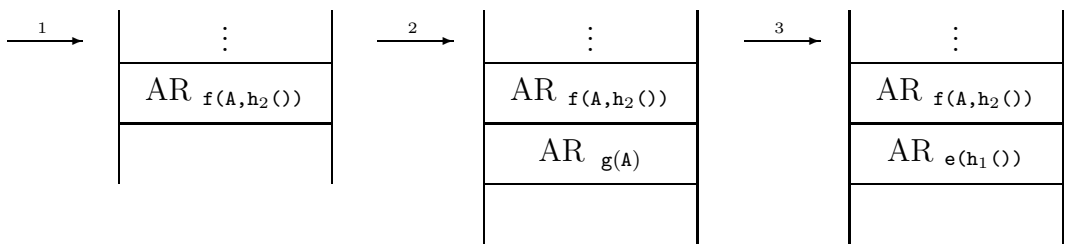
Beispiel 6.5 (vgl. Beispiel „gdvk.lsp“ im Anghang A.1)

```
f = λ x y .
       { e = λ v . { CONS(g(v),y) }
         g = λ w . { IF ATOM(w) THEN e(CONS(D,E))
                                ELSE CONS(x,y)
                     FI }
         g(x) }
f(A,f((B),C))
```

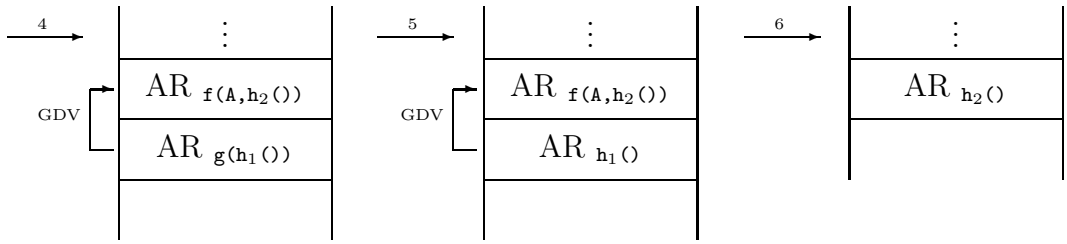
Die einzigen beiden dicken Parameter sind die beiden aktuellen Parameter CONS(D,E) und f((B),C). Beide haben gemäß Definition 6.7 als DPNIV–Information den Wert 0 und die zugehörigen Hilfsfunktionen h_1 und h_2 können vom Compiler so übersetzt werden, als sei das Hauptprogramm der jeweilige statische Vorgänger. Wir erhalten dann als übersetztes Programm:

```
h1 = λ . { CONS(D,E) }
h2 = λ . { f((B),C) }
f = λ x y .
       { e = λ v . { CONS(g(v),y) }
         g = λ w . { IF ATOM(w) THEN e(h1())
                                ELSE CONS(x,y)
                     FI }
         g(x) }
f(A,h2())
```

Dieses Beispiel ist relativ umfangreich, jedoch wird dadurch die Wirkung der obigen GDV–Optimierung noch einmal verdeutlicht. Die Abfolge im AR–Laufzeitkeller sieht wie folgt aus:



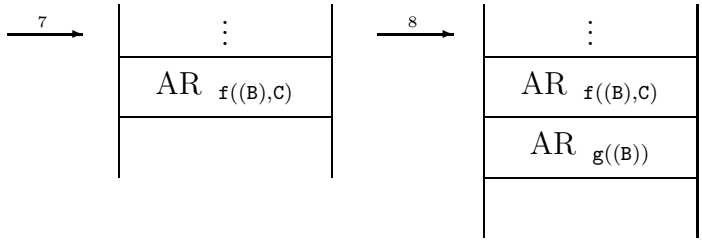
zu 3. Da die Auswertung von ATOM(A) den Wert true liefert, findet der Aufruf e(h_1()) statt. Da DPNIV(CONS(D,E))=0 ist, wurde die zugehörige Hilfsfunktion h_1 mit SN(h_1)=0 übersetzt und hat lediglich das Hauptprogramm als statischen Vorgänger. Daher wird optimiert, und das $AR_{g(A)}$ kann frühzeitig freigegeben werden.



zu 4. In e erfolgt der Aufruf g(h_1()) auf einem linken Ast der SF CONS. Im $\mathcal{RLK}$(g(v)) steht lediglich der Identifikator y, d.h. das maximale statische Niveau ist 1 und gemäß dem Inhalt von IR[1] ist der optimierte GDV das $AR_{f(A,h_2())}$. Das $AR_{e(h_1())}$ kann somit freigegeben werden.

zu 5. Nun erfolgt der entscheidende Aufruf, welcher dann im weiteren Verlauf der Abarbeitung zum falschen Ergebnis führt: Die Auswertung von ATOM(w) führt zum Aufruf der Hilfsfunktion h_1, und weil aus diesem AR kein Verweis auf das $AR_{g(h_1())}$ zeigt, wird das $AR_{g(h_1())}$ gelöscht. Wir sehen, daß dadurch auch der GDV–Verweis auf das $AR_{f(A,h_2())}$ verloren gegangen ist, obwohl der Aufruf g(h_1()), welcher diesen GDV gesetzt hatte, noch nicht beendet ist!

zu 6. Die Auswertung von ATOM liefert den Wert false zurück. Der Aufruf des dicken Parameters aus Schritt 5 ist beendet, und der GDV soll gemäß Definition 2.2 auf den alten Wert zurückgesetzt werden. Da der zuvor gültige GDV–Verweis aus Schritt 4 verloren ist, ist der neue GDV somit wieder ein Verweis auf das AR vom Hauptprogramm. Es wird nun CONS(A,h_2()) im else–Teil ausgeführt, und im rechten Ast von CONS wird die Hilfsfunktion h_2 zum dicken Parameter f((B),C) aufgerufen.



zu 8. Das Ergebnis vom Aufruf aus Schritt 4 liegt vor und somit ist der linke Ast von CONS(g(h_1()),h_2()) ausgewertet. Der rechte Ast war über den

Identifikator y in der zweiten Parameterzelle vom $AR_{f(A,h_2())}$ gebunden und ist fälschlicherweise gelöscht worden. Trotzdem findet (aufgrund des erzeugten Codes) ein Zugriff auf den zweiten Parameter desjenigen AR's statt, auf das unter IR[1] verwiesen wird. Aus Schritt 7 steht hier noch das $AR_{f((B),C)}$, und somit wird das Atom C als aktueller rechter Ast ermittelt. Der Aufruf f((B),C) findet somit nicht statt, und nach der Ausführung von CONS fällt das vermeintliche Endergebnis ((A (B).C).C) an.

Dieses Beispiel verdeutlicht, daß verhindert werden muß, daß GDV–Informationen mit der gewünschten frühzeitigen Freigabe von AR's verloren gehen können. Dazu folgende

Bemerkung 6.11 GDV–Verweise müssen gemäß Definition 2.2 nach dem Keller–Prinzip („FIFO“) verwaltet werden: Ein GDV–Verweis hat solange Gültigkeit bis derjenige Aufruf beendet ist, der den betreffenden GDV gesetzt hat. Danach ist der zuvor gültige GDV–Verweis wieder gültig, usw.

Wie wir im Beispiel 6.5 gesehen haben, kann dieses Keller–Prinzip durch den neuen GDV–Verweis verletzt werden!

Eine Lösungsmöglichkeit dieses Problems wäre z.B., daß ein neuer GDV–Verweis zwar viele AR's freigeben darf, aber mindestens auf dasjenige AR verweist, durch welches der unmittelbar zuvor gültige GDV gesetzt wurde. Es dürften somit maximal diejenigen AR's freigegeben werden, bei deren Anlage der GDV vom dynamischen Vorgänger übernommen wurde.

Diese Vorgehensweise impliziert jedoch, daß AR's evtl. nur wegen der GDV–Linkagezelle im AR–Keller gehalten werden müssen und schließt das Problem mit ein, dasjenige AR zu bestimmen, welches für den aktuell gültigen GDV verantwortlich ist.

Um die Speicherplatzfreigabe in der Situation „Funktionsaufruf“ durchführen zu können hat Honschopp die Rücksprungadressen und Hilfsvariablen in eigene Keller ausgelagert — sonst müßten die betreffenden AR's nur wegen dieser Informationen (oft nur die RA!) komplett im AR–Keller gehalten werden.

Genauso werden wir nun einen eigenen *GDV–Keller* für den GDV–Verweis einführen, in dem ebenfalls nur Push– und Pop–Operationen zulässig sind. Dadurch wird die kellerartige Verwaltung der GDV–Verweise garantiert, und die GDV–Optimierung gemäß Abschnitt 6.1.4 kann in vollem Umfang beibehalten werden. Eine HOpt wird durch eine um den GDV–Eintrag verkleinerte Linkage günstiger. Der Nachteil natürlich, daß ein weiterer Laufzeitkeller eingerichtet werden muß, muß durch eine effektive Implementation möglichst gering gehalten werden (siehe Abschnitt 6.4).

Der neue GDV–Keller wird nun wie folgt gehandhabt:

- Der GDV–Keller hat jeweils die gleichen Einträge wie zuvor die 1. Linkagezelle eines AR, d.h. gemäß Abschnitt 6.1.4:

  RA | MaxGDV | GDV–Verweis .

- Der GDV–Keller wird zum Programmstart mit MaxGDV:=1 und GDV–Verweis:=1 initialisiert. Dadurch wird gemäß Definition 2.2 auf das AR zum Hauptprogramm verwiesen, wenn kein weiterer GDV–Verweis vorliegt.

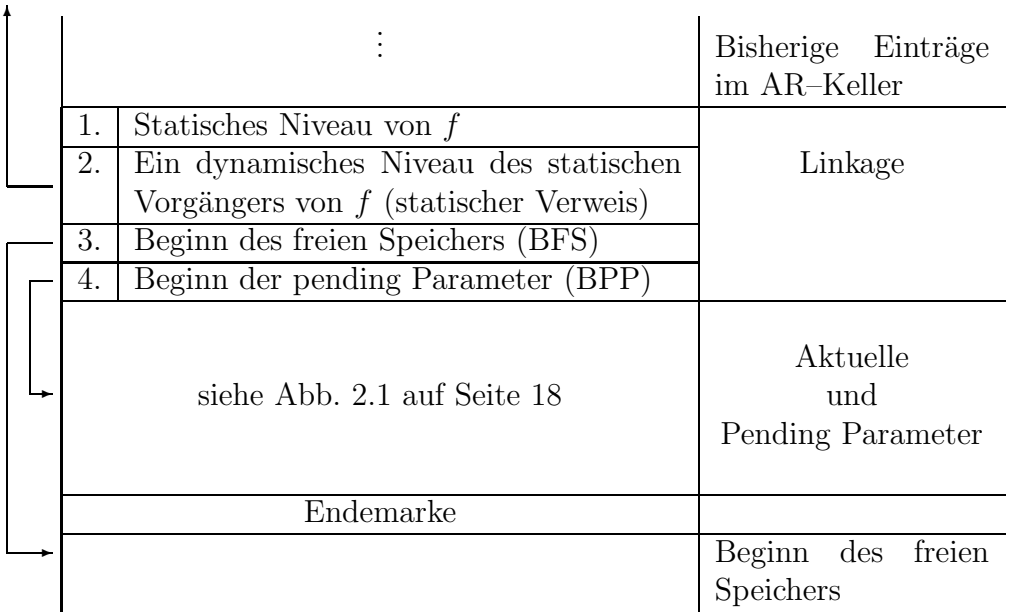
- Fällt ein neuer GDV–Verweis an, so wird der entsprechende Eintrag (s.o.) durch eine Push–Operation in dem Keller abgelegt.

- Der gültige GDV kann immer durch Abfragen des obersten GDV–Keller–Eintrags ermittelt werden.

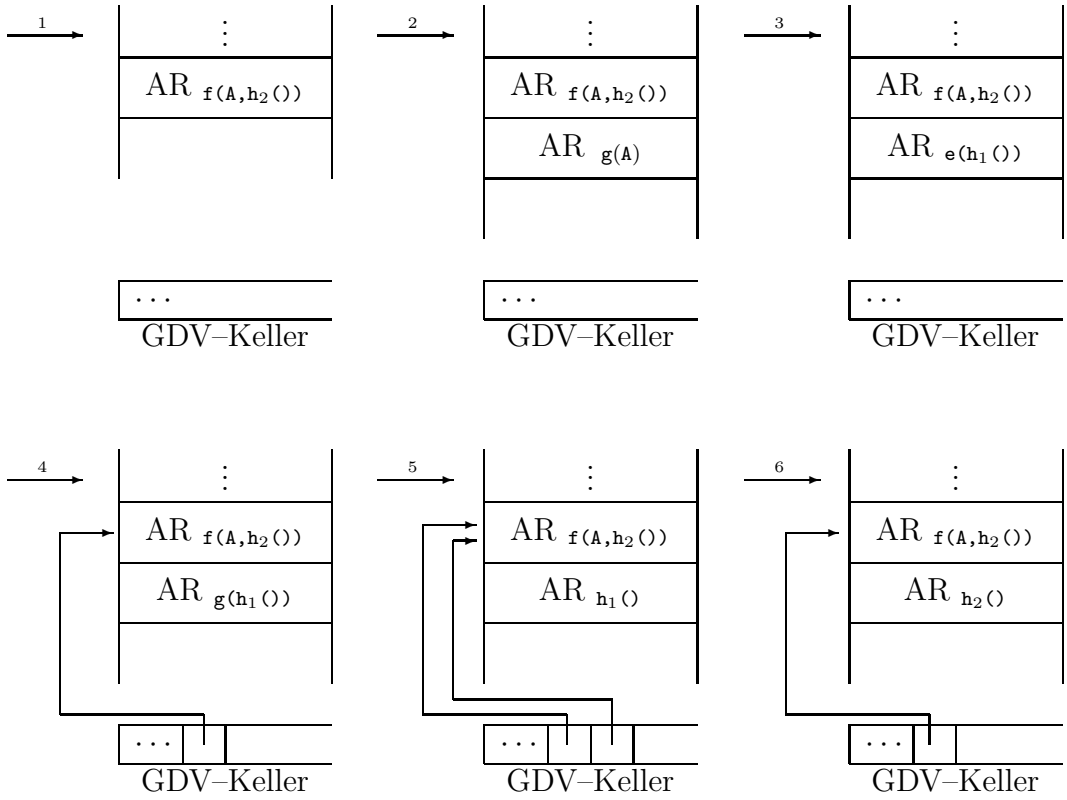
- Falls derjenige Aufruf beendet ist durch den ein GDV–Verweis gesetzt wurde, so wird der oberste GDV–Keller–Eintrag durch eine Pop–Operation wieder gelöscht.

Aus der Linkage eines Activation–Records verschwindet somit der 1. Linkage–Eintrag „Generalisierter dynamischer Vorgänger“ (siehe Abbildung 2.2), und wir erhalten damit folgenden neuen allgemeinen AR–Aufbau:

Abbildung 6.1: Activation Record ohne GDV–Eintrag



Bevor wir zur Relevanz des neuen GDV–Verweises, der dicken Parameter–Optimierung und des neuen GDV–Kellers kommen, wollen wir noch die korrekte Abarbeitung von Beispiel 6.5 mit Hilfe des GDV–Kellers vorführen:

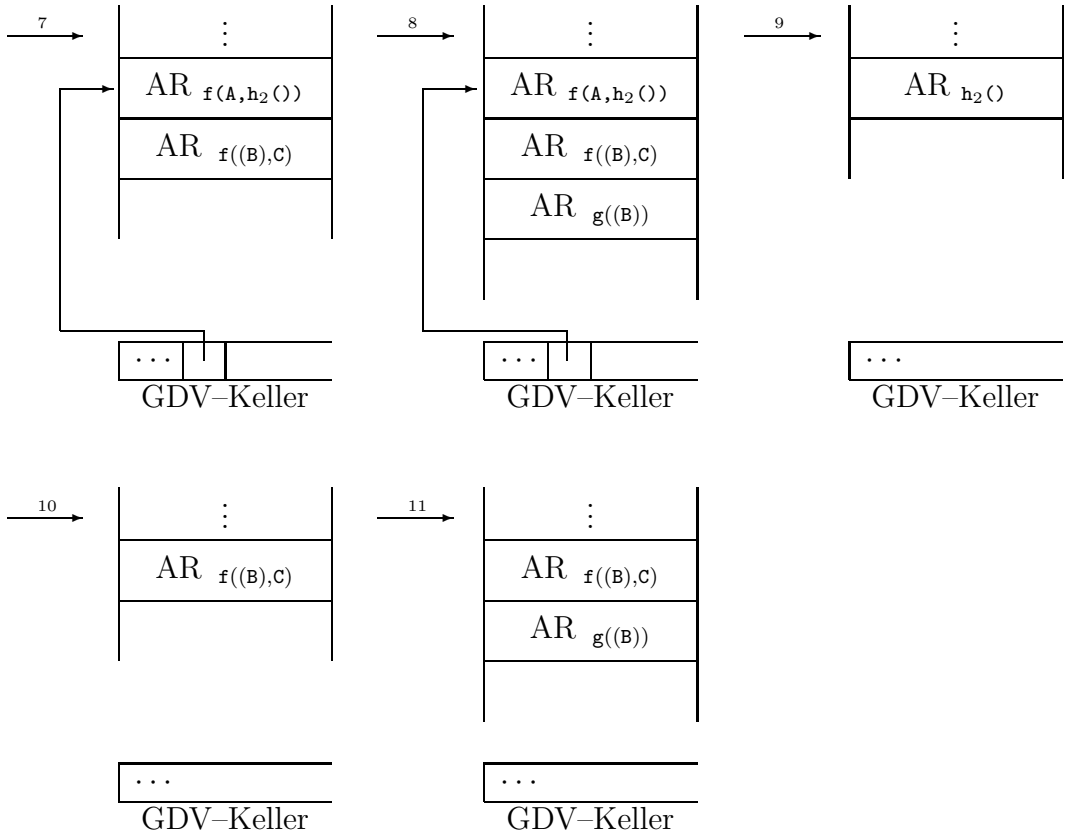


zu 4. Der GDV–Verweis zeigt nun nicht mehr aus der Linkage vom $AR_{g(h_1())}$, sondern aus dem neuen GDV–Keller heraus auf das $AR_{f(A,h_2())}$.

zu 5. Nun wird das $AR_{g(h_1())}$ gelöscht, jedoch der GDV–Verweis bleibt so lange im GDV–Keller stehen, bis der Aufruf $g(h_1())$ beendet ist. Der nächste GDV–Verweis wird ebenfalls in den GDV–Keller „gepusht“.

zu 6. Die in Schritt 5 aufgerufene Hilfsfunktion h_1 zum dicken Parameter CONS(D,E) ist ausgeführt worden, und der dadurch angelegte GDV–Verweis kann von der GDV–Kellerspitze gepopt werden. Übrig bleibt der erste GDV–Verweis, und das $AR_{f(A,h_2())}$ kann weiterhin nicht gelöscht werden.

Die weitere korrekte Abarbeitung des Beispiels ist gesichert:



In Schritt 9 konnte der gültige Parameter ermittelt werden, und das Beispiel liefert schließlich das korrekte Endergebnis ((A (B).C)(B).C).

Im nächsten Abschnitt sehen wir, daß die Optimierung von weiteren, bisher nicht in der HOpt–Klasse enthaltenen NSF–Aufrufen, zu erheblichen Kosteneinsparungen führen kann.

6.4 Die Relevanz der GDV–Optimierung

Wie wir in den Beispielen 6.1, 6.3 und 6.5 gesehen haben, werden durch die in diesem Kapitel vorgestellten Optimierungstechniken viele NSF–Aufrufe optimiert, die bisher *nicht* in der HOpt–Klasse enthalten waren.

Zum einen wird dies durch die in Abschnitt 6.1 vorgestellte GDV–Optimierung erreicht: Durch die in einem Programm eingebundenen GMNIV–Markierungen wird nicht mehr notwendig der dynamische Vorgänger zu einem GDV. Die Situation gemäß Lemma 4.1 ist soweit „entschärft“ worden, daß der GDV–Verweis nur noch diejenigen AR's im AR–Keller festhält, die noch benötigt werden könnten — AR's, die aus statischer Sicht sicher nicht mehr benötigt werden, werden durch den neuen GDV–Verweis freigegeben.

Zum anderen können nun Aufrufe optimiert werden, die bisher entsprechend der Situation in Lemma 4.3 durch Typ–2 Parameter eine frühzeitige

Speicherplatzfreigabe verhindert haben: Durch die in Abschnitt 6.2 vorgestellte dicke Parameter–Optimierung können die Hilfsfunktionen zu dicken Parametern mit einem ggf. niedrigeren statischen Niveau übersetzt werden und dadurch die statischen Verweise aus Typ 2–Parameterzellen (siehe Tabelle 2.1) auf AR's weiter unten im AR–Keller verweisen.

Durch die Einführung des GDV–Keller wird die Linkage eines jeden AR um 20% kleiner. Das Verschieben von AR's nach einer zunächst provisorischen Anlage geht somit schneller und eine HOpt wird günstiger. Ferner wird oft Speicherplatz gespart, da nicht mehr unnötig viele Kopien der selben GDV–Information im AR–Keller stehen können — Redundanz wird vermieden.

Der Nachteil, daß ein weiterer Laufzeitkeller eingerichtet werden muß, kann durch die Integration des GDV–Kellers in den bereits vorhandenen RA–Keller entschärft werden: Da RA–Keller und GDV–Keller die gleiche Datenstruktur besitzen, können sie in einem gemeinsamen Speicherbereich aufeinander zuwachsen (siehe Abbildung 10.2 auf Seite 157).

Wir wollen die tatsächlichen Einsparungen der Beispiele betrachten. Zuvor jedoch folgende

Bemerkung 6.12 Nachdem in diesem Kapitel der GDV–Keller eingeführt wurde, bedeutet die Angabe der *maximalen Kellertiefe* bei denjenigen Optimierungsstufen, die sich auf GMARK stützen und somit einen GDV–Keller benötigen, die Angabe der maximal erreichten Tiefe von AR–Keller + GDV–Keller. Im GDV–Keller steht dabei immer mindestens der Verweis auf das AR vom Hauptprogramm.

Beispiel 6.1 (Fortsetzung): Für das Beispiel, welches zur Motivation des neuen GDV diente, betrachten wir zum einen den Aufruf f(A) sowie zum anderen den 5–fach geschachtelten Aufruf f(f(f(f(f(A))))). Dann ergeben sich folgende Werte:

	Maximale Kellertiefe	
Optimierungsstufe	f(A)	f(f(f(f(f(A)))))
Keine Optimierung	19	47
Definitonsgemäßer GDV	19	47
Einführung des GDV–Kellers	18	46
Dicke Param. Opt. mittels GMARK	18	46
GDV–Verweis mittels GMARK	11	11

⇒ Für den Aufruf f(A) ergibt sich ein um ca. 42% geringerer Speicherbedarf. Für den Aufruf f(f(f(f(f(A))))) potenziert sich die Einsparung auf gut 76%, und der Speicherplatz–Mehraufwand für den geschachtelten Aufruf ist somit gänzlich verschwunden!

Beispiel 6.3 (Fortsetzung): Dieses Beispiel diente zu der Motivation der dicker Parameter–Optimierung. Der Aufruf g((X Y)) wird wie folgt optimiert:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	25
Definitonsgemäßer GDV	25
Einführung des GDV–Kellers	23
Dicke Parameter Opt. mittels GMARK	17
GDV–Verweis mittels GMARK	17

⇒ Es ergibt sich zusammen ein um 32% geringerer Kellerspeicher–Bedarf.

Beispiel 6.5 (Fortsetzung): Das Beispiel, welches zu der Motivation des GDV–Kellers diente, wird für den Aufruf f(A,f((B),C)) wie folgt optimiert:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	42
Definitonsgemäßer GDV	42
Einführung des GDV–Kellers	38
Dicke Parameter Opt. mittels GMARK	32
GDV–Verweis mittels GMARK	26

⇒ Es ergibt sich zusammen ein ca. 38% geringerer Kellerspeicher–Bedarf.

Bemerkung 6.13 Eine Sammlung weiterer Beispielprogramme und die Dokumentation ihres Kellerbedarfs findet sich in Anhang A.1.

Bemerkung 6.14 In dem dieser Arbeit zugrundeliegenden Sprachumfang von Pure–LISP können GDV–Verweise nur in linken Ästen, d.h. im if–Teil von Konditionals, sowie im ersten Parameter von CONS– bzw. EQ–Aufrufen gesetzt werden. Aufrufe aus diesen Positionen sind in der funktionalen, bzw. applikativen Programmierung in nahezu jedem Programm zu finden. Wird in einer umfangreicheren Implementation die Anzahl der mehrstelligen Standardfunktionen erhöht, so erhöht sich auch entsprechend die Anzahl möglicher linker Äste in einem Programm und somit auch die Relevanz der GDV–Optimierung.

Bemerkung 6.15 Für die Relevanz der GDV– und dicker Parameter–Optimierungen ist entscheidend, daß sie statischer Natur ist. Die Markierung des Programmtextes wird bereits vom Compiler mittels GMARK vorgenommen und stellt damit einen unerheblichen Mehraufwand dar. Für das Laufzeitsystem, d.h. dynamisch, beschränkt sich der Aufwand auf den Zugriff der GMNIV–Markierungen, was ebenfalls vernachlässigt werden kann. Die DPNIV–Markierungen sind in einem übersetzten Programm in der Form nicht mehr vorhanden, da sie lediglich während der Compilation zur Bestimmung der statischen Niveaus von Hilfsfunktionen dienten. Daß umgekehrt im allgemeinen kein Laufzeitgewinn zu erwarten ist, ist klar, da die eigentlich

aufwendigen Handlungen, nämlich das Bestimmen des maximalen Verweises aus einem zunächst provisorisch angelegten AR und das eventuell anschließende Versetzen dieses AR's an einen tieferen Platz im AR–Keller, natürlich durch unsere Technik nicht entfallen (siehe auch Bemerkung 5.5).

Im nächsten Abschnitt wollen wir den GMARK–Markierungsalgorithmus vorstellen:

6.5 Der GMARK Markierungsalgorithmus

Für die in den Abschnitten 6.1 und 6.2 eingeführten statischen Programmanalysen wollen wir nun einen Algorithmus beschreiben, der die beschriebenen Markierungen am Programmtext vornimmt. Der Compiler kann die Informationen dann bei der Übersetzung lesen und entsprechend in den Zielcode aufnehmen. Zur Laufzeit hat das Laufzeitsystem dann die nötigen Informationen für die oft kostengünstigerere Abarbeitung des Programms zur Verfügung.

Bemerkung 6.16 In Anlehnung an den in [Fe87] (S.223) vorgestellten Markierungsalgorithmus MARK, werden wir den GMARK–Markierungsalgorithmus an dieser Stelle für die Datensprache von LISP (siehe [Fe87], S.23) in einer „informellen Notation“ vorstellen. Dies hat im wesentlichen folgende Gründe:

1. Am Ende dieses Abschnittes geben wir den GMARK–Algorithmus als lauffähiges XLISP–Programm an. Da die Datensprache von LISP selber auf der Datenstruktur „Liste“ basiert ist die Manipulation eines LISP–Programms wesentlich einfacher, als dies bei einem LISP/N–Programm der Fall wäre.

2. Im Anhang B.2 dieser Arbeit ist der modifizierte und mit dem Markierungsalgorithmus versehene Honschopp–Compiler wiedergegeben. Dort ist der GPMARK–Markierungsalgorithmus (d.h. der GMARK–Algorithmus *zusammen* mit der in Kapitel 7 vorgestellten Optimierung) in Pascal formuliert, und operiert dann auf dem Zwischencode echter LISP/N–Programme.

3. Durch die Wahl der Sprache LISP in diesem Abschnitt sowohl als Programmiersprache für GMARK als auch für das zu markierende Eingabeprogramm wird die Portabilität der hier vorgestellten Techniken von LISP/N auf andere applikative Sprachen ersichtlich.

6.5.1 Beschreibung von GMARK

Der GMARK–Algorithmus durchläuft das Eingabeprogramm von rechts nach links nach der Methode des rekursiven Abstiegs. Dabei werden Nichtstan-

dardidentifikatoren derart gesammelt, daß sich das maximale statische Niveau zum einen für relevante lokale Kontexte und zum anderen für aktuelle dicke Parameter bestimmen läßt:

Sei Π ein LISP–Programm, dann wird Π unter gleichzeitiger Führung von

- einer Menge $\mathcal{M}$ von Nichtstandardidentifikatoren, welche die Identifikatoren des gerade aktuellen $\mathcal{RLK}$ enthält,
- einer Assoziationsliste $\mathcal{A}$ („A–Liste“), welche alle aktuell erreichbaren Identifikatoren und ihre statischen Niveaus in der Form (id,SN(id)) enthält,
- eines Zählers niv, welcher die aktuelle Schachtelungstiefe der Rümpfe angibt und mit 0 initialisiert wird und
- eines Zählers lac, welcher die aktuelle Schachtelungstiefe von linken Ästen angibt und mit 0 initialisiert wird,

gemäß der folgenden Regeln $1-6$ *von rechts nach links* durchlaufen:

1. (a) Beim Eintritt in eine Label–Funktion, d.h. die Deklaration einer Funktion g=(`LABEL` $\mathtt{g}_{id}$ (`LAMBDA` ($x_1 \ldots x_n$) r)), wird $\mathcal{A}$ zunächst gerettet und dann $\mathcal{A}$:=$\mathcal{A}\cup(g_{id}, niv)$ gesetzt und mit der Bearbeitung der Lambda–Funktion fortgefahren.
   (b) Beim Eintritt in eine Lambda–Funktion (`LAMBDA` ($x_1 \ldots x_n$) r) wird $\mathcal{M}$:=$\mathcal{M}\cup$„LAMBDA“ gesetzt (siehe Bemerkung 6.17). Anschließend werden $\mathcal{M}$ und $\mathcal{A}$ gerettet und dann $\mathcal{M}:=\emptyset$ (Beschränkung auf den $\mathcal{LK}$), niv:=$niv+1$ und dann $\mathcal{A}$:=$\mathcal{A}\cup\{(\text{LAMBDA}, niv), (x_1, niv), \ldots, (x_n, niv)\}$ gesetzt und mit der Bearbeitung von r fortgefahren.
   (c) Beim Austritt aus einer Lambda–Funktion wird niv:=$niv-1$ gesetzt und der Durchlauf von Π mit den gemäß (1.b) geretteten Werten von $\mathcal{M}$ und $\mathcal{A}$ fortgesetzt.
   (d) Beim Austritt aus einer Label–Funktion wird der Durchlauf von Π mit dem gemäß (1.a) gerettetem Wert von $\mathcal{A}$ fortgesetzt.
2. Beim Durchlaufen von r wird jeder auftretende Nichtstandardidentifikator ω, der noch nicht in $\mathcal{M}$ enthalten ist, zu $\mathcal{M}$ hinzugefügt, d.h. $\mathcal{M}$:=$\mathcal{M}\cup\omega$.
3. Ein Aufruf von QUOTE wird nicht bearbeitet.
4. (a) Beim Eintritt in den ersten Parameter eines Aufrufs von CONS oder EQ wird lac:=lac+1 gesetzt (Bearbeitung eines linken Astes).
   (b) Beim Austritt aus einem linken Ast von CONS oder EQ wird lac durch lac:=$lac-1$ wieder zurückgesetzt.

(a) Beim Eintritt in einen aktuellen Parameter a_i eines NSF–Aufrufs wird $\mathcal{M}$ gerettet, $\mathcal{M}$:=$\emptyset$ gesetzt (Beschränkung auf den $\mathcal{LK}$) und mit der Bearbeitung von a_i fortgefahren.

(b) Falls der aktuelle Parameter a_i in Punkt (4.a) ein dicker Parameter δ ist, so ist beim Austritt aus δ $\mathcal{M}$= {NS-Idf. $\in \delta$}. Für jeden Nichtstandardidentifikator $\omega \in \mathcal{M}$ wird dann aus der A–Liste $\mathcal{A}$ das zugehörige statische Niveau SN(ω) oder 0 (falls $\omega \notin \mathcal{A}$) ermittelt, und das Maximum über diese Werte gebildet. Die so bestimmte Information DPNIV(δ) wird direkt links von δ in der Form ($*$ DPNIV(δ) + 100 $*$) eingefügt.

(c) Beim Austritt aus einem aktuellen Parameter a_i wird mit $\mathcal{M}$:=$\mathcal{M}\cup${gemäß (4.a) gerettete Menge $\mathcal{M}$} fortgefahren.

5. (a) Beim Eintritt in eine bedingte Form (Konditional) `BF` (siehe auch Bemerkung 2.11) in r wird $\mathcal{M}$ zunächst gerettet.

(b) Für die Bearbeitung einer Bedingung `BE` von `BF` wird lac:=$lac+1$ gesetzt (Bearbeitung eines linken Astes).

(c) Beim Eintritt in eine Folgerung `FO` von `BF` wird der aktuelle Wert von $\mathcal{M}$ gerettet, und für das Durchlaufen von `FO` wird der gemäß (5.a) gerettete Wert von $\mathcal{M}$ zugrundegelegt (Ausblenden der übrigen Klauseln von `BF` entsprechend Definition 6.2).

(d) Beim Austritt aus `FO` wird die gemäß (5.b) gerettete Menge wieder zu $\mathcal{M}$ hinzugefügt.

6. Beim Erreichen eines angewandten Vorkommens (id, k) eines Nichtstandardidentifikators id in r innerhalb eines linken Astes (Kriterium für linken Ast: $lac > 0$) ist nun $\mathcal{M}$= {NS–Idf. $\in \mathcal{RLK}(id, k)$}. Für jeden Nichtstandardidentifikator $\omega \in \mathcal{M}$ wird aus der A–Liste $\mathcal{A}$ das zugehörige statische Niveau SN(ω) oder 0 (falls $\omega \notin A$) ermittelt, und das Maximum über diese Werte gebildet. Die so bestimmte Information GMNIV(id, k) wird direkt rechts von id in der Form ($*$ GMNIV(id, k) $*$) eingefügt.

Bemerkung 6.17 Kommentare zu einigen Punkten der GMARK–Beschreibung:

Zu 1.b Der Hilfsidentifikator „LAMBDA“ wird benutzt, um die Existenz einer Lambda–Funktion im $\mathcal{RLK}$ zu protokollieren. Dadurch, daß der Hilfsidentifikator die r umfassende kleinste NSF g als statischen Vorgänger hat, kann ein optimierter GDV–Verweis nicht den statischen Vorgänger der Lambda–Funktion, der mit g identisch ist, frühzeitig löschen. In LISP/N können Funktionsdeklarationen nicht in einem Anweisungsteil vorkommen, und ein ähnlicher Hilfsidentifikator ist nicht nötig.

Zu 3. Die SF QUOTE ist in LISP/N nicht notwendig, da wegen der Call By Name–Semantik die Argumente eines NSF–Aufrufs sowieso frühstens im Anweisungsteil der gerufenen Funktion ausgewertet werden.

Zu 6. Damit der GMARK–Algorithmus in einem Durchlauf die Marken für die GDV– *und* dicke Parameter–Optimierung ermitteln kann, wird zusätzlich ein Flag *ins* verwaltet, welches die angedeutete Ablaufsteuerung vornimmt: Hat *ins* den Wert „false“, so können keine Marken in den Programmtext eingefügt werden. Dadurch, daß nur Nichtstandardidentifikatoren auf linken Ästen markiert werden, werden unnötige GMNIV–Markierungen vermieden.

6.5.2 Beispiele

Wir verdeutlichen die Wirkungsweise des GMARK–Markierungsalgorithmuses an zwei Beispielen:

Beispiel 6.6 Gegeben sei das LISP–Programm LSTO, welches in einem Eingabebaum nach linksunten absteigt (vgl. Beispiel „lsto.lsp“ im Anhang A.1):

```
(LABEL LSTO (LAMBDA (tree)
   (COND ( (ATOM tree) tree )
         ( T           (CONS (LSTO (CAR tree)) NIL) )) ))
```

Nachdem GMARK das Programm markiert hat, sieht es folgendermaßen aus:

```
(LABEL LSTO (LAMBDA (tree)
   (COND
     ( (ATOM tree (* 1 *) ) tree )
     ( T  (CONS (LSTO (* 0 *) (* 101 *) (CAR tree (* 0 *) )) NIL) )) ))
```

Das nächste Beispiel ist ein wenig umfangreicher:

Beispiel 6.7
Gegeben sei folgendes LISP–Programm (vgl. Beispiel „aufg34.lsp“ im Anhang A.1):

```
(LAMBDA (x y)
  ((LAMBDA (g) (CONS ((LAMBDA (y) (g y)) 'A) (CONS (g x) (CONS x y)))
   ) (LABEL h (LAMBDA (x)
       (COND ((ATOM x) ((LAMBDA (h) (CONS h 'B)) y))
             (T        ((LAMBDA (y)
                         (CONS (h 'C) (CONS (h 'D) (CONS x y)))) 'E)))))
  ))
```

Wird nun GMARK auf dieses Beispiel angewendet, so erhalten wir:

```
(LAMBDA (x y)
  ((LAMBDA (g) (CONS ((LAMBDA (y) (g (* 0 *) y (* 0 *) )) (* 2 *) 'A)
                     (CONS (g (* 1 *) x (* 0 *) ) (CONS x (* 1 *) y)))
   ) (* 101 *)
     (LABEL h (LAMBDA (x)
       (COND ((ATOM x (* 2 *) ) ((LAMBDA (h) (CONS h (* 0 *) 'B)) y))
             (T ((LAMBDA (y)
                   (CONS (h (* 3 *) 'C) (CONS (h (* 3 *) 'D)
                                              (CONS x (* 3 *) y)))) 'E)))))
  ))
```

Definition 6.8 Ein durch GMARK markiertes Programm Π bezeichnen wir im folgenden mit Π^M.

Bemerkung 6.18 Die durch GMARK markierten Programme Π^M sind natürlich für den Programmierer weniger leserlich. Da es sich dabei aber lediglich um eine Zwischenstufe zur Erzeugung des Zielcodes handelt, wird der Programmierer mit dem markierten Text in der Regel nicht konfrontiert.

6.5.3 GMARK als XLISP–Programm

Wir wollen den oben beschriebenen GMARK–Algorithmus nun in Form eines lauffähigen XLISP–Programms angeben. Der Aufruf von GMARK erfolgt mit `(GMARK 'Π)`, wobei Π das zu markierende (gequotete) LISP–Programm in Datensprache ist:

```
(defun gmark (exp) (mark exp nil 0 nil exp t 0))

(defun mark (exp vars niv alist oldexp ins lac)
  (cond
    ((atom exp)
      (cond ((member exp '(T F NIL)) vars)
            ((and ins (> lac 0))     (prog1
                                       (union exp vars)
                                       (rplacd
                                         oldexp
                                         (cons (list '* (max vars alist 0) '*)
                                               (cdr oldexp)))))
            (t                        (union exp vars))))
    ((atom (car exp))
      (cond
       ((eq (car exp) 'QUOTE)  vars)
       ((eq (car exp) 'COND)   (condmark (cdr exp) vars niv alist ins lac))
       ((eq (car exp) 'LABEL)  (mark (caddr exp) nil niv
                                     (cons (cons (cadr exp) niv) alist)
                                     (cddr exp) ins lac))
       ((eq (car exp) 'LAMBDA) (union 'LAMBDA
                                      (mark
                                        (caddr exp) nil (1+ niv)
                                        (append (para (cons 'LAMBDA (cadr exp))
                                                      (1+ niv)) alist)
                                        (cddr exp) ins lac)))
```

```
       ((member (car exp) '(CAR CDR ATOM))
          (listmark (cdr exp) vars niv alist ins nil nil lac))
       ((member (car exp) '(CONS EQ))
          (mark (cadr exp)
                (listmark (cddr exp) vars niv alist ins nil nil lac)
                niv alist (cdr exp) ins (1+ lac)))
       (t (prog1 (union (car exp)
                        (listmark (cdr exp) vars niv alist ins t nil lac))
                 (cond ((and ins (> lac 0))
                          (rplacd exp (cons (list '* (max vars alist 0) '*)
                                            (cdr exp))))))))))
    (t (prog1 (mark (car exp) (listmark (cdr exp) vars niv alist ins t nil lac)
                    niv alist exp ins lac)
              (cond ((and ins (> lac 0))
                       (rplacd exp (cons (list '* (max vars alist 0) '*)
                                         (cdr exp))))))))))

(defun listmark (l vars niv alist ins nsf tmp lac)
  (cond
    ((null l) vars)
    (t        (prog2
                (cond ((and ins nsf
                            (cond ((atom (car l)) nil)
                                  (t              (not (eq (caar l) 'QUOTE)))))
                  (setq tmp (mark (car l) nil niv alist l nil lac))))
                (cond
                  (nsf (union (listmark (cdr l) vars niv alist ins nsf nil lac)
                              (mark (car l) nil niv alist l ins lac)))
                  (t   (mark (car l)
                             (listmark (cdr l) vars niv alist ins nsf nil lac)
                             niv alist l ins lac)))
                (cond ((and ins nsf
                            (cond ((atom (car l)) nil)
                                  (t              (not (eq (caar l) 'QUOTE)))))
                  (rplaca (rplacd l (cons (car l) (cdr l)))
                          (list '* (+ (max tmp alist 0) 100) '*)))))))))

(defun condmark (l vars niv alist ins lac)
  (cond ((null l) vars)
        (t        (mark (caar l)
                        (union
                          (condmark (cdr l) vars niv alist ins lac)
                          (mark (cadar l) vars niv alist (cdar l) ins lac))
                        niv alist (car l) ins (1+ lac)))))

(defun union (neu vars)
  (cond ((atom neu) (cond ((member neu vars) vars)
                          (t                (cons neu vars))))
        (t          (union (cdr neu) (union (car neu) vars)))))

(defun para (l niv)
  (cond ((null l) nil)
        (t        (cons (cons (car l) niv) (para (cdr l) niv)))))
```

```
(defun max (vars alist m)
  (cond ((null vars) m)
        ((> (niveau (car vars) alist) m)
                     (max (cdr vars) alist (niveau (car vars) alist)))
        (t           (max (cdr vars) alist m))))

(defun niveau (var alist)
  (cond ((null alist)          0)
        ((eq (caar alist) var) (cdar alist))
        (t                     (niveau var (cdr alist)))))
```

Im nächsten Kapitel werden wir die Informationen, die uns der GMARK–Markierungsalgorithmus zur Verfügung stellt, als Grundlage zu einer weiteren Optimierung nutzen.

Kapitel 7

Optimierung durch Parameterpermutation

Im letzten Kapitel haben wir den GDV–Markierungsalgorithmus GMARK vorgestellt. Er basiert auf einer Erkenntniss aus [Fe87], daß sich statisch, also bereits *vor* der eigentlichen Laufzeit, durch den sogennanten *relevanten lokalen Kontext* für einen Aufruf c im Anweisungsteil einer NSF festlegen läßt, welche Bereiche des Anweisungsteils bei der lokalen Fortsetzung von c noch erreicht werden könnten, und welche *sicher* nicht.

Wir sehen also, daß der Grad der möglichen GDV–Optimierung vom syntaktischen Umfeld eines Aufrufs abhängt, und somit also „rechtskontext–sensitiv“ ist.

In diesem Kapitel wollen wir daher untersuchen, ob wir zu kostengünstigereren Aufrufen auf linken Ästen kommen, wenn wir durch Manipulation am Kontext die Reihenfolge bei der Auswertung der Äste mehrstelliger Standardfunktionen permutieren.

Dabei werden wir dann feststellen, daß die statischen Vorinformationen, die der GMARK–Algorithmus liefert, zusammen mit gewissen Entscheidungskriterien ausreichen, um durch mögliche Permutationen die Relevanz der GDV–Optimierung aus Abschnitt 6.1 weiter zu erhöhen und damit die Situation gemäß Lemma 4.1 noch weiter zu „entschärfen“.

Der Permutations– und Markierungsalgorithmus GPMARK wird dann am Ende dieses Kapitels vorgestellt.

Betrachten wir zunächst folgendes Beispiel, um die späteren Schritte zu motivieren:

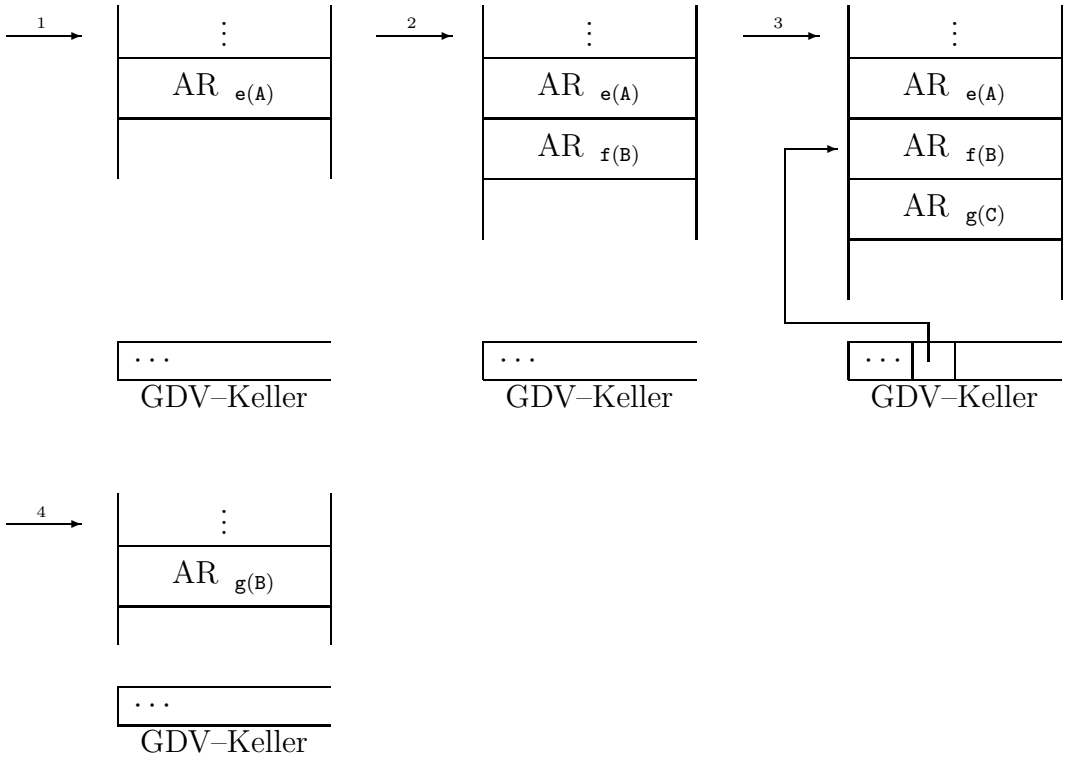
Beispiel 7.1 (vgl. Beispiel „perm.lsp“ im Anhang A.1)

```
g = λ x . { x }
e = λ y . { f = λ z . { CONS ( g(C) , g(z) ) }
            f(B) }
e(A)
```

Wir wollen dazu gleich die durch GMARK markierte Version angeben:

```
g = λ x . { x }
e = λ y . { f = λ z . { CONS ( g(*2*)(C) , g(z) ) }
            f(B) }
e(A)
```

Für den Aufruf e(A) sieht der Kellerablauf wie folgt aus:



Im $\mathcal{RLK}$(g(C)) steht der Aufruf g(z) und somit die Nichtstandardidentifikatoren g und z mit den statischen Niveau's 0 und 2. Somit wird der Aufruf g(C) von GMARK mit $(* \ 2 \ *)$ markiert, und der GDV–Verweis auf das $\mathrm{AR}_{f(B)}$ sichert korrekterweise den Wert von z für den Aufruf von g im rechten Ast.

Für die Bestimmung des Endresultats (die Liste (C . B)) müssen bis zu drei AR's gleichzeitig im AR–Keller gehalten werden.

Wir werden später sehen, daß obiges Beispiel durch einen „Parametertausch" mit maximal zwei gleichzeitig im AR–Keller gehaltenen AR's ausgeführt werden kann. Zuvor führen wir die dazu nötigen Grundlagen im folgendem Abschnitt ein.

7.1 Optimierung durch Parametertausch

Aufgrund der Auswertungsstrategie Call By Name von LISP/N liegt lediglich bei SF–Aufrufen bereits statisch die Auswertungs–Reihenfolge ihrer aktuellen Parameter zur Laufzeit fest: Nur SF–Aufrufe werden gemäß Call By Value ausgeführt und ihre Argumente sofort ausgewertet (siehe Bemerkung 1.9).

Da die Auswertungsreihenfolge von aktuellen Parametern für die Optimierung in diesem Kapitel von entscheidender Bedeutung ist, beschränken wir uns daher im folgenden auf die mehrstelligen SF von LISP/N. Dazu folgende

Bemerkung 7.1 In LISP/N mit dem Sprachumfang von Pure–LISP sind die einzigen n–stelligen Standardfunktionen f_s mit $n \geq 2$ die beiden 2–stelligen Standardfunktionen CONS und EQ.

Unter Berücksichtigung, daß LISP/N mit dem Sprachumfang von Pure–LISP und statischer Variablenbindung *seiteneffektfrei* ist und daß aufgrund der geltenden Church–Rosser Eigenschaft die Auswertungsreihenfolge von aktuellen Parametern für das Ergebnis eines Aufrufs (falls es existiert!) keine Rolle spielt, können wir folgende neue SF einführen:

Definition 7.1 Sei $c = \text{CONS}(a_1, a_2)$ ein Aufruf der 2–stelligen SF CONS mit den beliebig aber festen Parametern a_1 und a_2. Dann führen wir die 2–stellige SF *PCONS* durch folgende operational definierte Semantik ein:

$$[\![\texttt{PCONS}(a_1, a_2)]\!] \;:=\; [\![\texttt{CONS}(a_2, a_1)]\!] .$$

PCONS kompensiert somit die durch ein Vertauschen der Parameter bei CONS veränderte Semantik (siehe auch Definition 1.28). Wir verdeutlichen dies an folgendem

Beispiel 7.2

```
CONS(A,B)                       = PCONS(B,A)
                                = (A.B)

CONS(CONS((A B),(C)),(D E)) = PCONS((D E),CONS((A B),(C)))
                                = CONS(PCONS((C),(A B)),(D E))
                                = PCONS((D E),PCONS((C),(A B)))
                                = (A B C D E)
```

Die andere mehrstellige SF in LISP/N ist EQ. Hierzu folgende

Bemerkung 7.2 Für die SF EQ brauchen wir eine Funktion PEQ *nicht* einzuführen: Das Ergebnis der Auswertung von EQ ist unabhängig von der Auswertungsreihenfolge der beiden Parameter. Somit gilt für beliebige aber feste a_i aufgrund Definition 1.28:

$$[\![\texttt{EQ}(a_1, a_2)]\!] = [\![\texttt{EQ}(a_2, a_1)]\!] .$$

Wir führen nun folgende Redeweise ein:

Definition 7.2 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s mit $f_s \in \{\text{CONS,EQ}\}$, und sei f_s^p ebenfalls eine 2–stellige SF mit $f_s^p \in \{\text{PCONS,EQ}\}$. Dann sagen wir: Der SF–Aufruf c' geht aus c durch *Parametertausch* hervor, falls gilt:

$$c' = f_s^p(a_2, a_1) \qquad \text{mit } f_s^p := \begin{cases} \text{PCONS , falls } f_s = \text{CONS} \\ \text{EQ , falls } f_s = \text{EQ.} \end{cases}$$

Aufgrund der Definition 7.1 und Bemerkung 7.2 können wir unmittelbar folgende Feststellung machen:

Korollar 7.1 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s in einem echten LISP/N–Programm Π, und sei $c' = f_s^p(a_2, a_1)$ derjenige SF–Aufruf, der aus c durch Parametertausch hervorgeht. Das Programm $\tilde{\Pi}$ gehe aus Π hervor, indem wir den Aufruf c durch c' ersetzen. Dann gilt:

$$[\![\ \Pi\ ]\!] = [\![\ \tilde{\Pi}\ ]\!].$$

Bemerkung 7.3 In dem Korrektheitsbeweis des Honschopp–Laufzeitsystems von Kindler [Ki87] wird eine bestimmte Auswertungsreihenfolge von Parametern nicht benötigt. Somit ist auch von daher ein Parametertausch legitim.

Im folgenden betrachten wir das um die neue SF PCONS ergänzte LISP/N und haben somit insgesamt die sechs Standardfunktionen ATOM, CAR, CDR, CONS, PCONS und EQ zur Verfügung. Die Funktion PCONS ist für den Programmierer natürlich nicht zugänglich, sondern wird ggf. vom Compiler bei der Codeerzeugung eingeführt. Unter welchen Umständen dies erfolgen soll, werden wir im nächsten Abschnitt festlegen.

Zunächst kommen wir jedoch zurück zu unserm Beispiel:

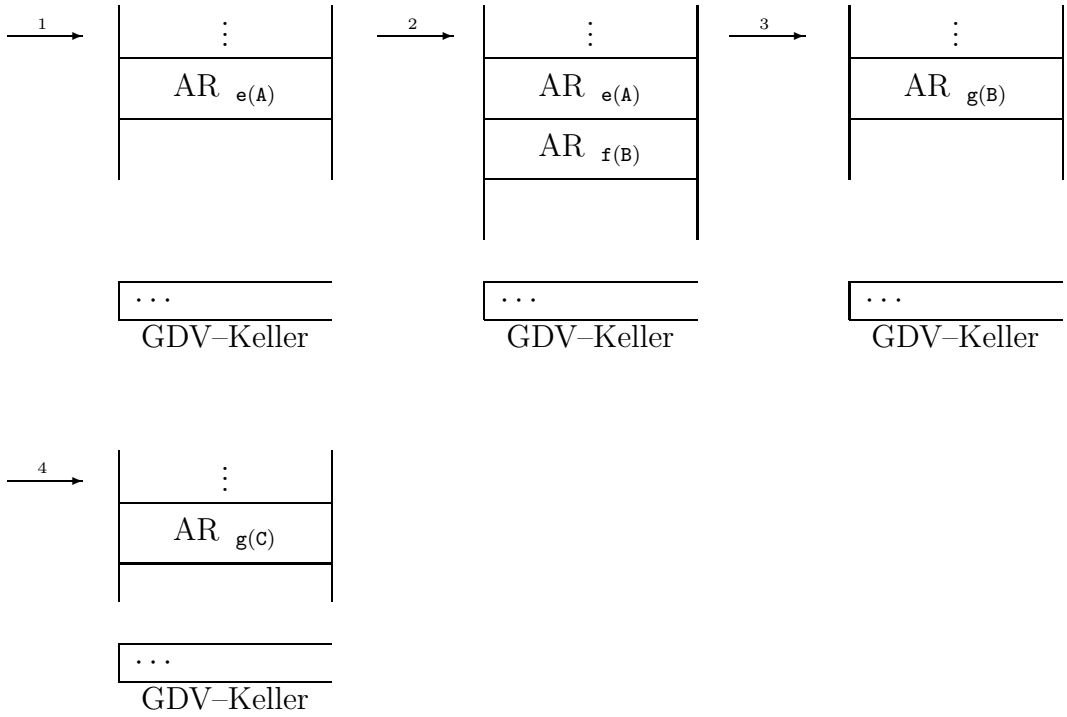
Beispiel 7.1 (Fortsetzung): Permutieren wir die Parameter von CONS und benennen CONS in PCONS um, d.h. führen einen Parametertausch durch, so erhalten wir für f folgende semantisch äquivalente Funktion:

```
f = λ z . { PCONS ( g(z), g(C) ) }
```

Das Programm wir dann von GMARK wie folgt markiert.

```
g = λ x . { x }
e = λ y . { f = λ z . { PCONS ( g(*0*)(z(*0*)), g(C) ) }
            f(B) }
e(A)
```

Der Aufruf im linken Ast ist nun g(z) und im $\mathcal{RLK}$(g(z)) steht der Aufruf g(C) und somit nur der Nichtstandardidentifikator g mit dem statischen Niveau 0. GMARK versieht den Aufruf g(z) somit mit der Marke $(* \ 0 \ *)$ und ein GDV–Verweis braucht *nicht* mehr gesetzt zu werden (siehe Abschnitt 6.1). Wir erhalten somit folgenden Ablauf im AR– und GDV–Keller:



Um zum Endergebnis (C . B) zu gelangen, müssen statt drei nur noch maximal zwei AR's im AR–Keller und kein GDV–Verweis mehr im GDV–Keller gespeichert werden — der Speicherplatzgewinn ist offensichtlich.

Im folgenden werden wir die Kriterien vorstellen, die es bereits *statisch* erlauben über einen möglichen Parametertausch zu entscheiden. Dabei wird der wichtige Grundsatz berücksichtigt, daß eine „Optimierungs"–Technik *keine* Verschlechterung verursachen darf — auch dann nicht, wenn sie sonst bei den meisten Anwendungen eine Verbesserung bewirken würde!

7.2 Die Entscheidungskriterien für den Parametertausch

Betrachten wir zur Laufzeit die Auswertung eines bestimmten NSF–Aufrufs, so wird dafür eine gewisse Anzahl an Speicherzellen im AR–Keller benötigt (mindestens 4 Zellen für die erste Linkage). Dadurch, daß LISP/N mit statischer Variablenbindung und dem Sprachumfang von Pure–LISP *seiteneffekt–frei* ist, machen wir folgende Feststellung:

Korollar 7.2 Sei $c = f(\ldots)$ der Aufruf einer NSF f und sei max_c die während der vollständigen Abarbeitung von c (also einschließlich aller infolge c bewirkter weiterer NSF–Aufrufe) maximal gleichzeitig belegte Anzahl von AR–Keller–Speicherzellen durch c. Falls c in einem aktuellen Parameter eines SF–Aufrufs $c' = f_s(\ldots)$ mit $f_s \in \{\text{CONS,EQ}\}$ vorkommt, so ist max_c *invariant* gegenüber einem Parametertausch von c'.

Die für einen Aufruf konkret benötigte *Anzahl* von AR–Speicherzellen bleibt somit durch einen Parametertausch konstant, läßt sich aber im allgemeinen statisch nicht vorhersagen:

Bemerkung 7.4 Die *Anzahl* der Speicherzellen, die für die Auswertung eines Aufrufs benötigt werden, ist im allgemeinen statisch *nicht* vorhersehbar (sie kann z.B. von den aktuellen Eingabedaten des Programms abhängen).

Im folgenden werden wir, jeweils motiviert durch ein Beispiel, die Kriterien kennenlernen, die zusammen *hinreichend* sind, um zu entscheiden, daß durch einen Parametertausch *keine* Verschlechterung des Speicherplatzverbrauchs eintreten kann, sondern in der Regel Speicherplatz gewonnen wird.
Es sei an dieser Stelle auf die Bemerkungen 6.6 und 6.3 hingewiesen, da wir GMNIV im folgenden auf aktuelle Parameter, d.h. insbesondere auf die Vorkommen der syntaktischen Zeichenreihen dieser Parameter, anwenden wollen.

In Beispiel 7.1 wurde durch den Parametertausch die GMNIV–Markierung des NSF–Aufrufs auf dem linken Ast erniedrigt, und für die Abarbeitung des linken Astes konnte ein entsprechend günstigerer GDV gesetzt werden (in diesem Fall nämlich gar kein GDV!). Diesen Sachverhalt formulieren wir nun als Kriterium für einen evtl. vorzunehmenden Parametertausch:

Lemma 7.1 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s mit den Parametern a_1 und a_2, und für den Aufruf c sei x:=GMNIV(a_1). Sei $c' = f_s^p(a_2, a_1)$, und für den Aufruf c' sei x':=GMNIV(a_2). Falls gilt:

$$x' < x \ , \qquad \textbf{(PK1)}$$

so ist dies ein *notwendiges* Kriterium für den Parametertausch bei c.

Wir werden dieses Lemma im Zusammenspiel mit einem weiteren Lemma in Satz 7.1 beweisen.

Sobald das Permutations–Kriterium PK1 erfüllt ist, wird durch einen Parametertausch ein günstigerer $\mathcal{RLK}$ für mögliche NSF–Aufrufe auf dem linken Ast geschaffen, und der GDV–Verweis bei der Auswertung des betreffenden SF–Aufrufs wäre günstiger.

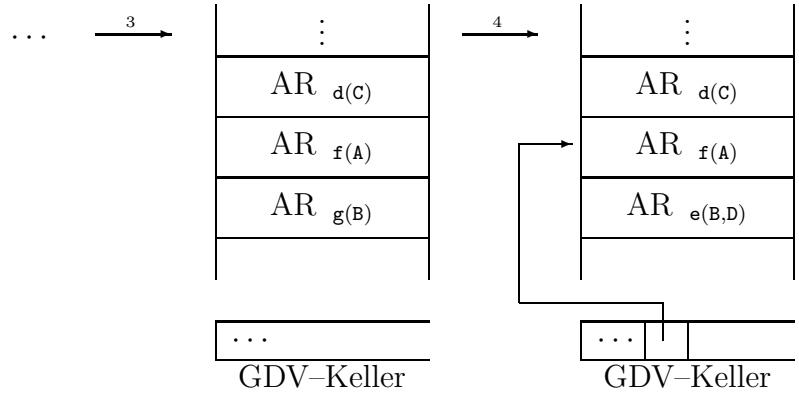
Für unser Beispiel 7.1 trifft dies zu: Mit den gleichen Bezeichnungen wie in Lemma 7.1 gilt $0 = x' < x = 2$ und ein Parametertausch hat den Speicherplatzverbrauch verringert.

Daß dieses Kriterium alleine noch nicht hinreichend ist, um sicherzustellen, daß durch den Parametertausch *keine* Vergrößerung der maximalen Kellertiefe eintreten kann, zeigen uns die folgenden beiden Beispiele:

Beispiel 7.3 Gegeben seien folgende Funktionsdefinitionen (vgl. Beispiel „aperm1.lsp“ im Anhang A.1):

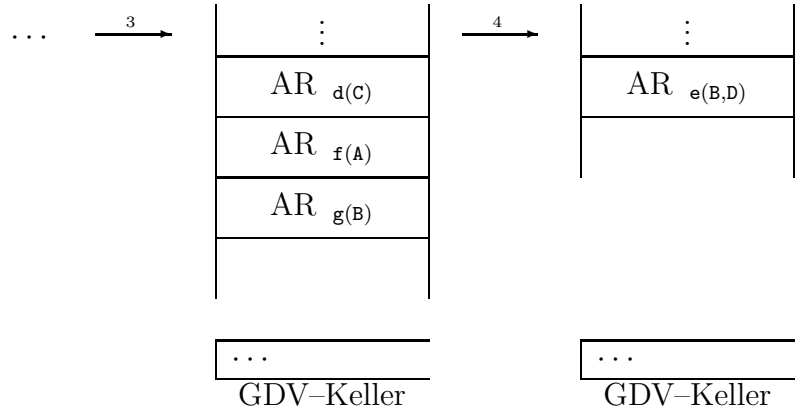
```
e = λ v w . { v }
d = λ z . { f = λ y . { g = λ x . { CONS ( y , e(x,D) ) }
                            g(B) }
            f(A) }
d(C)
```

Für die 2–stellige SF CONS ist das Kriterium PK1 erfüllt. Deshalb betrachten wir zunächst den Ablauf des Programms *mit Permutation* für den Aufruf d(C), und zwar ab der Stelle, ab der PCONS ausgewertet werden muß:



Für die Auswertung des linken Astes von PCONS, d.h. für den Aufruf e(B,D), kann lediglich das $AR_{g(B)}$ frühzeitig freigegeben werden.

Betrachten wir nun den Ablauf des Programms *ohne Permutation* für den Aufruf d(C), so wird das Problem deutlich:



Der Identifikator y auf dem linken Ast von CONS hat als Wert das Atom A, und es findet überhaupt kein Aufruf statt, bei dem ein GDV–Verweis auf

das $AR_{g(B)}$ gesetzt werden müßte. Der nächste Aufruf e(x,D) findet auf dem rechten Ast statt und bewirkt die Freigabe der alten AR's...

Im zweiten Beispiel wird wieder aufgrund des erfüllten Kriteriums PK1 permutiert und, im Gegensatz zu Beispiel 7.3, weiterhin ein NSF–Aufruf auf dem linken Ast ausgeführt. Dennoch tritt eine Verschlechterung durch den Parametertausch ein:

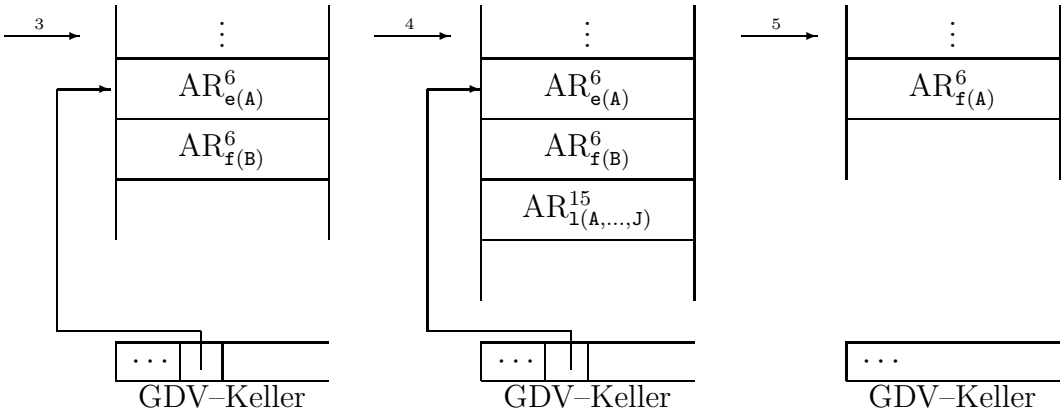
Beispiel 7.4 (vgl. Beispiel „aperm3.lsp“ im Anhang A.1)

```
f = λ x .
       { l = λ y1 ... y10 . { CONS(y1,CONS(y2,...,CONS(y9,y10)...)) }
         IF EQ(x,B) THEN l(A,B,...,J)
                    ELSE K
         FI }
e = λ x .
       { g = y . { CONS( f(x) , f(y) ) }
         g(B) }
e(A)
```

Trotz der günstigeren GDV–Situation kann durch den im allgemeinen *nicht* vorhersehbaren Speicherbedarf eines NSF–Aufrufs eine Verschlechterung durch einen Parametertausch entstehen.

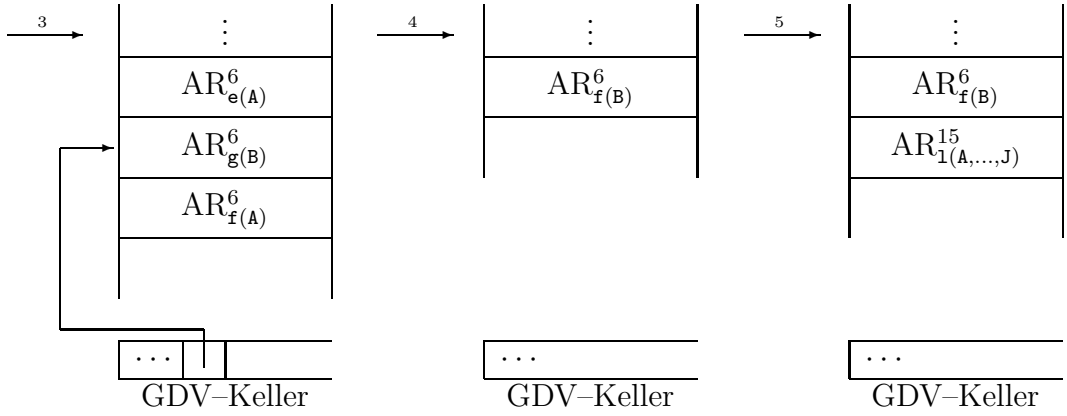
Um dies am Beispiel 7.4 zu verdeutlichen, sind die AR's neben den zugehörigen Aufrufen auch mit der Länge, d.h der Anzahl der benötigten Speicherzellen des AR im AR–Keller, beschriftet.

Für den Aufruf von CONS im Anweisungsteil von g ist das Kriterium PK1 erfüllt. Betrachten wir daher zunächst wieder den Ablauf im AR– und GDV–Keller *mit Permutation* für den Aufruf e(A) ab der Stelle, ab der f(B) auf dem linken Ast von PCONS aufgerufen wird:



Für die Auswertung von e(A) wird die maximale AR–Kellertiefe im 4. Aufruf mit $6 + 6 + 15 = 27$ Speicherzellen erreicht.

Ohne Permutation würde das Beispiel aber wir folgt abgearbeitet:



Im 5. Aufruf wird die maximale AR–Kellertiefe von 6+15 = 21 Speicherzellen erreicht, d.h. 6 Speicherzellen weniger als mit der permutierten Version.

Falls das Kriterium PK1 erfüllt ist, ist zwar sichergestellt (Beweis im Satz 7.1), daß durch einen Parametertausch die GDV–relevanten Aufrufe auf dem linken Ast tiefer als zuvor im AR–Keller ansetzen (falls sie ausgeführt werden müssen — siehe Beispiel 7.3), für den oder diejenigen Aufrufe aber, die dabei vom rechten auf den linken Ast permutiert wurden, kann sich die Position im Keller verschlechtert haben und infolge dessen eine größere Länge des benutzten AR–Kellers eintreten.

Folgendes zweite Kriterium für den Parametertausch soll dies verhindern:

Lemma 7.2 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s mit den Parametern a_1 und a_2, und für den Aufruf c sei $\tilde{x}$:=GMNIV(a_2). Sei $c' = f_s^p(a_2, a_1)$, und für den Aufruf c' sei x':=GMNIV(a_2). Falls gilt:

$$x' = \tilde{x} \ , \qquad \textbf{(PK2)}$$

so ist dies ein *notwendiges* Kriterium für den Parametertausch bei c.

Wir werden dieses Lemma im Zusammenspiel mit Lemma 7.1 in Satz 7.1 beweisen.

Bemerkung 7.5 Das Kriterium PK2 könnte auch als $x' \leq \tilde{x}$ formuliert werden, aber da der $\mathcal{RLK}$ des rechten Astes durch einen Tausch auf den linken Ast lediglich vergrößert werden kann, kann der Fall „<“ nicht eintreten.

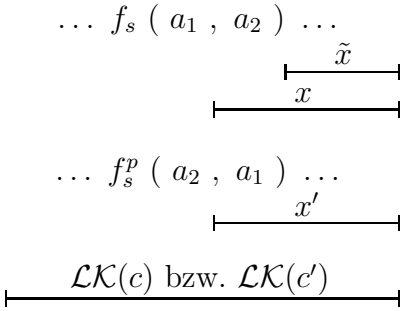
Für unser einführendes Beispiel 7.1 ist auch das Kriterium PK2 erfüllt: Mit den gleichen Bezeichnungen wie in Lemma 7.2 gilt nämlich $0 = x' = \tilde{x} = 0$ und ein Parametertausch würde weiterhin durchgeführt werden.

Zusammen sind die Kriterien PK1 aus Lemma 7.1 und PK2 aus Lemma 7.2 nun *hinreichend* um statisch entscheiden zu können, ob ein Parametertausch zur Laufzeit *keine* Erhöhung der maximal erreichten Kellertiefe bewirken kann. Dazu der entscheidende

Satz 7.1 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen Standarfunktion f_s, und sei $c' = f_s^p(a_2, a_1)$ die permutierte Version von c. Ferner sei *MaxK* die maximal erreichte Kellertiefe bei der Auswertung von c, und sei *MaxK'* die maximale erreichte Kellertiefe bei der Auswertung von c'. Falls für c PK1 *und* PK2 erfüllt sind, so gilt:

$$MaxK' \leq MaxK \ .$$

Beweis (Satz 7.1): Es ist zu zeigen, daß *MaxK'*≤*MaxK* gilt, wenn $x' = \tilde{x} < x$ gilt. Dazu veranschaulichen wir zunächst die zu x, x' und $\tilde{x}$ gehörenden $\mathcal{RLK}$'s durch folgende Abbildung:



Aufgrund der Feststellung in Korollar 7.2 ist die maximal benötigte Anzahl von AR–Kellerspeicher für die Auswertung nur von a_1 bzw. nur von a_2 *invariant* gegenüber einem Parametertausch. Lediglich die maximal erreichte Kellertiefe bei c bzw. c' kann sich ändern, wenn mindestens eine der Auswertungen von a_1 oder a_2 an einer anderen Stelle im AR–Keller beginnt. Durch die Forderung $x' = \tilde{x} < x$ soll ausgeschlossen werden, daß sowohl die Auswertung von a_1 als auch die von a_2 durch einen Parametertausch *nicht* an höherer Stelle im AR–Keller ansetzt, sondern evtl. an tieferer Stelle beginnen kann.

Für jeden NSF–Aufruf wird das zugehörige AR zunächst provisorisch angelegt und dann gemäß Bemerkung 2.8 der GDV–Verweis, ein Verweis auf ein dynamisches Niveau des statischen Vorgänger und alle statischen Verweise aus evtl. vorhandenen Parametern auf den maximalen Verweis hin untersucht. Anschließend wird das AR ggf. an die so ermittelte günstigerere Stelle verschoben, d.h. eine HOpt durchgeführt. Entscheidend ist nun, daß sich durch einen Parametertausch lediglich die GDV–Verweise in den beiden Ästen ändern können, da sich der $\mathcal{RLK}$ entsprechend ändern kann — die statischen Verweise aus Linkage und Parameterbereich bei möglichen NSF–Aufrufen innerhalb von a_1 oder a_2 bleiben unverändert!

Zum Beweis des Satzes zeigen wir nun, daß bei der Abarbeitung sowohl von a_1 als auch bei der von a_2 durch einen Parametertausch kein ungünstigerer GDV–Verweis entstehen kann, wenn PK2 erfüllt ist. Sind PK1 *und* PK2 erfüllt, so ist eine Verringerung der maximalen Kellertiefe bei der Auswertung von c' möglich. (Es reicht an dieser Stelle aus, den $\mathcal{RLK}(a_1)$ bzw. $\mathcal{RLK}(a_2)$ zu

betrachten, da der letztendlich entscheidende $\mathcal{RLK}$ für die in a_1 bzw. a_2 enthaltenen NSF–Aufrufen noch weitere Nichtstandardidentifikatoren enthalten kann, die aber durch x' bzw. x erfasst werden.):

1. Zum $\mathcal{RLK}(a_1)$ gehört *nach* dem Parametertausch nicht mehr der Parameter $a_2 \Rightarrow \tilde{x} \leq x$ und ein Parametertausch kann keinen ungünstigereren GDV–Verweis für die Bearbeitung von a_1 ergeben. (Falls c selber *nicht* auf einem linken Ast vorkommt, ist für a_1 dann *kein* GDV–Verweis mehr notwendig).

2. Zum $\mathcal{RLK}(a_2)$ gehört *nach* dem Parametertausch zusätzlich der Parameter $a_1 \Rightarrow x' \geq \tilde{x}$. Nach Vorraussetzung (PK2) gilt aber $x' = \tilde{x}$ und bei der Auswertung von a_2 kann kein ungünstigerer GDV–Verweis entstehen.

$\overset{1,2}{\Longrightarrow}$ Ist PK2 erfüllt, ist $MaxK' \not> MaxK$ ausgeschlossen. Sind PK1 *und* PK2 erfüllt, d.h. gilt zusätzlich $x' < x$, so ist ein möglicher GDV–Verweis auf dem linken Ast von f_s^p definitiv günstiger als auf dem linken Ast von f_s und die maximal erreichte Kellertiefe kann sich dadurch verringern, d.h. es gilt $MaxK' \leq MaxK$. □

Bemerkung 7.6 Ist *nur* PK2 erfüllt, so würde in der Situation gemäß Satz 7.1 gelten: $MaxK' = MaxK$. Erst durch das zusätzlich erfüllte Kriterium PK1 wird der für einen Parametertausch relevante Fall „$<$“ ermöglicht.

Aufgrund des Satzes 7.1 sind die beiden Kriterien PK1 und PK2 somit hinreichend, um bereits *vor* der eigentlichen Laufzeit über eine mögliche Parameter–Permutation entscheiden zu können. Eine statisch durchgeführter Parametertausch führt dann dynamisch sicher zu keiner Erhöhung des Speicherbedarfs, aber häufig zu einer Optimierung der GDV–Situation und somit zu einem Speicherplatzgewinn.

Wir werden im folgenden noch ein drittes Permutations–Kriterium einführen, welches jedoch auf den Satz 7.1 keine Auswirkungen hat:

7.2.1 Der CONSTANT–Algorithmus

Falls auf einem linken Ast nur ein konstanter S–Ausdruck steht, so findet sicher kein NSF–Aufruf auf diesem linken Ast statt, und eine mögliche Permutation aufgrund der Kriterien PK1 und PK2 würde zu keiner Verbesserung führen können. Falls jedoch durch die Permutation ein NSF–Aufruf auf dem linken Ast erzeugt würde, müßte zur Laufzeit die notwendige GMNIV–Markierung abgefragt werden, um letztendlich festzustellen, daß der auf den rechten Ast permutierte konstante S–Ausdruck keinen Einfluß auf das maximale statische Niveau im $\mathcal{RLK}$ des neuen linken Astes haben kann und sich die GDV–Situation nicht geändert hat.

Um diese angedeutete Situation einer *sicher unnötigen* Permutation zu vermeiden, führen wir folgendes dritte Kriterium mit Hilfe des unten erklärten Algorithmus CONSTANT ein:

Lemma 7.3 Sei $c = f_s(a_1, a_2)$ der Aufruf einer 2–stelligen SF f_s mit den Parametern a_1 und a_2. Falls gilt:

$$\text{CONSTANT}(a_1)=\text{false} , \qquad \textbf{(PK3)}$$

so sei dies ein *notwendiges* Kriterium für den Parametertausch bei c.

Bemerkung 7.7 In Anlehnung an Bemerkung 6.16 wollen wir den CONSTANT–Algorithmus in diesem Kapitel für die Bearbeitung von LISP–Programmen in Datensprache vorstellen. Dabei wird er hier zunächst in einer informellen Notation beschrieben, und im Abschnitt 7.4 ist er Bestandteil eines XLISP–Programms. Im Anhang B.2 ist dann der CONSTANT–Algorithmus im modifizierten Honschopp–Compiler als Pascal–Prozedur enthalten und operiert auf dem Zwischencode von echten LISP/N–Programmen.

Bemerkung 7.8 Der hier vorgeschlagene CONSTANT–Algoritmus stellt einen praktikabelen Kompromiß zwischen Zweck und Aufwand dar: Es werden lediglich die mehr oder weniger „offensichtlichen“ Ausdrücke erkannt, die sicher zu keinem NSF–Aufruf zur Laufzeit führen können. Zum Beispiel wird bei dem Ausdruck

```
(COND (T (QUOTE (A B)))
      (T (f x)))
```

nicht erkannt, daß er nur die Liste (A B) als Ergebnis liefern kann, und der NSF–Aufruf (f x) nie erreicht wird. (Nebenbei ist eine konstante Bedingung auch nur in der letzten Klausel sinnvoll!).

Der CONSTANT–Algoritmus:

Wird CONSTANT(ℓ) mit einem linken Ast ℓ als Eingabe aufgerufen, so wird ℓ gemäß der folgenden Regeln von links nach rechts nach der Methode des rekursiven Abstiegs untersucht. Wird vom Algoritmus festgestellt, daß für die Auswertung von ℓ *kein* NSF–Aufruf stattfinden kann (beachte Bemerkung 7.8!), so liefert CONSTANT den Wert „true“, ansonsten „false“ als Ergebnis zurück. Der Kürze halber schreiben wir den Aufruf CONSTANT(ℓ) auch als $\Gamma(\ell)$:

1. Ist ℓ ein Atom, d.h. ein Identifikator oder die leere Liste NIL, dann wird unterschieden:

   (a) $\ell \in$`{T,F,NIL}`: $\Rightarrow \Gamma(\ell)$`:=`true.

   (b) Sonst: $\Rightarrow \Gamma(\ell)$`:=`false.

2. Ist ℓ eine Liste und ℓ_0:=CAR(ℓ) ein Atom. Dann werden folgende Fälle unterschieden:

   (a) $\ell_0 \equiv$QUOTE: Das Argument CDR(ℓ) wird unausgewertet übergeben.
   $\Rightarrow \Gamma(\ell)$:=true.

   (b) $\ell_0 \in$\{CAR,CDR,ATOM\}: Das Argument wird ausgewertet.
   $\Rightarrow \Gamma(\ell)$:=Γ(CDR(ℓ)).

   (c) $\ell_0 \in$ \{CONS,EQ\}: Beide Argumente werden ausgewertet.
   $\Rightarrow \Gamma(\ell)$:=Γ(CADR(ℓ)) $\wedge$ Γ(CADDR(ℓ)).

   (d) $\ell_0 \equiv$COND: Rekursiver Abstieg in die Struktur des Konditionals (siehe Bemerkung 2.11).
   $\Rightarrow \Gamma(\ell)$:=($\Gamma(\mathtt{B}_1) \wedge \Gamma(\mathtt{F}_1)) \wedge \ldots \wedge (\Gamma(\mathtt{B}_n) \wedge \Gamma(\mathtt{F}_n)$)

   (e) Sonst: ℓ_0 ist Identifikator einer NSF oder eine NSF selbst.
   $\Rightarrow \Gamma(\ell)$:=false.

3. Ist ℓ eine Liste und ℓ_0:=CAR(ℓ) wieder eine Liste.
   $\Rightarrow \Gamma(\ell)$:=$\Gamma(\ell_0)$.

Es folgen einige kleine Beispiele, um die Wirkung von CONSTANT zu verdeutlichen:

Beispiel 7.5

```
CONSTANT( y )                          = false
CONSTANT( (QUOTE y) )                  = true
CONSTANT( (CAR y) )                    = false
CONSTANT( (g T) )                      = false
CONSTANT( (EQ NIL (CDR (QUOTE (A))) ) = true
CONSTANT( (COND (x F)
                (T T)) )               = false
CONSTANT( (COND ((ATOM NIL) T)
                (T          F)) )      = true
```

Im nächsten Abschnitt verdeutlichen wir die Relevanz unseres Parametertausches aufgrund der Permutationskriterien PK1 bis PK3:

7.3 Die Relevanz des Parametertausches

Aufgrund der Optimierung durch Parameterpermutation kann die Abarbeitung von mehrstelligen Standardfunktionen kostengünstiger werden. In LISP/N kommen dabei derzeit nur CONS und EQ in Frage. NSF–Aufrufe

auf den linken Ästen von CONS oder EQ, die trotz des in Abschnitt 6.1 eingeführten verbesserten GDV–Verweises durch einen GDV–Verweis nur relativ wenige AR's frühzeitig freigeben oder eine HOpt ganz verhindern, können durch den Tausch mit dem rechten Ast evtl. günstiger abgearbeitet werden.

Durch den Parametertausch kann sich somit die Häufigkeit und insbesondere die Güte der GDV–Optimierung aus Abschnitt 6.1 erhöhen.

Wir wollen nun die praktische Relevanz der im vorherigen Abschnitt eingeführten Parameter–Permutation bei mehrstelligen SF–Aufrufen dokumentieren:

Beispiel 7.1 (Fortsetzung): Bei dem einführenden Beispiel 7.1 sind die Kriterien PK1 bis PK3 für den Aufruf von CONS erfüllt. Es kann permutiert werden und wir erhalten für den Aufruf e(A) folgende maximale Kellertiefen:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	26
Definitonsgemäßer GDV	26
Einführung des GDV–Kellers	24
Dicke Parameter Opt. mittels GMARK	24
GDV–Verweis mittels GMARK	24
Parameter–Permutation mittels GPMARK	17

⇒ Es ergibt sich ein um ca. 34,6% geringerer Keller–Bedarf.

Beispiel 7.6 (vgl. Beispiel „perm1.lsp“ im Anhang A.1)

```
g = λ x . { f = λ y . { IF ATOM(x)
                          THEN x
                          ELSE CONS(CONS(g(CDR(x)),y),x)
                        FI }
            f(x) }
g((A B C D E F G H))
```

Für den inneren Aufruf von CONS sind die Kriterien PK1 bis PK3 erfüllt: Mit den gleichen Bezeichnungen wie in Lemma 7.1 und Lemma 7.2 gilt: $x' = 1$, $\tilde{x} = 1$ und $x = 2 \Rightarrow x' = \tilde{x} < x \Rightarrow$ PK1 *und* PK2 gelten. Ferner liefert CONSTANT(g(CDR(x))) den Wert true und somit ist auch PK3 erfüllt $\Longrightarrow$ beim inneren Aufruf von CONS wird ein Parametertausch durchgeführt.

Für den Aufruf g((A B C D E F G H)) ergeben sich dann folgende Werte:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	179
Definitonsgemäßer GDV	137
Einführung des GDV–Kellers	127
Dicke Parameter Opt. mittels GMARK	127
GDV–Verweis mittels GMARK	127
Parameter–Permutation mittels GPMARK	79

⇒ Es ergibt sich ein um ca. 55,9% geringerer Keller–Bedarf.

Beispiel 7.7 (vgl. Beispiel „aufg39.lsp“ im Anhang A.1)

```
p = λ x . { CONS( g(B) , c(x) ) }
g = λ x . { h(x) }
h = λ x . { CONS( k(D) , x ) }
c = λ x . { CONS( k(E) , x ) }
k = λ x . { x }
p(A)
```

Für *jeden* Aufruf der SF CONS im obigen Programm sind die Kriterien PK1 bis PK3 erfüllt, und es kann permutiert werden: Mit den gleichen Bezeichnungen wie in Lemma 7.1 und Lemma 7.2 gilt jedesmal: $x' = 0$, $\tilde{x} = 0$ und $x = 1 \Rightarrow x' = \tilde{x} < x \Rightarrow$ PK1 *und* PK2 gelten. Ferner liefert CONSTANT für jeden linken Ast den Wert true und somit ist auch PK3 erfüllt.

Für den Aufruf p(A) ergeben sich dann folgende maximale Kellertiefen in den jeweiligen Optimierungsstufen:

Optimierungsstufe	Maximale Kellertiefe
Keine Optimierung	26
Definitonsgemäßer GDV	26
Einführung des GDV–Kellers	25
Dicke Parameter Opt. mittels GMARK	25
GDV–Verweis mittels GMARK	25
Parameter–Permutation mittels GPMARK	11

⇒ Es ergibt sich ein um ca. 57,7% geringerer Keller–Bedarf.

Bemerkung 7.9 Bei einem aufgrund der erfüllten Kriterien PK1 bis PK3 durchgeführten Parametertauschs ist nicht sichergestellt, ob dynamisch optimiert wird. Dies ist aber auch nicht notwendig, denn dadurch, daß keine Verschlechterung eintreten kann (Satz 7.1), sondern oft konkret optimiert wird, haben diese Maßnahmen ihre Berechtigung. Das Kriterium PK3 schränkt den Kreis der unnötigen Permutationen lediglich ein.

Bemerkung 7.10 Gemäß Bemerkung 7.1 sind hier nur 2–stellige Standardfunktionen (CONS und EQ) für die mögliche Permutation untersucht worden. Die Grundzüge unserer Technik lassen sich aber auch auf n–stellige Standardfunktionen mit $n \geq 2$ übertragen: z.B. könnten die Kriterien PK1 bis PK3 paarweise auf die Argumente angewendet werden, bis kein Tausch mehr durchgeführt werden kann. Es reicht dann für Standardfunktionen, wie z.B. die in vielen Implementationen übliche Funktion LIST, nicht mehr aus, durch Einführung einer zusätzlichen Funktion PLIST die Parameter–Permutation semantisch zu kompensieren. Vielmehr müßte jedem Aufruf von LIST ein

„Permutations–Vektor“ mitgegeben werden, damit zur Laufzeit die andere Reihenfolge der Argumente beim Ermitteln des Resultates von LIST berücksichtigt werden könnte.

Bemerkung 7.11 Vorraussetzung für die Optimierung durch Parametertausch aufgrund des Satzes 7.1 ist die gleichzeitige Optimierung des GDV–Verweises gemäß Kapitel 6: Würde ein neuer GDV–Verweis weiterhin rigoros auf den dynamischen Vorgänger verweisen, so würde das Kriterium PK1 keinen Sinn ergeben — die GDV–Situation auf dem linken Ast bliebe die gleiche. Ferner würde das Kriterium PK2 nicht sicherstellen, daß für den zuvor rechten Ast durch einen Parametertausch nicht eine schlechtere Situation eintreten könnte (siehe z.B. Beispiel 7.1: der Aufruf g(z) hätte einen GDV–Verweis auf das $AR_{f(B)}$ und würde demnach höher im Keller beginnen).

Ein Parametertausch *ohne* die GDV–Optimierung könnte nur bei schärferen Kriterien Sinn ergeben: z.B. könnte mittels dem CONSTANT–Algorithmus untersucht werden, ob auf dem rechten Ast sicher *kein* NSF–Aufruf stattfindet. Ein Parametertausch hätte dann ggf. zur Folge, daß zuvor GDV–relevante NSF–Aufrufe vom linken auf den rechten Ast gelangen und auf dem linken Ast dann kein NSF–Aufruf mehr stattfindet.

Bemerkung 7.12 Die Optimierung durch Parameterpermutation beruht auf rein statischen Kriterien und kann daher bereits zur Compilationszeit durchgeführt werden. Das Laufzeitsystem wird lediglich um die neue Funktion PCONS erweitert und kann sonst wie gehabt die Programme ausführen.

Die Relevanz der Optimierung ist, was den Speicherplatzbedarf angeht, durch obige Beispiele dokumentiert. Laufzeit kann durch die bisher vorgestellten Optimierungen prinzipiell nicht eingespart werden, da vom Laufzeitsystem nicht weniger Aktionen durchgeführt werden müssen. Falls ein NSF–Aufruf erst aufgrund der Optimierung durch Parametertausch eine HOpt bewirkt, so fallen dann die (relativ geringen) Mehrkosten für die HOpt an. Falls ein Aufruf auch ohne Permutation eine HOpt bewirkt hat, so tritt durch einen Parametertausch *kein* Laufzeitverlust für die Abarbeitung dieses Aufrufs auf (siehe auch Bemerkung 6.15).

Bemerkung 7.13 Im Anhang A.1 befindet sich eine Sammlung weiterer Beispielprogramme zusammen mit der Dokumentation der erreichten Verbesserungen.

Bemerkung 7.14 Die Relevanz von Parameterpermutationen für den Static Scope Shallow–Binding Interpreter mit der LCC–Optimierungstechnik nach Felgentreu [Fe87] (siehe Kapitel 3) wird derzeit von J. Schumacher [Sch92] untersucht.

7.4 Der GPMARK Markierungsalgorithmus

Der GDV– und Permutations–Markierungsalgorithmus GPMARK ist im wesentlichen der GMARK–Algorithmus (siehe Abschnitt 6.5), erweitert um die Erkennung und Durchführung von Parameterpermutationen aufgrund der Permutationskriterien PK1 bis PK3 gemäß Abschnitt 7.2. Für die Kriterien PK1 und PK2 wird lediglich die bereits von GMARK berechenbare GMNIV–Information benötigt, und für die Bearbeitung von PK3 ist GMARK um den in Abschnitt 7.2.1 vorgestellten CONSTANT–Algorithmus zu erweitern.

Bemerkung 7.15 In Anlehnung an die Bemerkungen 6.16 und 7.7 wollen wir GPMARK an dieser Stelle ebenfalls in einer informellen Notation und für die Bearbeitung von LISP–Programmen in Datensprache beschreiben. Anschließend geben wir dann den GPMARK–Markierungsalgorithmus als lauffähiges XLISP–Programm an.

In Anhang B.2 ist der GPMARK–Algorithmus für die Bearbeitung des Zwischencodes von echten LISP/N–Programmen enthalten. Er ist dort Bestandteil des in Pascal formulierten Compilers.

7.4.1 Beschreibung von GPMARK

Zur Beschreibung von GPMARK ist lediglich die folgende Regel (7) der Beschreibung von GMARK in Abschnitt 6.5.1 hinzuzufügen:

7. Beim Erreichen eines SF–Aufrufs $c = f_s(a_1, a_2)$ mit $f_s \in \{CONS, EQ\}$ wird zunächst die Menge $\mathcal{M}$ gerettet. Bei den folgenden drei Berechnungen dürfen *keine* GMNIV oder DPNIV eingefügt werden (weil evtl. noch permutiert wird, und sich die Marken dann ändern können!), wobei die Variablen $\tilde{x}$, x und x' denjenigen in den Lemmata 7.1 und 7.2 entsprechen:

   - $\tilde{x}$:=GMNIV(a_2)
   - x :=GMNIV(a_1)
   - x' :=max(DPNIV(a_1),$\tilde{x}$),

     wobei zu bemerken ist, daß DPNIV an dieser Stelle nicht für einen dicken Parameter, sondern für den beliebigen aktuellen Parameter a_1 angewendet wird. Durch obige Maximumbildung erhalten wir dann die GMNIV–Information für den linken Ast *nach* einem Parametertausch *ohne* dafür konkret eine Permutation durchführen zu müssen.

   Falls nun gilt:

$$(x' < x) \ \wedge \ (x' = \tilde{x}) \ \wedge \ (CONSTANT(a_1) = false) \ ,$$

so sind die Kriterien PK1 bis PK3 erfüllt, und es wird a_1 mit a_2 vertauscht.

Der gerettete Wert von $\mathcal{M}$ (s.o.) wird zurückgeholt und mit der Bearbeitung des (evtl.) permutierten SF–Aufrufs fortgefahren.

7.4.2 Ein Beispiel

Wir verdeutlichen die Wirkungsweise des GPMARK–Markierungsalgorithmus an folgendem Beispiel:

Beispiel 7.8 Gegeben sei folgendes LISP–Programm (vgl. Beispiel „perm2.lsp“ im Anhang A.1):

```
(LABEL G
   (LAMBDA (X)
      ((LABEL H
          (LAMBDA (Y)
             (COND ((ATOM X) X)
                   (T (CONS (CONS (G (CDR X)) (G (CDR Y))) X)) )))
       X )))
```

Nachdem GPMARK das Programm markiert und permutiert hat, sieht es folgendermaßen aus:

```
(LABEL G
   (LAMBDA (X)
      ((LABEL H
          (LAMBDA (Y)
             (COND ((ATOM X (* 2 *) ) X)
                   (T (CONS (PCONS (G (* 1 *) (* 102 *) (CDR Y (* 0 *)))
                                   (G (* 1 *) (* 101 *) (CDR X (* 0 *))))
                            X)) )))
       X )))
```

Definition 7.3 Ein durch GPMARK markiertes und ggf. permutiertes Programm Π bezeichnen wir im folgenden mit Π^{PM}.

7.4.3 GPMARK als XLISP–Programm

Zum Abschluß wollen wir den oben beschriebenen GPMARK–Algorithmus noch in Form eines lauffähigen XLISP–Programms angeben. Dazu werden wir nur die Funktionen wiedergeben, die sich gegenüber dem GMARK–Programm in Abschnitt 6.5.3 geändert haben (GPMARK und MARK) bzw. neu hinzugekommen sind (CONSTANT und CONSCOND). Die Funktionen LISTMARK, CONDMARK, UNION, PARA, MAX und NIVEAU sind identisch mit denen in Abschnitt 6.5.3.

Der Aufruf von GPMARK erfolgt mit `(GPMARK 'Π)`, wobei Π das zu markierende (gequotete) LISP–Programm in Datensprache ist:

```
(defun gpmark (exp) (mark exp nil 0 nil exp t 0))

(defun mark (exp vars niv alist oldexp ins lac)
  (cond
    ((atom exp)
      (cond ((member exp '(T F NIL)) vars)
            ((and ins (> lac 0))     (prog1
                                       (union exp vars)
                                       (rplacd
                                         oldexp
                                         (cons (list '* (max vars alist 0) '*)
                                               (cdr oldexp)))))
            (t                        (union exp vars))))
    ((atom (car exp))
      (cond
       ((eq (car exp) 'QUOTE)  vars)
       ((eq (car exp) 'COND)   (condmark (cdr exp) vars niv alist ins lac))
       ((eq (car exp) 'LABEL)  (mark (caddr exp) nil niv
                                     (cons (cons (cadr exp) niv) alist)
                                     (cddr exp) ins lac))
       ((eq (car exp) 'LAMBDA) (union 'LAMBDA
                                      (mark
                                        (caddr exp) nil (1+ niv)
                                        (append (para (cons 'LAMBDA (cadr exp))
                                                      (1+ niv)) alist)
                                        (cddr exp) ins lac)))
       ((member (car exp) '(CAR CDR ATOM))
          (listmark (cdr exp) vars niv alist ins nil nil lac))
       ((member (car exp) '(CONS EQ))
          (prog2
            (cond
              ((and
                ins (not (constant (cadr exp)))
                (< (max (mark (cadr exp) vars niv alist nil nil lac) alist 0)
                   (max (mark (caddr exp) vars niv alist nil nil lac) alist 0))
                (= (max (mark (cadr exp) vars niv alist nil nil lac) alist 0)
                   (max vars alist 0)))
               (rplacd (cond ((eq (car exp) 'CONS) (rplaca exp 'PCONS)))
                       (cons (caddr exp) (list (cadr exp))))))
            (mark (cadr exp)
                  (listmark (cddr exp) vars niv alist ins nil nil lac)
                  niv alist (cdr exp) ins (1+ lac))))
       (t (prog1 (union (car exp)
                        (listmark (cdr exp) vars niv alist ins t nil lac))
                 (cond ((and ins (> lac 0))
                          (rplacd exp (cons (list '* (max vars alist 0) '*)
                                            (cdr exp)))))))))
    (t (prog1 (mark (car exp) (listmark (cdr exp) vars niv alist ins t nil lac)
                    niv alist exp ins lac)
              (cond ((and ins (> lac 0))
                       (rplacd exp (cons (list '* (max vars alist 0) '*)
                                         (cdr exp))))))))

(defun constant (l)
```

```
  (cond
    ((atom l) (cond ((member l '(T F NIL)) t)
                    (t                     nil)))
    ((atom (car l))
      (cond
        ((eq (car l) 'QUOTE)              t)
        ((eq (car l) 'COND)               (constcond (cdr l)))
        ((member (car l) '(CAR CDR ATOM)) (constant (cadr l)))
        ((member (car l) '(CONS EQ))      (and (constant (cadr l))
                                               (constant (caddr l))))
        (t                                nil)))
    (t        (constant (car l)))))

(defun constcond (l)
  (cond ((null l)                  t)
        ((and (constant (caar l))
              (constant (cadar l))
              (constcond (cdr l))) t)
        (t                         nil)))
```

Im nächsten Kapitel werden wir sehen, wie sich die Effizienz des Laufzeitsystems weiter steigern läßt und dadurch auch die notwendigen Maßnahmen für eine HOpt reduziert werden können.

Kapitel 8

Ein effizienter Laufzeitkeller

8.1 Die ineffiziente Handhabung der pending Parameter

Aufgrund des funktionalen Konzeptes von LISP/N können zur Laufzeit Aufrufe mit „pending Parametern“, d.h. zusätzlichen Argumentgruppen, auftreten. Zur Auswertung von funktionalen Ergebnissen oder Funktionsausdrücken muß die evtl. fehlende Parameterliste dann gemäß der Kopierregel in das zugehörige Activation Record kopiert werden, um einen vollständigen Aufruf mit aktuellen Parametern zu erhalten. Die durch die Kopierregel vorgegebene Semantik wird aber in Honschopp's Laufzeitsystem [Ho83] nicht effizient durchgeführt (siehe auch [Hu92]).

Beipielsweise wird für den Aufruf

$$c \;=\; f\ (a_1, \ldots, a_n)\ (b_1, \ldots, b_m)\ (c_1, \ldots, c_p)$$

der NSF f mit der aktuellen Parameterliste $(a_1, \ldots, a_n)$ und den beiden pending Parameterlisten $(b_1, \ldots, b_m)$ und $(c_1, \ldots, c_p)$ ein AR_f gemäß der Abbildung 2.2 auf Seite 28 angelegt. Falls in dem unmittelbar zuvor angelegten AR, d.h. dem dynamischen Vorgänger von c, noch pending Parameterlisten eingetragen sind, so werden diese, jeweils getrennt durch eine Trennmarke, im Anschluß an c_p ebenfalls in das AR_f kopiert. Die Auswertung von $f(a_1, \ldots, a_n)$ muß als Ergebnis einen Funktionsausdruck, d.h. beispielsweise den unvollständigen Funktionsaufruf g ohne aktuelle Parameterliste, liefern. In das für den Aufruf g anzulegende AR_g wird dann die erste pending Parameterliste $(b_1, \ldots, b_m)$ als aktuelle Parameterliste eingetragen. Die verbliebenen pending Parameterliste(n) werden wieder als pending Parameterliste(n) an das AR_g angehängt.

Wir sehen also, daß pending Parameter solange in jedes neu anzulegende AR weitergereicht werden müssen, bis sie als aktuelle Parameter benötigt werden. Die dabei entstehenden Kosten hängen natürlich stark von der Anzahl

der weiterzureichenden Parameter ab. Um diese Situation zu verdeutlichen, betrachten wir folgendes Beispiel:

Beispiel 8.1 Gegeben seien

- die kelleraufwendige rekursive Fibonacci–Funktion, die jedoch aufgrund des minimalen Sprachumfangs von Pure–LISP nicht auf Zahlen, sondern stattdessen auf der Listenlänge operiert,
- eine 10–stellige NSF g, sowie
- die NSF f, welche als Ergebnis immer den Funktionsausdruck g liefert und daher zur Laufzeit mit einer aktuellen Parameterliste versorgt werden muß:

```
fib = λ x . { siehe Beispiel 5.3 auf Seite 65 }
```

```
g = λ x1 ... x10 . { CONS(x1,CONS(x2,...,CONS(x9,x10)...)) }
```

```
f = λ x . { IF ATOM(fib(x)) THEN g
                            ELSE g
            FI }
```

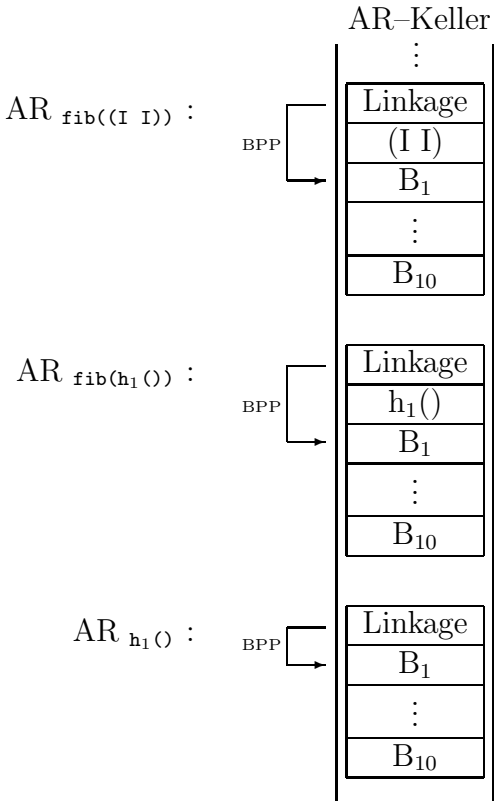
Für den Aufruf f((I I))($B_1,\ldots,B_{10}$) entsteht nach drei weiteren Aufrufen der in Abbildung 8.1 dargestellte Zustand im AR–Keller, wobei h_1=λ.{CDR(x)} die im Rumpf von *fib* deklarierend vorkommende Nichtstandard–Hilfsfunktion h_1 für den dicken Parameter „CDR(x)“ im linken Ast von CONS sei (vgl. Abschnitt 2.2). Die Darstellung basiert auf der Abarbeitung des Programms *mit GDV–Optimierung* (siehe Kapitel 6): Durch den NSF–Aufruf fib((I I)) auf dem linken Ast in dem Anweisungsteil von f konnte durch den neuen GDV–Verweis das $AR_{f((I\ I))(B_1,\ldots,B_{10})}$ frühzeitig freigegeben werden und steht nicht mehr im AR–Keller.

Wir sehen also, daß nach dem 4. NSF–Aufruf 3 Kopien der umfangreichen pending Parameter ($B_1,\ldots,B_{10}$) gleichzeitig im AR–Keller gehalten werden müssen. Erst beim 10. und somit letzten Aufruf werden die pending Parameter zu aktuellen Parametern, und das Beispiel wird mit der Ausgabe des Ergebnisses beendet.

Bemerkung 8.1 Zusammenfassend können wir die ineffiziente Realisierung der Übergabe von pending Parametern gemäß der vorgegebenen Kopierregel–Semantik wie folgt charakterisieren:

1. Es sind oft viele Kopien der gleichen pending Parameter *gleichzeitig* im AR–Laufzeitkeller. Durch diese redundanten Informationen wird Speicherplatz verschwendet.

Abbildung 8.1: AR–Keller zu Beispiel 8.1 nach dem 4. NSF–Aufruf



2. Der Eintrag „BPP“ in jeder Linkage eines AR ist nötig, um den Anfang der pending Parameter schnell zu ermitteln.

3. Das Kopieren der pending Parameter vom dynamischen Vorgänger kostet Zeit.

4. Jedes AR wird zunächst provisorisch angelegt, um die von ihm ausgehenden Verweise in den Laufzeitkeller zu überprüfen. Dabei werden die vom dynamischen Vorgänger übernommen pending Parameter jedesmal neu überprüft.

5. Kann aufgrund der festgestellten Verweise eine Speicherplatz–Optimierung (HOpt) durchgeführt werden, so muß das zunächst provisorisch angelegte AR noch einmal komplett verschoben werden, wodurch die vom dynamischen Vorgänger übernommenen pending Parameter ein zweitesmal kopiert werden.

8.2 Die Einführung des pending Parameter–Kellers

Damit pending Parameter nicht von AR zu AR weitergereicht werden müssen, bis sie irgendwann benötigt werden, und weil die zuletzt aufgerufe-

nen pending Parameter immer zuerst wieder als aktuelle Parameter benutzt werden müssen, bietet sich an, für sie einen eigenen Kellerspeicher (FIFO–Prinzip) zu reservieren: den *pending Parameter–Keller* (kurz: *PP–Keller*).

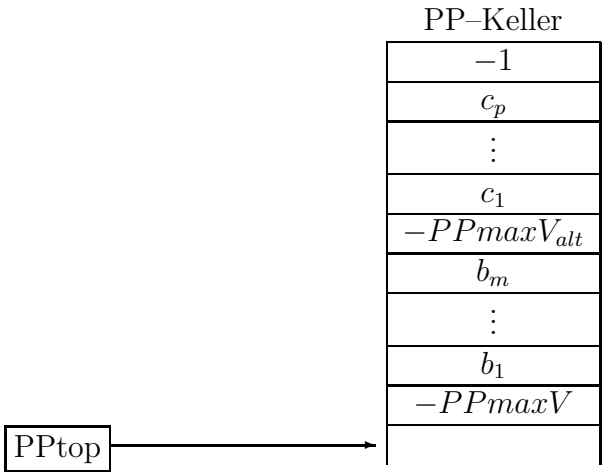
Fallen zur Laufzeit bei einem NSF–Aufruf pending Parameter an, d.h. werden nach einer aktuellen Parameterliste noch (evtl. mehrere) pending Parameterlisten an die Linkage des anzulegenden AR angehängt, so werden diese pending Parameter in den PP–Keller verschoben und dabei gleichzeitig die von diesen Parametern ausgehenden maximalen statischen Verweise in den AR–Keller bestimmt. Es bleiben somit keine AR's mit pending Parametern mehr im AR–Keller stehen.

Anhand des Aufrufs

$$c \;=\; f\ (a_1, \ldots, a_n)\ (b_1, \ldots, b_m)\ (c_1, \ldots, c_p)$$

wollen wir die Organisation des neuen PP–Kellers in Abbildung 8.2 darstellen:

Abbildung 8.2: Die Organisation des pending Parameter–Kellers



Der abgebildete PP–Keller wächst von oben nach unten, und es wurden vor dem Aufruf c noch keine pending Parameter im PP–Keller abgelegt.

Der globale Zeiger PPtop zeigt immer auf den ersten freien Platz im PP–Keller, und der oberste Eintrag im PP–Keller gibt immer den aktuellen maximalen statischen Verweis aus dem PP–Keller in den AR–Keller an (PPmaxV). Der Eintrag PPmaxV wird in negativer Form in den PP–Keller geschrieben, um gleichzeitig als notwendige und eindeutige Trennmarke zwischen den einzelnen pending Parameterlisten zu dienen.

Zum Programmstart wird der PP–Keller mit „−1“ als ersten Eintrag initialisiert, um bei einer möglichen Abfrage nach dem aktuellen PPmaxV einen Verweis auf das erste AR im AR–Keller zu liefern.

Vor jedem Verschieben einer pending Parameterliste in den PP–Keller wird das Maximum aus dem obersten Kellereintrag PPmaxV und und dem maximalen statischen Verweis aus den neuen, in den PP–Keller zu verschiebenden pending Parametern ermittelt. Nach dem Verschieben wird dieser neue Wert für PPmaxV in negativer Form in den PP–Keller gepusht.

Wird die oberste pending Parameterliste für einen NSF–Aufruf als aktuelle Parameterliste benötigt, so wird die oberste pending Parameterliste bis zur nächsten Trennmarke (negativer Eintrag) in das anzulegende AR verschoben. Die dann als oberster Eintrag zurückbleibende Trennmarke gibt nun wieder den gültigen maximalen statischen Verweis aus dem PP–Keller an.

Bemerkung 8.2 Analog zu der Bemerkung 8.1 wollen wir die effizientere Handhabung von pending Parametern durch den PP–Keller aufführen:

zu 1. Da pending Parameter nicht mehr weitergereicht werden, sondern solange im PP–Keller stehen bleiben, bis sie benötigt werden, existieren die jeweiligen pending Parameter immer nur *einmal* im Laufzeitspeicher.

zu 2. Der Eintrag „BPP“ aus jeder Linkage wird ersetzt durch den einen globalen Zeiger „PPtop“. Gegenüber der durch den eingeführten GDV–Keller schon um den Eintrag GDV verkürzten Linkage reduziert sich jede Linkage nun auf nur noch 3 Einträge (siehe Abbildung 8.3).

zu 3. Das Kopieren der pending Parameter vom dynamischen Vorgänger entfällt vollständig — sie stehen bereits im PP–Keller.

zu 4. Der oberste Eintrag im PP–Keller (PPmaxV) gibt immer den aktuellen maximalen statischen Verweis aus dem PP–Keller an. Nur wenn neue pending Parameter in den PP–Keller abgelegt werden sollen, wird der neue oberste Eintrag von PPmaxV wie folgt ermittelt:

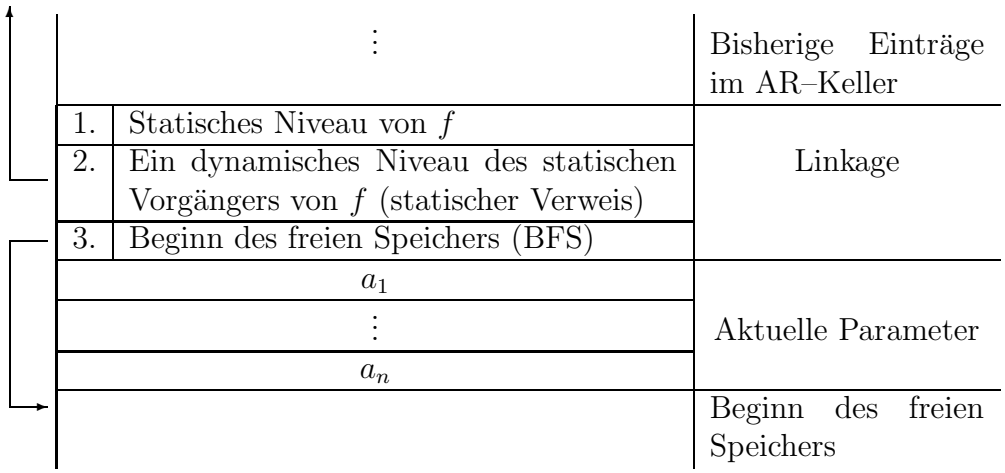
PPmaxV := max (stat. Verw. aus neuen PP's , PPmaxV).

Damit braucht jeder pending Parameter nur noch einmal auf seinen statischen Verweis hin untersucht zu werden. Für den Fall, daß pending Parameter zu aktuellen Parametern werden, kann der maximale Verweis aus allen Parametern direkt dem Wert von PPmaxV entnommen werden.

zu 5. Die Kosten für eine HOpt werden günstiger, da evtl. vorhandene pending Parameter nicht mehr mit verschoben werden müssen. (Im nächsten Abschnitt werden wir die Laufzeit–Kosten für eine HOpt noch weiter reduzieren.)

Wie erwähnt, reduziert sich die Linkage auf nur noch drei Einträge. Ebenfalls kommen keine Trennmarken mehr in einem Activation Record vor. Sie dienten nur zum Trennen von pending Parameterlisten und stehen nun im PP–Keller mit der zusätzlichen Information PPmaxV. Die Endemarke ist redundant zum Eintrag „BFS“ in der Linkage und kann ebenfalls fortgelassen werden. In Abbildung 8.3 ist der nun minimale allgemeine AR–Aufbau dargestellt.

Abbildung 8.3: Ein minimales Activation Record



8.3 Eine kostengünstigere HOpt

Wie in Kapitel 2 beschrieben, wird für jeden NSF–Aufruf das im AR–Laufzeitkeller abzuspeichernde AR zunächst provisorisch angelegt. Dabei wird es an der Spitze des AR–Kellers aufgebaut und kann dort stehenbleiben, falls nach der Überprüfung aller statischen Verweise *keine* HOpt durchgeführt werden kann.

Ist jedoch eine HOpt möglich, so wird das AR komplett an die neue Stelle im AR–Keller kopiert und anschließend sowohl der Verweis BFS in der Linkage als auch das zugehörige Indexregister nochmals aktuallisiert.

Sei n die Stelligkeit der aktuellen Parameterliste des zunächst provisorisch angelegten AR. Da eine Linkage 3 Einträge ausmacht (siehe Abbildung 8.3), sind im Falle einer HOpt noch $4 + n$ zusätzliche Schreib–Operationen notwendig. Da die n aktuellen Parameter vom Laufzeitsystem zunächst ermittelt werden müssen und dabei im allgemeinen statisch nicht entscheidbar ist, wie groß der maximale statische Verweis aus den aktuellen Parametern zur Laufzeit sein wird (z.B. bei formalen Identifikatoren), müssen die n Parameter zwischengespeichert werden. Dabei bietet sich die AR–Kellerspitze ab der 4.

freien Position an, da im Falle *keiner* HOpt nur noch die Linkage in den 3 freien Zellen darüber eingetragen werden muß und das AR dann schon an der richtigen Stelle steht.

Um zu einer kostengünstigereren HOpt zu gelangen, wird nun die Linkage und das entsprechende Indexregister erst dann beschrieben, wenn die endgültige Plazierung des AR im Keller feststeht, d.h. über eine mögliche HOpt bereits entschieden worden ist (die nötigen Informationen dazu liegen dem Laufzeitsystem bereits vor!). Der Aufwand für eine HOpt beschränkt sich damit auf das Verschieben der n Parameter der aktuellen Parameterliste!

Bemerkung 8.3 Falls bei einem Aufruf pending Parameter zu aktuellen Parametern werden müssen, wird die oberste pending Parameterliste aus dem neuen PP–Keller erst dann an das anzulegende AR angehängt, wenn die Position des AR im AR–Keller bereits feststeht. Nur in solchen Fällen besteht somit *kein* zeitlicher Mehraufwand für das Verschieben eines AR im Rahmen einer HOpt!

8.4 Die Relevanz der Optimierungen

Durch die in Abschnitt 8.1 eingeführte optimierte Handhabung von pending Parametern durch das Laufzeitsystem kann sowohl Speicherplatz als auch Laufzeit eingespart werden.

Durch die Optimierung in Abschnitt 8.3 bleibt der Speicherplatzbedarf einer Programmausführung derselbe: Da die Linkage eines zunächst provisorisch angelegten AR erst dann geschrieben wird, wenn die Position des AR im AR–Keller feststeht, kann lediglich Laufzeit eingespart werden.

Dadurch, daß nur noch die aktuelle Parameterliste eines NSF–Aufrufs (falls überhaupt vorhanden!) im Rahmen einer frühzeitigen Speicherplatzfreigabe verschoben werden muß, wird eine HOpt kostengünstiger.

In den Kapiteln 5 bis 7 haben wir Optimierungen vorgestellt, durch die jetzt viele NSF–Aufrufe, die bisher nicht in der HOpt–Klasse enthalten waren, eine frühzeitige Speicherplatzfreigabe durchführen. Sobald aber eine HOpt durchgeführt werden soll, fällt als zusätzliche Maßnahme das Verschieben eines AR an eine tiefere Stelle im AR–Keller an. Die dabei auftretenden Laufzeiteinbußen sind im Vergleich zu den möglichen Speicherplatzeinsparungen sehr gering (vgl. Bemerkungen 5.5, 6.15 und 7.12).

Durch die in diesem Kapitel minimierte Linkage (nur noch 3 Zellen!) und durch den Wegfall der unnötigen Endemarke werden die Aktionen des Laufzeitsystems für den Umgang mit AR's reduziert. Dabei hat sich gezeigt, daß die zuvor angesprochenen möglichen Laufzeiteinbußen aufgrund der Speicherplatzoptimierungen durch den neuen effizienteren Laufzeitkeller im allgemeinen mehr als wettgemacht werden (vgl. Tabellen im Anhang A.1)!

Fallen zur Laufzeit jedoch pending Parameterlisten an, so können durch die Vermeidung redundanter pending Parameter im AR–Keller, durch effektivere Bestimmung des maximalen Verweises für provisorisch angelegte AR's und durch weniger AR–Zellen, die im Rahmen einer HOpt verschoben werden müssen, enorme Speicherplatz– und Laufzeitgewinne mit den in Abschnitt 8.2 eingeführten PP–Keller erziehlt werden.

Bevor wir die Einsparungen an Beispiel 8.1 dokumentieren, folgende

Bemerkung 8.4 Entsprechend Bemerkung 6.12 werden von nun an für diejenigen Optimierungsstufen, die den PP–Keller nutzen, bei der Angabe der *maximalen Kellertiefe* auch die Einträge im PP–Keller berücksichtigt.

Beispiel 8.1 (Fortsetzung):
Für den Aufruf von `f((I I I I I I I I I I I I))`$(B_1,\ldots,B_{10})$ erhalten wir folgende Werte:

Optimierungsstufe	Maximale Kellertiefe	Laufzeit
Keine Optimierung	449	173,8s
Definitonsgemäßer GDV	273	184,2s
Einführung des GDV–Kellers	272	180,9s
Dicke Parameter Opt. mittels GMARK	272	180,9s
GDV–Verweis mittels GMARK	254	185,9s
Parameter–Permutation mittels GPMARK	—	—
Effizienter Laufzeit–Keller	84	56,3s

$\Rightarrow$ ca. 81,3% Kellerspeicher– und etwa 67,6% Zeitersparnis.

Dadurch, daß die pending Parameterliste in diesem Beispiel (relativ) umfangreich ist, verursachen die redundanten Informationen im AR–Keller, das Kopieren von pending Parametern vom dynamischen Vorgänger, sowie die Durchführung einer HOpt erhebliche Speicherplatz– und Laufzeitkosten *ohne* den effizienten Laufzeitkeller.

Für den Aufruf fib((I I I I I I I I I I I I)) wird die Fibonacci–Rekursion *ohne* eine gleichzeitig im Speicher zu haltende pending Parameterliste durchgeführt. Hier wird dann auch deutlich, daß die zwischenzeitlichen Verlängerungen der Laufzeit durch die vorherigen Optimierungsstufen durch den effizienteren Laufzeitkeller mehr als ausgeglichen werden:

Optimierungsstufe	Maximale Kellertiefe	Laufzeit
Keine Optimierung	168	57,2s
Definitonsgemäßer GDV	102	60,0s
Einführung des GDV–Kellers	101	57,8s
Dicke Parameter Opt. mittels GMARK	101	57,8s
GDV–Verweis mittels GMARK	101	62,8s
Parameter–Permutation mittels GPMARK	—	—
Effizienter Laufzeit–Keller	73	56,3s

$\Rightarrow$ ca. 56,5% Kellerspeicher– und ca. 1,6% Zeitersparnis.

Vergleichen wir die Zeiten mit dem Aufruf zuvor, so fällt auf, daß auch der Mehraufwand für die pending Parameterliste nahezu verschwunden ist!

Wir haben gesehen, daß durch relativ einfache Maßnahmen am Laufzeitsystem sowohl Speicherplatz als auch Laufzeit eingespart werden kann. Insbesondere werden die Kosten für eine HOpt reduziert, und der mögliche zeitliche Mehraufwand durch die vorhergehenden Optimierungen mehr als ausgeglichen.

Im nächsten Kapitel werden wir für viele Aufruf–Situationen die Auswertungsstrategie „Call By Need“ einführen und dadurch, im Zusammenspiel mit den bisher vorgestellten Optimierungstechniken, zu einem sehr leistungsfähigen Laufzeitsystem für LISP/N kommen.

Kapitel 9

Die Auswertungsstrategie Call By Need

In gewöhnlichen imperativen Sprachen (wie z.B. Pascal, Modula, C, ...) werden die Argumente (aktuelle Parameter) eines NSF–Aufrufs bereits *ausgewertet* der gerufenen Funktion übergeben (Call By Value). Diese Vorgehensweise impliziert jedoch zum einen, daß auch diejenigen Argumente ausgewertet werden, die im Anweisungsteil der gerufenen Funktion überhaupt nicht benötigt werden, und zum andern, daß die Berechnung eines Funktionsaufrufs nicht terminiert, wenn die Auswertung eines Argumentes es nicht tut.

Für Programmiersprachen, die auf der Semantik des λ–Kalküls basieren (z.B. LISP), ist daher die Auswertungsstrategie Call By Name vorzuziehen. Hierbei werden die Argumente eines NSF–Aufrufs *unausgewertet*, also textuell (namentlich), der gerufenen Funktion übergeben und erst dann ausgewertet, wenn sie benötigt werden („Lazy Evaluation"), d.h. eventuell auch gar nicht. Der entscheidende Nachteil von Call By Name ist jedoch, daß die Argumente *jedesmal* dann, wenn sie benötigt werden, ausgewertet werden. Eine gegebenenfalls *mehrfache* Auswertung der Argumente ist möglich, und daher ist Call By Name in der Regel wesentlich ineffizienter als Call By Value!

Die Auswertungsstrategie Call By Need dagegen vereinigt die Vorzüge und insbesondere die Semantik von Call By Name mit der Effizienz von Call By Value: Bei einem NSF–Aufruf werden die Argumente unausgewertet übergeben. Wird ein Argument im Anweisungsteil der gerufenen Funktion jedoch erstmals benötigt, so wird es wie gehabt ausgewertet und das Ergebnis dieser Auswertung dann für jede weitere versuchte Auswertung des gleichen Arguments herangezogen. Auf diese Weise kann ein Argument eines NSF–Aufrufs wenn überhaupt, dann nur noch *einmal* ausgewertet werden.

Wir veranschaulichen die Grundzüge der drei Auswertungsstrategien Call By Value, Call By Name und Call By Need an folgendem einfachen Beispiel:

Beispiel 9.1 Gegeben sei folgendes Programm

```
g = λ y . { B }
f = λ x . { CONS ( x , x ) }
f(g(A))
```

mit dem Aufruf f(g(A)). Im Anweisungsteil von f kommt der formale Identifikator x zweimal vor, d.h. zur Laufzeit wird sowohl für die Auswertung des linken als auch des rechten Astes von CONS auf den Inhalt der Parameterzelle zum formalen Parameter x zugegriffen.

Der grobe Ablauf der Programmausführung für die drei Strategien sieht dann wie folgt aus:

Call By Value: $\texttt{f(g(A))} \longrightarrow \texttt{g(A)} \longrightarrow \texttt{f(B)} \overset{x=B}{\longrightarrow} \texttt{CONS(B,B)}$

Der aktuelle Parameter g(A) wird *ausgewertet* der NSF f übergeben, und damit findet im Anweisungsteil von f kein NSF–Aufruf mehr statt.

Call By Name: $\texttt{f(g(A))} \overset{x=g(A)}{\longrightarrow} \texttt{CONS(g(A),g(A))} \longrightarrow \texttt{g(A)} \longrightarrow$

$\texttt{CONS(B,g(A))} \longrightarrow \texttt{g(A)} \longrightarrow \texttt{CONS(B,B)}$

Der aktuelle Parameter g(A) wird unausgewertet (textuell) der NSF f übergeben, und sowohl für die Auswertung des linken Astes als auch für die des rechten Astes von CONS muß der NSF–Aufruf g(A) ausgeführt werden.

Call By Need: $\texttt{f(g(A))} \overset{x=g(A)}{\longrightarrow} \texttt{CONS(g(A),g(A))} \longrightarrow \texttt{g(A)} \overset{x=B}{\longrightarrow} \texttt{CONS(B,B)}$

Wieder wird der aktuelle Parameter g(A) unausgewertet der NSF f übergeben, aber nur einmal im Anweisungsteil von f ausgeführt: Nach dem ersten Zugriff auf x und der daraus resultierenden Ausführung von g(A) (hier im linken Ast von CONS) ist das Ergebnis des Aufrufs g(A) für jeden weiteren versuchten Zugriff auf x bereits bekannt. Für die Auswertung des rechten Astes von CONS braucht dadurch kein NSF–Aufruf mehr ausgeführt zu werden.

Wir werden in diesem Kapitel eine Technik beschreiben, wie für viele Aufruf–Situationen die Auswertungsstrategie Call By Name in dem kellerartigen Laufzeitsystem nach Honschopp [Ho83] für die funktionale Sprache LISP/N (das „N“ steht für Call By Name) durch die semantisch äquivalente, jedoch wesentlich effizientere Strategie Call By Need ersetzt werden kann. Dabei können alle zuvor vorgestellten Optimierungen in vollem Umfang beibehalten werden, und es entsteht dann ein insgesamt sehr leistungsfähiges Laufzeitsystem.

9.1 Zur Auswertung formaler Identifikatoren

Die Auswertung eines formalen Identifikators x findet immer dann statt, wenn ein Vorkommen von x zur Laufzeit erreicht wird und x *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs vorkommt.

Wird der Wert von x zur Laufzeit benötigt, so greift das Laufzeitsystem durch

$$\text{AR–Keller [IR[SN}(x)] + \text{Rel.Adr.}(x) \text{]}$$

auf die zum formalen Parameter x zugehörige Parameterzelle innerhalb eines AR zu, wobei mittels der Indexregister (IR) das betreffende AR, und durch die Relativ–Adresse (Rel.Adr) von x die Position innerhalb des AR, bestimmt wird.

Bei der betrachteten Implemetierung in [Ho83] werden gemäß Tabelle 2.1 je Parameter drei Informationen (Typ, Adresse und statischer Verweis) in jede Parameterzelle eines AR eingetragen. Entsprechend dem Typ des Parameters x werden die in Tabelle 9.1 auf Seite 130 aufgeführten Aktionen zur Bestimmung des Wertes von x durchgeführt.

Wir sehen also, daß nur die Auswertung eines formalen Identifikators x, dessen aktueller Wert ein dicker Ausdruck, d.h. vom Typ–2 ist, mindestens einen NSF–Aufruf nach sich zieht und somit aufwendig ist. Aufgrund der Call By Name Semantik von LISP/N können dicke Parameter frühstens im Anweisungsteil der gerufenen NSF ausgeführt werden, und zwar jedesmal aufs neue, wenn der Wert des zugehörigen formalen Parameters benötigt wird. Wir verdeutlichen diese Ineffizienz an folgenden fünf Beispielen. Zunächst die Fibonacci–Rekursion aus Kapitel 5, in der die mehrfache Auswertung eines formalen Parameters innerhalb desselben Anweisungsteils auftritt:

Beispiel 5.3 (Fortsetzung): Bei einem Aufruf der NSF *fib* wird der formale Identifikator x zunächst im ersten linken Ast „ATOM(x)“ ausgewertet und anschließend mindestens noch einmal, d.h im ungünstigsten Fall aber noch dreimal ausgewertet werden müssen. Werden alle else–Teile durchlaufen, muß x somit viermal ausgewertet werden! In diesem Beispiel ist dies besonders unökonomisch, da x die rekursiven Aufrufe von fib enthalten kann und das Ergebnis der Fibonacci–Funktion bekanntlich die Anzahl der nötigen Aufrufe wiedergibt — bei Call By Name also wesentlich mehr!

Tabelle 9.1: Wertbestimmung eines formalen Identifikators x

Typ von x	Aktionen zur Wertbestimmung von x
0	Der Wert von x ist ein (konstanter) S–Ausdruck, d.h. ein Atom oder eine Liste und kann durch einen einfachen Zugriff auf den Heap an der „Heap–Adresse“ (siehe Tabelle 2.1) ermittelt werden. Insbesondere werden zur Wertbestimmung von x *keine* NSF–Aufrufe durchgeführt.
1	Der Wert von x besteht aus der Startadresse einer SF oder NSF f sowie aus „einem dynamischen Niveau des statischen Vorgängers“ zu f. Insbesondere brauchen somit auch hier *keine* NSF–Aufrufe zur Wertbestimmung von x durchgeführt werden. Der Wert von x liegt also bereits vor, d.h. die aufzurufende Funktion f ist bekannt und es muß (um einen vollständigen Aufruf von f zu erhalten) eine Parameterliste vom Laufzeitsystem bereitgestellt werden...
2	Zur Wertbestimmung von x muß gemäß der Handhabung dicker Parameter (siehe Abschnitt 2.2) eine parameterlose Nichtstandard–Hilfsfunktion h ausgeführt werden. Der Wert von x liegt bei diesem Parameter–Typ–2 somit nicht unmittelbar vor, sondern muß erst durch den NSF–Aufruf h() und infolgedessen noch evtl. vielen weiteren NSF–Aufrufen bestimmt werden. Die Auswertung eines Typ–2 Parameters ist damit *teuer*.

In dem folgenden einfachen Beispiel wird die mehrfache Auswertung desselben formalen Parameters x durch Vorkommen des zugehörigen formalen Identifikators x in unterschiedlichen Anweisungsteilen ausgelöst:

Beispiel 9.2

```
f = λ x . { g = λ . { x }
            CONS(g(),x) }
f(CONS(A,B))
```

Die notwendige Hilfsfunktion h zum dicken Parameter CONS(A,B) muß zunächst im Anweisungsteil von g und dann im rechten Ast des Anweisungsteils von f ausgewertet werden.

Im dritten und vierten Beispiel ergibt die (mehrfache) Auswertung des formalen Parameters x keinen S–Ausdruck, sondern einen (gewöhnlichen) Funktionsidentifikator f:

Beispiel 9.3

```
f = λ y . { y }
g = λ   . { f }
e = λ x . { CONS( x(A) , x(B) ) }
e(g())
```

Die Auswertung des dicken Parameters g() liefert als Ergebnis den Funktionsidentifikator der NSF f, und findet sowohl im linken als auch im rechten Ast von CONS statt.

Beispiel 9.4

```
f = λ y . { y }
g = λ   . { f }
e = λ x . { h = λ . { x }
            IF h()(T) THEN x
                      ELSE x
            FI }
e(g())(F)
```

Die Auswertung des dicken Parameters g() liefert als Ergebnis den Funktionsidentifikator der NSF f und findet in den Anweisungsteilen von h und e statt.

Ein formaler Identifikator wird als aktueller Parameter an eine andere NSF weitergereicht, dort ausgewertet, und nach der Rückkehr in den Anweisungsteil erneut ausgewertet:

Beispiel 9.5 (vgl. Beispiel „need2c.lsp“ im Anhang A.1)

```
f = λ x . { g = λ y . { e = λ z . { CONS(z,z) }
                        CONS(e(y),y) }
            CONS(g(x),x) }
f(CAR((A B)))
```

Die Auswertung des dicken Parameters CAR((A B)) erfolgt durch Auswertung der formalen Identifikatoren x und y in den jeweiligen rechten Ästen und durch Auswertung von z im rechten und linken Ast von CONS — die Hilfsfunktion zu CAR((A B)) wird also insgesamt viermal aufgerufen.

9.2 Die Einführung von Call By Need

Der Sinn von Call By Need ist es, wie der Name schon andeutet, die Auswertung eines aktuellen Parameters eines NSF–Aufrufs nur dann vorzunehmen, wenn der formale Identifikator des zugehörigen formalen Parameters (siehe Definition 1.25) im Anweisungsteil der gerufenen NSF benötigt wird. Im Unterschied zu Call By Name soll dies auch nur *einmal* erfolgen.

Aufgrund der in Abschnitt 1.2 vorgestellten Kopierregel–Semantik von LISP/N machen wir folgende Feststellung:

Korollar 9.1 Sei $c = g(\ldots)$ der Aufruf einer NSF g in einem echten LISP/N–Programm Π. Falls c zur Laufzeit ausgeführt wird und die Ausführung terminiert, so ist das Ergebins von c entweder

- ein S–Ausdruck s und steht im Accumulator (AC) als Verweis in den Heap oder
- der gewöhnliche Funktionsidentifikator einer SF oder NSF f. (In diesem Fall muß das Laufzeitsystem eine aktuelle Parameterliste bereitstellen, um zu einem vollständigen Aufruf der Funktion f zu gelangen und damit den nächsten Kopierregelschritt einleiten zu können.)

Das Ergebnis des Aufrufs von Π, d.h. dem Aufruf des Hauptprogramms, ist immer ein S–Ausdruck oder die leere Zeichenreihe („Void“), wobei Void ausschließlich als Ergebnis für das Hauptprogramm zugelassen ist.

Wir wir in Abschnitt 9.1 gesehen haben, ist lediglich die Auswertung eines Typ–2 Parameters teuer, da dann der Aufruf einer Nichtstandard–Hilfsfunktion h anfällt und ausgeführt werden muß. Entsprechend obigen Korollars 9.1 liefert auch dieser NSF–Aufruf entweder einen S–Ausdruck oder einen gewöhnlichen Funktionsidentifikator als Ergebnis zurück.

Die Idee, die nun der Call By Need–Realisierung zugrunde liegt, ist die Aktualisierung einer Typ–2 Parameterzelle eines dicken Parameters mit dem Ergebnis seiner Auswertung. Diese Aktualisierung soll nach Möglichkeit bereits nach der *ersten* Auswertung des dicken Parameters erfolgen, so daß alle weiteren Zugriffe auf die Parameterzelle keinen Typ–2 Parameter mehr vorfinden, sondern

- einen Typ–0 Parameter entsprechend dem Inhalt des AC, falls die Auswertung des dicken Parameters einen S–Ausdruck als Ergebnis liefert, oder
- einen Typ–1 Parameter mit der Startadresse und einem dynamischen Niveau des statischen Vorgängers von f, falls die Auswertung des dicken Parameters einen gewöhnlichen Funktionsidentifikator einer Funktion f als Ergebnis liefert.

Dazu folgende

Definition 9.1 Sei ρ das Ergebnis der Auswertung eines dicken Parameters. Die Aktualisierung der zugehörigen Typ–2 Parameterzelle entsprechend ρ bezeichnen wir als *Update*. Falls ρ als S–Ausdruck bzw. als gewöhnlicher Funktionsidentifikator bekannt ist, d.h. die Typ–2 Parameterzelle mit einer Typ–0 bzw. Typ–1 Parameterzelle überschrieben werden soll, so sprechen wir auch von einem *Typ–0 Update* bzw. einem *Typ–1 Update*.

Bemerkung 9.1 Wir werden in den Abschnitten 9.3 und 9.5 sehen, daß die hier vorgestellte Call By Need–Realisierung nicht optimal ist, d.h. in manchen Aufruf–Situationen weiterhin gemäß Call By Name verfahren wird. Jedoch werden sehr viele für die Praxis relevante Aufruf–Situationen gemäß Call By Need erfasst und die festgestellten Laufzeit– und Speicherplatz–Gewinne dokumentieren die Relevanz auch eines nicht ganz vollständigen Call By Need (Abschnitt 9.6).

Da LISP/N eine voll getypte Sprache ist, liegt der Ergebnis–Typ der Auswertung eines formalen Identifikators x in einem echten LISP/N–Programm Π bereits statisch, also *vor* der eigentlichen Laufzeit, fest (siehe auch Definition 1.27). Muß für die Auswertung von x ein dicker Parameter ausgeführt werden, so liegt durch den „Mode“ des zugehörigen formalen Parameters x auch der Ergebnis–Typ des dicker Parameter–Aufrufs statisch fest. Aufgrund Korollar 9.1 stellen wir fest:

Korollar 9.2 Sei (x, k) das Vorkommen eines formalen Identifikators x in einem echten LISP/N–Programm Π. Sei (x, k') das gemäß der Bindungsrelation δ_Π eindeutig zugeordnete deklarierende Vorkommen des formalen Parameters x zu (x, k) (vgl. Definition 1.25). Dann ist das Ergebnis (der Wert) der Auswertung von (x, k)

- ein S–Ausdruck $\Longleftrightarrow$ der Mode von (x, k') ist *gleich* S–EXPR oder
- ein gewöhnlicher Funktionsidentifikator $\Longleftrightarrow$ der Mode von (x, k') ist *ungleich* S–EXPR.

Somit ist bereits *statisch* entscheidbar, für welche Vorkommen von formalen Identifikatoren zur Laufzeit ein Typ–0, bzw. ein Typ–1 Update nötig werden könnte. Dies ist um so wichtiger, da die in den nächsten beiden Abschnitten vorgestellten Realisierungen für einen Typ–0 bzw. einen Typ–1 Update nicht identisch sind. Der Grund für die verschiedenen Techniken folgt aus folgendem Korollar:

Korollar 9.3 Sei $c = g(\ldots)$ der Aufruf einer NSF g in einem echten LISP/N–Programm Π, und RA_c sei die zugehörige Rücksprungadresse von c. Aufgrund Korollar 9.1 ist das Ergebnis der Ausführung von c entweder ein S–Ausdruck s oder ein gewöhnlicher Funktionsidentifikator einer Funktion f. Zum *Zeitpunkt* der Fortsetzung des Codes an der Stelle RA_c wird dann unterschieden:

- Ist das Ergebnis von c ein S–Ausdruck s, dann enthält der AC zu dem Zeitpunkt, wenn der Code an der Stelle RA_c fortgesetzt wird, einen Verweis in den Heap auf s.
- Ist das Ergebnis von c ein gewöhnlicher Funktionsidentifikator einer Funktion f, dann war zum Zeitpunkt des Aufrufs c mindestens eine pending Parameterliste vorhanden, die nun zur aktuellen Parameterliste für den Aufruf von f wird. Insbesondere bedeutet dies, daß nicht unmittelbar nach c der Code an der Stelle RA_c fortgesetzt wird, sondern gemäß der Kopierregel solange weiterere Aufrufe stattfinden, bis ein S–Ausdruck (d.h. *kein* Funktionsidentifikator!) als Ergebnis im AC anfällt.

Die erste Aussage von Korollar 9.3 verdeutlichen wir am Beispiel 9.2:

Beispiel 9.2 (Fortsetzung): Sei RA_{g_x} die vom Compiler erzeugte Rücksprungadresse für einen möglichen Funktionsaufruf bei der Wertbestimmung von x im Anweisungsteil von g. Dann enthält der AC zum Zeitpunkt des Rücksprungs an die Stelle RA_{g_x} einen Heap–Verweis auf die Liste (A.B).

Die zweite Aussage von Korollar 9.3 verdeutlichen wir am Beispiel 9.4:

Beispiel 9.4 (Fortsetzung): Sei RA_{h_x} die vom Compiler erzeugte Rücksprungadresse für einen möglichen Funktionsaufruf bei der Wertbestimmung von x im Anweisungsteil von h. Dann enthält der AC zum Zeitpunkt des Rücksprungs an die Stelle RA_{h_x} einen Verweis auf das spezielle Atom T im Heap und nicht den gewöhnlichen Funktionsidentifikator der NSF f.

Wir wissen also aufgrund des Ergebnis–Typs welcher Update für einen formalen Identifikator zur Laufzeit durchgeführt werden könnte, um Call By Need zu realisieren. Für die dazu notwendigen Maßnahmen werden wir im folgenden noch zwischen zwei möglichen Vorkommens–Kategorien von formalen Identifikatoren unterscheiden:

Bemerkung 9.2 Betrachten wir die gemäß Definition 2.1 *übersetzten* LISP/N–Programme Π'. Dann teilen wir die möglichen Vorkommen (x, k) von formalen Identifikatoren x in Π' in zwei Kategorien ein:

1. (x, k) kommt *nicht* innerhalb eines aktuellen Parameters eines NSF–Aufrufs in Π' vor oder
2. (x, k) kommt innerhalb eines aktuellen Parameters a_i eines NSF–Aufrufs in Π' vor. (Entsprechend Bemerkung 2.2 kann a_i dann *kein* dicker Parameter sein.)

Im folgenden begründen wir die aufgrund Bemerkung 9.2 notwendig verschiedenen Maßnahmen, die der Compiler bei der Übersetzung von formalen Identifikator–Vorkommen für die Realisierung der Auswertungstrategie Call By Need durchführen muß:

Zu 1.) Wird ein Vorkommen (x, k) gemäß Punkt 1 in Bemerkung 9.2 zur Laufzeit erreicht, so wird unmittelbar der aktuelle Wert von x durch die Aktionen gemäß Tabelle 9.1 bestimmt.

$\Rightarrow$ Die mögl. Auswertung von x findet an der Position (x, k) in Π' statt .

$\Rightarrow$ Der Compiler kann die für einen Typ–0 oder Typ–1 Update notwendigen Maßnahmen bei der Übersetzung von (x, k) in den Zielcode mit aufnehmen (siehe Abschnitte 9.4 und 9.5).

Zu 2.) Wird ein Vorkommen (x, k) gemäß Punkt 2 in Bemerkung 9.2 zur Laufzeit erreicht, so ist x ein aktueller Parameter eines Aufrufs c einer NSF f, also $c = f(\ldots, x, \ldots)$

$\Rightarrow$ Der Inhalt der zu (x, k) zugehörigen AR–Parameterzelle wird lediglich in das neu anzulegende AR_c zum Aufruf c *kopiert*.

$\Rightarrow$ Für einen ggf. mögl. Update der zu (x, k) zugehörenden AR–Zelle ist die *Adresse* dieser Zelle dem Aufruf c mitzugeben.

Bevor wir die Maßnahmen für einen Typ–0 bzw. einen Typ–1 Update im einzelnen vorstellen können, führen wir für die soeben angedeutete Adress–Weitergabe einen neuen Parameter–Typ ein (siehe auch [Hu92]):

9.3 Typ–4 Parameter mit Referenz–Verweis

Die in Punkt 2 von Bemerkung 9.2 angedeutete Situation haben wir schon in Beispiel 9.5 kennengelernt: Nehmen wir das Vorkommen des formalen Identifikators x im linken Ast der NSF g. Damit bei der Wertbestimmung von x im rechten Ast in g der dicke Parameter CAR((A B)) nicht zum 4. mal ausgewertet werden muß, ist es notwendig, die *Adresse* der Parameterzelle zu x den Aufrufen g(x) und e(y) mitzugeben, damit nach der *ersten* Auswertung des dicken Parameters (im linken Ast in e) ein Update der Parameterzelle zu x erfolgen kann.

Zunächst werden wir diese und ähnliche Situationen gemäß dem folgendendem vorläufigem Schema handhaben:

1. Wird ein formaler Identifikator x als aktueller Parameter in einem NSF–Aufruf c weitergegeben, so ist x zunächst vom Typ–3: Gemäß Tabelle 2.1 muß der aktuelle Wert aus der zu x zugehörigen Parameterzelle in das neu anzulegende AR_c kopiert werden. Dabei sei ADR_x die absolute Kelleradresse der Parameterzelle zu x und ADR_y sei die abolute Kelleradresse der neu zu belegenden Parameter–Zelle im AR_c.

   Ist der Parameter an der Adresse ADR_x vom Typ–2, d.h. ein dicker Parameter δ, dann wird dieser Parameter nicht kopiert, sondern folgender neuer Eintrag in dem AR_c vorgenommen:

$$\text{ADR}_y \texttt{ := Typ–4} \mid \text{ADR}_x \ .$$

D.h. wir erhalten einen Typ-4 Parameter mit einem *Referenz–Verweis* auf die „Quelle“ von δ (diejenige Adresse, an welcher δ als Typ–2 Parameter gespeichert wurde).

2. Jeder Zugriff auf eine Parameter–Zelle mit einem Typ–4 Parameter führt über eine indirekte Adressierung sofort zu einem Zugriff an der angegebenen Referenz–Adresse.

3. Ist in der Situation von Punkt 1 der Parameter an der Adresse ADR_x nicht vom Typ–2, so wird er (wie üblich) an die Adresse ADR_y *kopiert*. Dies gilt auch für den neuen Typ–4 Parameter.

Wenden wir diese Strategie auf unser Beispiel 9.5 an, so findet der Aufruf h() der Hilfsfunktion h zum dicken Parameter CAR((A B)) nur noch einmal statt. Beim Aufruf g(h()) wird ein Typ–4 Parameter mit einem Referenz–Verweis auf die Parameterzelle zu x gespeichert. Beim anschließenden Aufruf e(h()) wird dieser Typ–4 Parameter kopiert. Im linken Ast von e wird h() bei der Auswertung des formalen Identifikators z ausgeführt und ein Typ–0 Update der Parameterzelle zu z durchgeführt. Dort ist jedoch ein Typ–4 Parameter mit einer Referenz auf die Parameterzelle zu x gespeichert. Also wird der Typ–0 Update an der Referenz–Adresse durchgeführt und die folgenden Zugriffe auf z und y in den rechten Ästen von CONS lesen dann über die indirekten Zugriffe durch die Typ–4 Zellen den Typ–0 Parameter. Im rechten Ast von f wird direkt auf die Zelle zu x zugegriffen und der dicke Parameter ist insgesamt nur einmal ausgewertet worden.

Das diese Vorgehensweise aber noch nicht korrekt funktioniert, soll uns folgendes einfache Beispiel demonstrieren:

Beispiel 9.6

```
g = λ y z . { CONS(z,y) }
f = λ x . { g(A,x) }
f(CAR((B C)))
```

Sei h die zum dicken Parameter CAR((A B)) vom Compiler eingeführte Nichtstandard–Hilfsfunktion (siehe Abschnitt 2.2). Nach dem NSF–Aufruf f(h()) erfolgt der Aufruf von g: Bei der (zunächst provisorischen) Anlage des $\text{AR}_{g(A,h())}$ wird als zweiter Parameter ein Typ–4 Parameter mit einem Referenz–Verweis auf die Parameterzelle zu x gespeichert. Anschließend wird das $\text{AR}_{f(h())}$ frühzeitig freigegeben und an seine Stelle das $\text{AR}_{g(A,h())}$ gesetzt. Anschließend soll z im linken Ast von CONS ausgewertet werden. Es erfolgt ein indirekter Zugriff über die Referenz–Adresse des Typ–4 Parameters an der absoluten Keller–Adresse des formalen Parameters x. Da das zugehörige $\text{AR}_{f(h())}$ schon frühzeitig freigegen wurde, und das $\text{AR}_{g(A,h())}$ an seiner Stelle

steht, kann die Typ–2 Zelle zu CAR((B C)) nicht mehr erreicht werden und stattdessen wird das Atom A gelesen. Das Programm liefert fälschlicherweise das Endergebnis (A . A), anstatt (B . A).

Um das Problem zu lösen muß sichergestellt werden, daß diejenigen AR's, auf derern Parameterzellen durch Typ–4 Parameter verwiesen wird, sich noch im AR–Keller befinden, d.h. nicht (zu) frühzeitig freigeben werden.

In dem Beispiel 9.5 ist dies dadurch gegeben, daß die AR's mit den Parameterzellen zu x und y durch GDV–Verweise im AR–Keller gehalten wurden. Diesen Umstand nutzen wir in folgendem Lemma aus:

Lemma 9.1 Sei c der Aufruf einer NSF f in einem echten LISP/N–Programm Π, und habe c als aktuellen Parameter den formalen Identifikator x, also $c = f(\ldots, x, \ldots)$. (Gemäß Definition 1.7 kann x auch ein pending Parameter von c sein). Sei ADR_x die absolute Kelleradresse der Parameterzelle zu x im AR_c. Dann ist das AR_c bei einem möglichen Update an der Adresse ADR_x bei der Auswertung von c noch im AR–Keller vorhanden, falls für den beim Aufruf c gültigen maximalen GDV–Verweis (MaxGDV: siehe Abschnitt 6.1.4) gilt:

$$\mathrm{ADR}_x < \text{AR–Keller [MaxGDV + 2]} , \qquad \textbf{(NK1)}$$

wobei mit der rechten Seite der BFS–Verweis in der Linkage des AR zum maximalen GDV angesprochen wird (vgl. Abbildung 8.3).

Beweis (Lemma 9.1): Der Linkage–Verweis BFS (Beginn Freier Speicher) verweist immer an den Anfang des folgenden AR oder an die erste freie Stelle im AR–Keller, falls es sich um den BFS–Verweis in dem obersten AR handelt. Falls NK1 gilt, so liegt die ADR_x entweder innerhalb desjenigen AR, auf welches der MaxGDV verweist, oder an noch tieferer Stelle im AR–Keller.
$\Longrightarrow$ Aufgrund der Definition 2.2 des GDV–Verweises wird ein GDV–Verweis erst dann wieder auf den alten Wert zurückgesetzt, wenn derjenige Aufruf c' beendet ist, der zum Setzen des GDV–Verweises führte (diese kellerartige Verwaltung gilt natürlich auch für den neuen GDV–Verweis!). Dieser Aufruf c' ist aber entweder mit c identisch oder erfolgte bereits *vor* c.
$\Longrightarrow$ Während der Ausführungen zum Aufruf c ist die ADR_x eine durch den GDV–Verweis vor der frühzeitigen Speicherplatzfreigabe geschützte Zelle und das AR_x ist bei einem Update an der Adresse ADR_x während der Ausführungen von c noch im AR–Keller vorhanden. □

Bemerkung 9.3 Das Problem kann auch auf folgende Weise gelöst werden: Die in Typ–4 Parameterzellen vorhandenen Verweise werden bei der zunächst provisorischen Anlage eines AR ebenfalls überprüft und das referenzierte AR dadurch nicht freigegeben. Dadurch können dann aber auch frühzeitige Speicherplatzfreigaben verhindert werden: In Beispiel 9.6 würde das $\mathrm{AR}_{f(h())}$ durch den folgenden Aufruf g(A,h()) dann nicht freigegeben.

Bemerkung 9.4 Da Typ–4 Parameter nur auf AR's verweisen die bereits durch einen GDV–Verweis geschützt sind, braucht für Typ–4 Parameter *kein* Verweis mehr im Rahmen einer frühzeitigen Speicherplatzfreigabe überprüft werden!

Bevor wir die nun möglichen Parameter–Typen in einer Tabelle veranschaulichen und die Handhabung des neuen Typ–4 Parameters dort zusammenfassen, erläutern wir, warum unsere Call By Need–Optimierung nicht immer vollständig sein kann. Dazu betrachten wir zunächst folgendes Beispiel:

Beispiel 9.7

```
g = λ v w . { CONS(v,w) }
f = λ x . { g(x,x) }
f(CAR((A B)))
```

Der Aufruf h() der Hilfsfunktion zum dicken Parameter CAR((A B)) wird der NSF f als Typ–2 Parameter übergeben. Anschließend wird beim Aufruf g(h(),h()) der dicke Parameter *dupliziert*. Im Anweisungsteil von g erfolgt dann die Ausführung von h() sowohl durch Auswertung von v als auch durch Auswertung von w. Aufgrund des nicht erfüllten Kriteriums NK1 beim Aufruf g(h(),h()) werden keine Typ–4 Parameterzellen für die beiden (gleichen) Typ–2 Parameter eingerichtet — das $AR_{f(h())}$ wäre zum Zeitpunkt eines Updates auch schon nicht mehr im AR–Keller.

Bemerkung 9.5 Die in Bemerkung 9.3 angedeutete Alternative könnte dieses Problem lösen: Die Verweise der dann eingerichteten Typ–4 Parameterzellen beim Aufruf von g würden das $AR_{f(h())}$ vor der frühzeitigen Freigabe schützen. Wie schon erwähnt, kann diese Vorgehensweise aber zu einem erhöhten Speicherplatzbedarf führen und Verweise der Typ–4 Parameter müßten weiterhin überprüft werden (siehe Bemerkung 9.4). Ferner liegen die Konstellationen, die nicht zu einer Weitergabe einer Referenz–Adresse durch einen Typ–4 Parameter führen, häufig im Ermessen des Programmierers: Das Programm in Beispiel 9.7 könnte auch wie folgt formuliert werden:

```
g = λ v . { CONS(v,v) }
f = λ x . { g(x) }
f(CAR((A B)))
```

Dann würde zwar wieder kein Typ–4 Parameter eingerichtet, jedoch würde die Auswertung des formalen Identifikators v im linken Ast von CONS zu einem Typ–0 Update führen (siehe Abschnitt 9.4) und die Hilfsfunktion h nur einmal ausgewertet werden.

Bemerkung 9.6 Eine aussichtsreichere Alternative wäre zu untersuchen, ob aktuelle Typ–2 Parameter eines NSF–Aufrufs identisch sind. Falls sie gleich sind, könnte mit Typ–4 Parametern innerhalb eines AR auf einen der gleichen Parameter verwiesen werden.
→ Es bleibt zu untersuchen, wieweit diese Maßnahmen bereits statisch vorbereitet werden können, denn dynamisch wäre die Untersuchung auf Gleichheit vermutlich relativ aufwendig.

Wir ersetzen nun die auf Seite 19 angegebene Tabelle 2.1 durch die Tabelle 9.2 mit dem neuen Typ–4 Parameter. Die neue Handhabung formaler Identifikatoren als aktuelle Parameter von NSF–Aufrufen ist in den Hinweisen zur Tabelle zusammengefaßt:

Tabelle 9.2: Die neuen möglichen Einträge für Parameter in einem AR

Typ des Parameters	Typ-Nr.	Adresse	statischer Verweis
Konstante (S-Ausdruck)	0	Heap-Adresse	statisches Niveau[1]
Funktionsidentifikator	1	Startadresse	ein dyn. Niveau des stat. Vorgäng.[3]
Funktionsidentifikator für einen „dicken Parameter“[2]	2	Startadresse	ein dyn. Niveau des stat. Vorgäng.[3]
Parameter mit Referenz–Verweis[5]	4	AR-Keller-Adresse	—[6]
Formaler Identifikator	(3)	aktueller Wert[4]	

Hinweise zu Tabelle 9.2:

Die Hinweise (1) − (3) sind wörtlich denen zu Tabelle 2.1 zu übernehmen. Geändert bzw. hinzugekommen sind folgende Erklärungen:

(4) Formale Identifikatoren x werden zunächst als Typ–3 Parameter übergeben. Das Laufzeitsystem ermittelt dann unmittelbar den aktuellen Wert aus der zu x zugehörigen Parameterzelle an der Adresse ADR_x und macht daraufhin folgenden Eintrag an der Adresse ADR_y des neu anzulegenden Parameters:

- Ist der Parameter an der Adresse ADR_x vom Typ–0 oder Typ–1, so wird dieser an die Adresse ADR_y *kopiert*.
- Ist der Parameter an der Adresse ADR_x vom Typ–2, und ist das Kriterium NK1 erfüllt, d.h. gilt:

$$\mathrm{ADR}_x < \text{AR–Keller [MaxGDV + 2]} ,$$

so wird folgender Eintrag an der Adresse ADR_y vorgenommen:

$$\mathrm{ADR}_y \texttt{ := } \text{Typ–4 | } \mathrm{ADR}_x$$

(5) Soll der Wert eines formalen Identifikators durch Zugriff auf die zugehörige Parameterzelle an der Adresse ADR_y ermittelt werden und ist dort ein Typ–4 Referenz–Parameter eingetragen, so wird der Zugriff an die dort eingetragene AR–Keller–Adresse ADR_x umgeleitet.

(6) Ein statischer Verweis auf den Anfang eines AR ist nicht mehr Vorhanden. Für die Bestimmung des von einem zunächst provisorisch angelegten AR ausgehenden maximalen Verweises in den AR–Keller brauchen Typ–4 Parameterzellen *nicht* überprüft werden (siehe Bemerkung 9.4).

In den nächsten beiden Abschnitten stellen wir nun die konkreten Maßnahmen zur Call By Need–Realisierung vor:

9.4 Der Typ–0 Update

Ist der zu einem formalen Identifikator x zugehörende Mode *gleich* S–EXPR, so muß nach Korollar 9.1 die Auswertung von x zur Laufzeit immer einen S–Ausdruck, d.h. ein Atom oder eine Liste liefern (falls die Auswertung terminiert). Aufgrund des Korollar 9.3 werden wir hier den Typ–0 Update unter Zuhilfenahme des RA–Kellers vorstellen.

Bevor die Aktionen zur Wertbestimmung eines formalen Identifikators x und evtl. anschließend ein Typ–0 Update durchgeführt werden können (siehe Tabelle 9.1), muß der Typ der zugehörigen Parameterzelle bestimmt werden. Dazu erfolgt ein einfacher Zugriff in den AR–Keller an der Adresse ADR_x mit

$$\mathrm{ADR}_x \texttt{ := } \mathrm{IR}[\mathrm{SN}(x)] + \mathrm{Rel.Adr.}(x)$$

Ist dort ein Parameter vom Typ–4 gespeichert, so wird gemäß Tabelle 9.2 (Hinweis (5)) der Zugriff an die dort eingetragene Referenz–Adresse umgeleitet:

$$\mathrm{ADR}_x \texttt{ := } \text{Referenz–Verweis in AR-Keller}[\mathrm{ADR}_x]$$

Ist der an der Adresse ADR_x eingetragene Parameter nun vom Typ–0, so wird lediglich der Accumulator (AC) mit der dort eingetragenen Heapadresse geladen. Ein Parameter vom Typ–1 ist nicht zulässig, wenn der Mode von x gleich S–EXPR ist.

9.4.1 Entscheidungs–Kriterium für einen Typ–0 Update

Ist an der Adresse ADR_x ein Typ–2 Parameter eingetragen, so muß zur Wertbestimmung von x ein dicker Parameter ausgewertet werden. Dabei ist sowohl ein Typ–0 Update als auch der in Abschnitt 9.5 eingeführte Typ–1 Update nur notwendig, wenn *nach* der Auswertung des dicken Parameters durch

einen erneuten Zugriff an der Adresse ADR_x derselbe dicke Parameter nochmals ausgewertet werden kann. Mit dem Kriterium in folgendem Hilfssatz werden wir *sicher unnötige* Typ–0 Updates vermeiden:

Lemma 9.2 Sei in der Parameterzelle an der Adresse ADR_x in einem AR_c ein Typ–2 Parameter gespeichert. Ist der Mode des zugehörigen formalen Parameters x gleich S–EXPR und wird der Wert von x durch Zugriff an der Adresse ADR_x benötigt so muß der Aufruf $c' = h()$ einer Nichtstandard–Hilfsfunktion h ausgeführt werden (siehe Abschnitt 2.2). Sei folgendes Kriterium NK2 gegeben:

$$\mathrm{ADR}_x \geq \text{AR–Keller [max(MaxGDV,PPmaxV) + 2] ,} \qquad \textbf{(NK2)}$$

wobei mit der rechten Seite der BFS–Verweis des AR zum maximalen GDV oder des AR, auf welches der maximale Verweis aus dem PP–Keller verweist (siehe Abschnitt 8.2), angesprochen wird.

Dann gilt: Falls beim Aufruf c' das Kriterium NK2 erfüllt ist, so kann *nach* c' an der Adresse ADR_x im AR_c kein Zugriff mehr erfolgen.

Beweis (Lemma 9.2): Da das Prinzip der Honschopp–Optimierung die frühzeitige Speicherplatzfreigabe in der Situation „Funktionsaufruf“ ist, *muß* das AR_c bei jedem NSF–Aufruf durch Verweise im AR–Keller gehalten werden, wenn evtl. noch Informationen daraus benötigt werden könnten! Dabei haben wir nun alle diejenigen Verweise zu untersuchen, die c' „überleben“ können bzw. erst unmittelbar *nach* c' zurückgesetzt werden können. Verweise, die durch weitere NSF–Aufrufe während c' entstehen, müssen vor dem Ende von c' wieder verschwunden sein, können c' also nicht überleben. Deshalb brauchen wir nur Verweise zu untersuchen, die entweder beim Aufruf c' entstehen, oder bereits vorhanden sind. Dabei unterscheiden wir die vier möglichen Bereiche, aus denen Verweise das AR_c schützen könnten:

1. Der GDV–Verweis (im GDV–Keller),
2. der Verweis auf ein dynamisches Niveau des statischen Vorgänger (in der 2. Linkage–Zelle von $\mathrm{AR}_{c'}$ — vgl. Abbildung 8.3),
3. die aktuellen Parameter von c' (im $\mathrm{AR}_{c'}$) und
4. die pending Parameter von c' (im PP–Keller).

Die erste und vierte Möglichkeit wird bereits durch die Vorraussetzung ausgeschlossen. Die dritte Möglichkeit ist auch nicht gegeben, da die vom Compiler eingeführten Hilfsfunktionen h immer parameterlos sind (siehe Abschnitt 2.2) und der Aufruf c' selber keine weiterern Parameterlisten haben darf (c' hat einen S–Ausdruck als Ergebnis). Die zweite Möglichkeit ist auch unmöglich, da dann der statische Verweis in der Typ–2 Parameterzelle an der Adresse

ADR_x auf das eigene AR_c zeigen müßte (ähnliche Situation wie im Beweis 4.1).
$\Rightarrow$ Die Adresse ADR_x wird beim Aufruf c' weder durch einen GDV–Verweis, noch durch einen statischen Verweis aus möglicherweise im PP–Keller gespeicherten pending Parametern geschützt und deshalb kann *nach* c' *kein* Zugriff mehr an der Adresse ADR_x im AR_c erfolgen. □

Das Kriterium NK2 sichert somit zum einen die Existenz einer zu aktualisierenden Parameterzelle bei einem Typ–0 Update und zum anderen gewährleistet es, daß durch Auslassen der Typ–0 Update–Maßnahmen aufgrund des Kriteriums NK2 *keine* mehrfache Auswertung des selben dicken Parameters stattfinden kann.

9.4.2 Beschreibung des Typ–0 Updates

Für die folgende Beschreibung der Maßnahmen für einen Typ–0 Update verwenden wir die gleichen Bezeichnungen wie in Lemma 9.2. Es sei also an der Adresse ADR_x im AR_c ein Typ–2 Parameter vorhanden, dessen Auswertung einen S–Ausdruck liefert und dessen Wert nun benötigt wird:

1. Falls beim Aufruf c' ADR_x<AR–Keller[max(MaxGDV,PPmaxV)+2] gilt (also NK2 *nicht* erfüllt ist), protokollieren wir die Adresse ADR_x, indem wir sie zusammen mit der $RA_{c'}$ von c' im RA–Keller abspeichern:

   $$\text{PUSH}_{\text{RA-Keller}}(\ ADR_x \mid RA_{c'}\ )$$

   Falls NK1 nicht erfüllt ist, wird wie üblich nur die $RA_{c'}$ im RA–Keller abgelegt.

2. Der Aufruf c' wird wie üblich abgearbeitet.

3. Ist c' ausgeführt, so liegt sein Ergebnis als S–Ausdruck im AC, d.h. einem Verweis in den Heap vor. Der oberste Eintrag im RA–Keller ist (wieder) die $RA_{c'}$ von c'. Bevor die Abarbeitung an der Stelle $RA_{c'}$ im Code fortgesetzt wird, wird

   $$ADR_x \ := \text{POP}_{\text{RA-Keller}}$$

   ausgeführt (wobei die $RA_{c'}$ an dieser Stelle ausgeblendet wird).

4. Falls eine Adresse gespeichert wurde, d.h. gilt

   $$ADR_x > 0\ ,$$

   so wird an der Adresse ADR_x ein Typ–0 Update wie folgt durchgeführt:

   $$\text{AR–Keller}\ [\ ADR_x\ ]\ :=\ \text{Typ–0} \mid \text{Verweis im AC} \mid 0\ .$$

An der Adresse ADR_x ist nun ein Typ–0 Parameter mit einem Heap–Verweis sowie dem (trivialen) statischen Niveau 0 gespeichert.

5. Jede weitere versuchte Auswertung von x durch direkten oder indirekten Zugriff (letzterer durch Typ–4 Parameter) auf die soeben aktualisierte Zelle ergibt dann keine Typ–2 sondern eine Typ–0 Parameterzelle, und das Ergebnis liegt unmittelbar vor.

Es folgt noch eine abschließende Bemerkung zum Entscheidungs–Kriterium NK2:

Bemerkung 9.7 In Abschnitt 9.3 haben wir gesehen, daß die Anwendung des Kriteriums NK1 für die Entscheidung, ob ein Typ–4 Parameter mit Referenz–Verweis eingerichtet wird, oder ob kopiert werden soll, eine *notwendige* Maßnahme zur Korrektheit der Optimierung ist (siehe Beispiel 9.6).

In Punkt 1 der Beschreibung des Typ–0 Updates wird das Kriterium NK2 benutzt, um sicherzustellen, daß das AR mit der zu aktualisierenden Parameterzelle noch nicht frühzeitig freigegeben wurde. Somit werden sicher unnötige Typ–0 Updates vermieden. Dies ist jedoch *keine* notwendige Maßnahme zur Korrektheit der Optimierung: Ist das zu aktualisierende AR frühzeitig freigegeben worden, so kann dies nur bei der Auswertung des dicken Parameters erfolgt sein. Diese Auswertung ist aber zum Zeitpunkt des Typ–0 Updates komplett abgeschlossen und der unötige Update erfolgt dann *oberhalb* des aktuell genutzten AR–Kellers und kann keinen Schaden anrichten — es tritt lediglich ein erhöhter Aufwand ein.

Im nächsten Abschnitt stellen wir eine Realisierung des Typ–1 Updates vor, die mit ähnlich einfachen Kriterien und Maßnahmen wie der Typ–0 Update auskommt und ebenfalls in vielen Aufruf–Situationen „optimal“ arbeitet, d.h. die Auswertung eines dicken Parameters nur einmalig zuläßt:

9.5 Der Typ–1 Update

Ist der zu einem formalen Identifikator x zugehörende Mode *ungleich* S–EXPR, so muß nach Korollar 9.1 die Auswertung von x zur Laufzeit immer einen gewöhnlichen Funktionsidentifikator liefern (falls die Auswertung terminiert). Aufgrund des Korollars 9.3 werden wir einen Typ–1 Update nicht unter Zuhilfenahme des RA–Kellers realisieren können, da zum Zeitpunkt der Fortsetzung des Codes an der RA_c von c das Ergebnis von c nicht mehr vorliegt.

Wir nutzen die in Korollar 9.3 angedeutete Tatsache aus, daß zu Beginn der Auswertung von c eine pending Parameterliste im PP–Keller (siehe Abschnitt 8.2) eingetragen sein muß, welche nach Beendigung von c für einen vollständigen Aufruf der Ergebnis–Funktion herangezogen wird. Deshalb werden wir

die absolute Kelleradresse einer später zu aktualisierenden Typ–2 Parameterzelle nicht im RA–Keller, sondern im PP–Keller protokollieren. Zu dem Zeitpunkt, wenn diese pending Parameterliste als aktuelle Parameterliste in ein AR übernommen wird (also c gerade beendet ist!), wird dann die für die Anlage des AR verantwortliche Funktion als Typ–1 Parameter an die Adresse der Typ–2 Zelle geschrieben.

9.5.1 Entscheidungs–Kriterien für einen Typ–1 Update

In Bemerkung 9.7 haben wir festgehalten, daß die Sicherung des zu aktualisierenden AR im AR–Keller mittels Kriterium NK2 *keine* notwendige Maßnahme ist, jedoch durchgeführt wird, weil dadurch keine Updates ausgelassen werden und ein (geringer) unötiger Mehr–Aufwand für einen Typ–0 Update vermieden werden kann.

Für den Typ–1 Update ist es jedoch eine *notwendige* Bedingung, daß das zu aktualisierende AR zum Zeitpunkt des Updates noch im AR–Keller steht: Durch die Auswertung des dicken Parameters kann das AR mit der Typ–2 Parameterzelle frühzeitig freigegeben werden. Da der Zeitpunkt des Updates von der Übernahme der pending Parameterliste als aktuelle Parameterliste in ein AR abhängt, können an der dann mit dem erzeugten Typ–1 Parameter überschriebenen Adresse ADR_x bereits Informationen stehen, die für den laufenden Aufruf noch benötigt werden! Dazu folgendes kurze Beispiel:

Beispiel 9.8

```
g = λ v . { p = λ y . { v(y) }
            p }
f = λ x . { x((A B)) }
f(g(CAR))
```

Sei h die Hilfsfunktion, die vom Compiler für den dicken Parameter g(CAR) eingeführt wird (siehe Abschnitt 2.2). Beim ersten Aufruf f(h()) wird im $\mathrm{AR}_{f(h())}$ eine Typ–2 Parameterzelle eingerichtet. Sei diese Adresse ADR_x. Wenden wir kein Kriterium zur Sicherstellung der Existenz des $\mathrm{AR}_{f(h())}$ an, so wird beim folgenden NSF–Aufruf h()((A B)) die Adresse ADR_x der PP–Liste ((A B)) mitgegeben und bereits jetzt das $\mathrm{AR}_{f(h())}$ frühzeitig freigegeben. Es folgt der Aufruf p((A B)), bei dem die Liste ((A B)) zur aktuellen Parameterliste wird. Würde jetzt bei einem Typ–1 Update an der Adresse ADR_x geschrieben werden, so würde nicht der originale Typ–2 Parameter erreicht werden, sondern es würde der im folgenden Aufruf benötigte Typ–1 Parameter CAR im $\mathrm{AR}_{g(CAR)((AB))}$ überschrieben werden!
Hinweis: Dieses Beispiel werden wir in leicht modifizierter Form in Beispiel 9.9 wiederfinden und dort auch den Ablauf im AR–Keller veranschaulichen. Die Adresse ADR_x wird dort zwar durch einen GDV–Verweis fixiert, jedoch wird dann ein durchgeführter Typ–1 Update aus ganz anderem Grunde weiterhin zu einem Fehler führen (s.u.).

Für das notwendige Enscheidungskriterium verwenden wir im folgenden Lemma das bereits eingeführte Kriterium NK2 und beweisen anschließend die Relevanz für die neue Situation:

Lemma 9.3 Sei in der Parameterzelle an der Adresse ADR_x in einem AR_c ein Typ–2 Parameter gespeichert. Ist der Mode des zugehörigen formalen Parameters x ungleich S–EXPR und wird der Wert von x durch Zugriff an der Adresse ADR_x benötigt, so findet folgender Aufruf statt:

$$c' = h()(\ldots)^1 \ldots (\ldots)^n \ ,$$

wobei $n \geq 1$ pending Parameterlisten vorhanden sein müssen. Falls beim Aufruf c' das Kriterium NK2 *nicht* erfüllt ist, d.h.

$$\mathrm{ADR}_x < \text{AR–Keller [max(MaxGDV,PPmaxV) + 2]}$$

gilt, so ist das AR_c bei einem Typ–1 Update zum Zeitpunkt der Übernahme der pending Parameterliste $(\ldots)^1$ als aktuelle Parameterliste für einen Folge–Aufruf noch im AR–Keller vorhanden.

Beweis (Lemma 9.3): i.) Falls MaxGDV≥PPmaxV gilt, so ist das AR_c durch einen GDV–Verweis während der ganzen Ausführung von c' geschützt (siehe Beweis zu Lemma 9.1). Weil c' aufgrund der Kopierregel erst beendet ist, wenn alle pending Parameter von c' verbraucht sind, ist somit das AR_c zum Zeitpunkt der Übernahme von $(\ldots)^1$ noch im AR–Keller vorhanden.
ii.) Falls PPmaxV≥MaxGDV gilt, so ist das AR_c durch einen statischen Verweis aus dem PP–Keller geschützt, und zwar solgange, bis der oder die betreffenden Parameter als aktuelle Parameter verbraucht sind.
$\Rightarrow$ Das AR_c steht mindestens bis zu dem Zeitpunkt im AR–Keller, bis $(\ldots)^1$ zu einer aktuellen Parameterliste wird.
$\Longrightarrow$ Aus i. und ii. folgt die Behauptung. □

Umgekehrt ist noch zu zeigen, daß durch Auslassen von Typ–1 Updates aufgrund eines nicht erfüllten Kriteriums NK2 keine mehrfache Auswertung eines dicken Parameters eintreten kann. Dazu formulieren wir zunächst folgendes Lemma und übertragen dann die neue Situation auf die Aussage von Lemma 9.2:

Lemma 9.4 Sei die Situation gemäß Lemma 9.3 gegeben. Falls beim Aufruf c' das Kriterium NK2 aus Lemma 9.2 gilt, so kann nach der Übernahme der Liste $(\ldots)^1$ als aktuelle Parameterliste *kein* Zugriff mehr an der Adresse ADR_x im AR_c erfolgen.

Beweis (Lemma 9.4): Es kann der Beweis zu Lemma 9.2 wörtlich übernommen werden, nur mit dem Unterschied, daß der Aufruf c' dort mit der Auswertung des dicken Parameters bereits beendet ist. Der für die Aussage relevante Zeitpunkt eines Updates ist aber auch in der Situation von Lemma 9.4 der gleiche: Wird die PP–Liste $(\ldots)^1$ als aktuelle Parameterliste benötigt,

so ist der aktuelle Aufruf das Ergebnis der unmittelbar zuvor beendeten Auswertung des dicken Parameters. □

Genau wie beim Typ–0 Update sichert das Kriterium NK2 für einen Typ–1 Update zum einen die (hier notwendige!) Existenz einer zu aktualisierenden Parameterzelle und zum anderen gewährleistet es, daß durch Auslassen der Typ–1 Update–Maßnahmen aufgrund des Kriteriums NK2 *keine* mehrfache Auswertung des selben dicken Parameters stattfinden kann.

Im folgenden Beispiel sehen wir jedoch, daß ein Typ–1 Update an der Adresse der Typ–2 Parameterzelle nicht immer möglich ist und in einigen Fällen zu einer falschen Konstellation in der statischen Verweiskette führen kann:

Beispiel 9.9

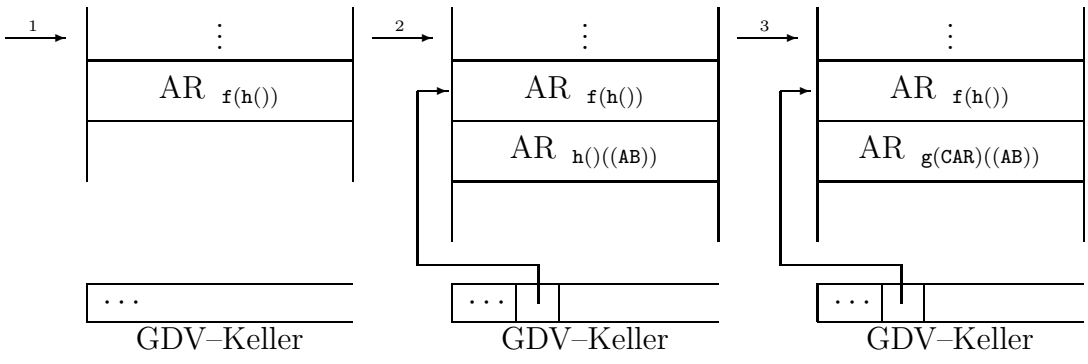
```
g = λ v . { p = λ y . { v(y) }
            p }
f = λ x . { CONS( x((A B)) , x((C D)) ) }
f(g(CAR))
```

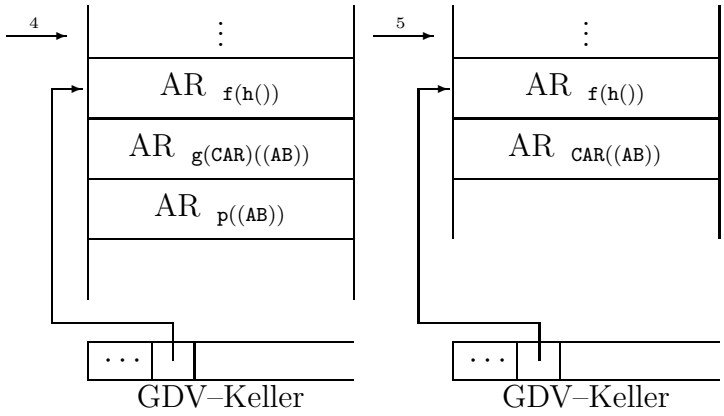
Bevor wir den Ablauf für die ersten fünf Aufrufe (also bis einschließlich der Auswertung des linken Astes von CONS) im AR– und GDV–Keller veranschaulichen, folgende

Bemerkung 9.8 In der Honschopp–Implementation [Ho83] kann auch für einen SF–Aufruf ein AR angelegt werden! Dies kommt in folgenden beiden Situationen vor:

- Ein SF–Ausdruck ist das Ergebnis einer NSF–Aufrufs und wird mit einer notwendig vorhandenen pending Parameterliste als aktuelle Parameterliste versorgt oder
- die Auswertung des formalen Identifikators x in einem formalen Funktionsaufruf x(...) liefert eine SF.

In beiden Fällen wird vom Laufzeitsystem für den SF–Aufruf ein AR angelegt, um eine einheitliche und effiziente Parameterübergabe beizubehalten.





zu 1. Die NSF f wird mit dem dicken Parameter g(CAR) als Argument aufgreufen. Es wird eine Typ–2 Parameterzelle an einer Adresse ADR_x eingerichtet.

zu 2. Der Aufruf des dicken Parameters erfolgt nun im linken Ast von CONS durch Auswertung des formalen Identifikators x und das Kriterium NK2 ist erfüllt, da die für den Typ–2 Aufruf verantwortliche Parameterzelle an der Adresse ADR_x durch einen GDV–Verweis im Keller gehalten wird. Zusammen mit der Liste (A B) wird die Adresse ADR_x im PP–Keller protokolliert. (Details werden in Abschnitt 9.5.2 erläutert.)

zu 3. Die pending Parameterliste (A B) wird an den Aufruf von g weitergereicht, d.h. bleibt dank des neuen PP–Kellers einfach dort einfach stehen.

zu 4. Dies ist der Zeitpunkt, an dem die pending Parameterliste (A B) als aktuelle Parameterliste gebraucht wird. Der Funktionsausdruck p ist das Ergebnis des Aufrufs des dicken Parameters g(CAR). Würde nun an der ebenfalls aus dem PP–Keller geholten Adresse ADR_x ein Typ–1 Update erfolgen, so würde ein Fehler in der statischen Verweiskette entstehen: Jede Typ–1 Parameterzelle enthält einen statischen Verweis auf ein dynamisches Niveau des statischen Vorgängers (siehe Tabelle 9.2). Der statische Vorgänger von p ist aber die NSF g. Würde der entsprechende Verweis auf das $\text{AR}_{g(CAR)((A\ B))}$ in einer Typ–1 Parameterzelle an der Adresse ADR_x geschrieben werden, so würde auf das *nachfolgende* AR verwiesen. „Rückwärts“ gerichtete Verweise sind aber im AR–Keller nicht zulässig, da die dadurch verwiesen AR's vor einer frühzeitigen Freigabe nicht mehr geschützt werden könnten.

Diese mögliche Fehler–Situation werden wir nun durch ein sehr einfaches und dynamisch schnell zu überprüfendes Kriterium NK3 ausschließen:

Korollar 9.4 Eine Typ–1 Parameterzelle darf nur dann im Rahmen eines Typ–1 Updates an der Parameter–Adresse ADR_x geschrieben werden, falls dabei folgendes Kriterium erfüllt ist:

$$n < \mathrm{ADR}_x \ , \qquad \textbf{(NK3)}$$

wobei n der notwendige Eintrag „ein dynamisches Niveau des statischen Vorgängers“ (siehe Tabelle 9.2) in der zu schreibenden Typ–1 Zelle sei.

Mit dem Kriterium NK3 wird somit gewährleistet, daß bei einem erneuten späteren Zugriff an der Adresse ADR_x einer ursprünglichen Typ–2 Parameterzelle nur dann ein Typ–1 Parameter vorliegt, wenn die statische Verweiskette aufgrund des dort vorhandenen statischen Verweises auch korrekt umgeladen werden kann.

In Beispiel 9.9 würde somit *kein* Typ–1 Update durchgeführt werden und der dicke Parameter g(CAR) muß im rechten Ast von CONS ein zweitesmal ausgewertet werden. Das die Relevanz des Typ–1 Updates dennoch gegeben ist, zeigen die beiden einführenden Beispiele 9.3 und 9.4. Sie werden vollständig im Sinne von Call By Need optimiert, d.h. jeder dicke Parameter nur maximal einmal ausgewertet. Wir kommentieren weitere Beispiele in Abschnitt 9.6.

9.5.2 Beschreibung des Typ–1 Updates

Für die folgende Beschreibung der Maßnahmen für einen Typ–1 Update sei die Situation gemäß Lemma 9.3 gegeben. Es sei also an der Adresse ADR_x im AR_c ein Typ–2 Parameter vorhanden, dessen Auswertung einen Funktionsidentifikator liefert, welcher nun benötigt wird:

1. Falls beim Aufruf c' ADR_x<AR–Keller[max(MaxGDV,PPmaxV)+2] gilt (also NK2 *nicht* erfüllt ist), protokollieren wir die Adresse ADR_x, indem wir sie dem aktuell obersten Eintrag $-PPmaxV$ im PP–Keller (vgl. Abbildung 8.2, wobei die Liste $(\ldots)^1$ aus Lemma 9.3 dort der Liste $(b_1, \ldots, b_m)$ entspricht) hinzufügen:

   $$\text{PP–Keller[PPtop}-1] \ := \ - \ \mathrm{ADR}_x \mid \text{PPmaxV}$$

   Falls NK2 jedoch erfüllt ist, wird nichts gemacht.

2. Die Auswertung des Aufrufs c' wird wie üblich begonnen.

3. Soll eine pending Parameterliste als aktuelle Parameterliste in ein AR kopiert werden, so wird zunächst

   $$\mathrm{ADR}_x \ := \mathrm{POP}_{\text{PP-Keller}}$$

   ausgeführt (wobei die Information −PPmaxV ausgeblendet wird).

4. Falls eine Adresse gespeichert wurde, d.h. gilt

$$\mathrm{ADR}_x > 0 \ ,$$

so wurde neben der Information PPmaxV zusätzlich die Kelleradresse ADR_x einer Typ–2 Parameterzelle gespeichert. Der Funktionsidentifikator f des gerade auszuführenden Aufrufs ist dann das Ergebnis der Auswertung eines dicken Parameters an der Adresse ADR_x. Ist das Kriterium NK3 erfüllt und gilt somit

$$\mathrm{DN}(\mathrm{SV}(f)) < \mathrm{ADR}_x \ ,$$

so wird ein Typ–1 Update an der Adresse ADR_x wie folgt durchgeführt:

$$\text{AR–Keller [} \mathrm{ADR}_x \text{] := Typ–1 | } \mathrm{Startadr}_f \text{ | } \mathrm{DN}(\mathrm{SV}(f)) \ .$$

An der Adresse ADR_x ist dann ein Typ–1 Parameter mit einer Startadresse und einem (erreichbaren!) dynamischen Niveau des statischen Vorgängers gespeichert.

5. Die Abarbeitung von c' wird wie üblich fortgesetzt, nur daß jede weitere versuchte Auswertung von x durch direkten oder indirekten Zugriff (letzterer durch Typ–4 Parameter) auf die soeben aktualisierte Zelle an der Adresse ADR_x im AR_c keine Typ–2, sondern eine Typ–1 Parameterzelle liefert, und das Ergebnis unmittelbar vorliegt.

9.6 Semantik und Relevanz der Call By Need–Realisierung

Falls auf eine aktualisierte Parameterzelle nicht mehr zugegriffen wird, kann sich die Semantik des Programms trivialerweise nicht geändert haben. Daß dies aber auch der Fall ist, wenn ein Update „erfolgreich“ ist, d.h. auf die aktualisierte Zelle nach einem Update mindestens noch einmal zugegriffen wird, enthält die Aussage des folgenden Satzes:

Satz 9.1 Ein Typ–0 Update bzw. ein Typ–1 Update verändert die Semantik eines Programms nicht.

Wir geben hier keinen formalen *Beweis* an, sondern stellen die wesentlichen beiden Gründe für die Korrektheit des Satzes kurz vor:

1. Die Bindungsrelation δ_Π eines LISP/N–Programms Π ändert sich während der Programmausführung nicht, da das LISP/N–Laufzeitsystem eine *statische Variablenbindung* (static scoping) besitzt.

2. LISP/N mit dem hier betrachteten Sprachumfang von Pure–LISP ist *seiteneffekt–frei*, d.h. es gibt keine destruktiven Sprachkonstrukte wie z.B. SETQ, RPLACA oder NCONC, die es erlauben, den Wert einer bereits belegten Speicherzelle nachträglich zu manipulieren.

Es wird somit bei jedem Zugriff auf den Wert eines formalen Identifikators immer der gleiche, zum Zeitpunkt der Definition (binden eines aktuellen Parameters eines NSF–Aufrufs an den zugehörigen formalen Parameter der gerufenen NSF) gültige Wert ermittelt. Die mehrfache Auswertung desselben Typ–2 Parameters ergibt deshalb auch immer das gleiche Ergebnis, und es ändert sich an der Semantik nichts, wenn direkt das Ergebnis nach einem Update ermittelt wird. □

Bei einer Call By Need–Realisierung kann im allgemeinen nicht ausgeschlossen werden, daß auf eine aktualisierte Parameterzelle *nicht* mehr zugegriffen wird. Dies folgt unmittelbar aus folgender

Bemerkung 9.9 Im allgemeinen ist die *Erreichbarkeit* eines Identifikators statisch nicht entscheidbar. Für unsere Call By Need–Realisierung bedeutet dies, daß im allgemeinen statisch nicht bekannt ist

- wie *oft* auf die zu einem formalen Identifikator zugehörige AR–Parameterzelle zugegriffen wird und
- durch *welche* formalen Identifikator–Vorkommen in den Anweisungsteilen eines Programms der Zugriff zur Laufzeit (falls überhaupt) erfolgt.

(Übliches Beispiel: der Ablauf eines Konditionals kann von den aktuellen Eingabedaten des Programms abhängen. / siehe auch: „Die Unlösbarkeit des allgemeinen SB–Optimierungsproblems“ [Fe87]).

Daher muß für *jede* Auswertung eines dicken Parameters, für die gemäß der Kriterien NK1 bis NK3 ein Update unterstützt werden kann, ein Update auch durchgeführt werden, um möglichst vollständig im Sinne von Call By Need zu operieren. Dazu folgende

Bemerkung 9.10 Bei allen Maßnhamen zur Call By Need–Unterstützung (Typ–4 Parameter, Typ–0 und Typ–1 Update) handelt es sich im wesentlichen um sehr einfache Aktionen, die bei einer effizienten Implementierung zeitlich unerheblich sein sollten (indirekte Zugriffe, Lesen und Schreiben von Zellen, „>“– und „<-Entscheidungen, ...). Durch die Typ–4 Parameterzellen kann sogar Laufzeit eingespart werden: Es wird zwar einmalig das Kriterium NK1 überprüft, jedoch braucht bei einer Weitergabe eines Typ–4 Parameters nie mehr der statische Verweis für eine mögliche frühzeitige Speicherplatzfreigabe überprüft werden. Durch die Speicherung der zu aktualisierenden Adressen zusammen mit bereits benötigten Informationen (RA bzw. PPmaxV), wie dies in den Parameterzellen eines AR üblich ist, wird der vorhandene Speicherplatz effizient genutzt, und es tritt *kein* höherer Bedarf an Kellerspeicher durch Call By Need ein.

Wir sehen also, daß der Aufwand für unsere Call By Need–Maßnahmen relativ unerheblich ist. Eventuell unnötige Updates sind daher unrelevant im Vergleich zu den bei den meisten Anwendungen erziehlten enormen Laufzeit– *und* Speicherplatz–Gewinnen.

Im folgenden dokumentieren wir die durch die Ersetzung der Auswertungsstrategie Call By Name durch die semantisch äquivalente Strategie Call By Need erzielten Verbesserungen an einigen Beispielen. Dazu geben wir zusätzlich die Anzahl der im AR–Keller angelegten AR's an, wobei sich diese Angaben etwa proportional zu der Laufzeit verhalten.

Beispiel 5.3 (Fortsetzung): Die Relevanz von Call By Need für die Fibonacci–Funktion wurde schon in der Fortsetzung von Beispiel 5.3 auf Seite 129 angedeutet. Hier nun die konkreten Werte für den Aufruf fib((I I I I I I I I I I I I I I)):

Optimierungsstufe	Max. Kellertiefe	Laufzeit	# AR's
Keine Optimierung	168	57,2s	14741
Definitonsgemäßer GDV	102	60,0s	14741
Einführung des GDV–Kellers	101	57,8s	14741
Dicke Parameter Opt. mittels GMARK	101	57,8s	14741
GDV–Verweis mittels GMARK	101	62,8s	14741
Parameter–Permutation mittels GPMARK	—	—	—
Effizienter Laufzeit–Keller	73	56,3s	14741
Auswertungs–Strategie Call By Need	73	9,3s	1505

⇒ ca. 56,5% Kellerspeicher– und ca. 83,7% Zeitersparnis.

Beispiel 9.10 Die bekannte Ackermann Funktion auf Listenbasis (vgl. Beispiel „ack.lsp“ im Anhang A.1):

```
ack = λ x y . { IF ATOM(x)
                   THEN CONS(I,y)
                   ELSE IF ATOM(y)
                          THEN ack(CDR(x),(I))
                          ELSE ack(CDR(x),ack(x,CDR(y)))
                FI FI }
```

Für den Aufruf ack((A B),(C D E)) ergeben sich folgende Werte:

Optimierungsstufe	Max. Kellertiefe	Laufzeit	# AR's
Keine Optimierung	211	24,3s	5348
Definitonsgemäßer GDV	211	24,5s	5348
Einführung des GDV–Kellers	189	24,1s	5348
Dicke Parameter Opt. mittels GMARK	189	24,1s	5348
GDV–Verweis mittels GMARK	189	26,8s	5348
Parameter–Permutation mittels GPMARK	—	—	—
Effizienter Laufzeit–Keller	137	24,2s	5348
Auswertungs–Strategie Call By Need	62	0,6s	106

⇒ ca. 70,6% Kellerspeicher– und 97,5% Zeitersparnis.

Beispiel 9.11 Die Funktion TAK von H. Takeuchi in einer Version auf Listenbasis (vgl. Beispiel „tak.lsp“ im Anhang A.1):

```
less = λ x y . { IF ATOM(x)
                    THEN IF ATOM(y) THEN F
                                    ELSE T
                         FI
                    ELSE IF ATOM(y) THEN F
                                    ELSE less(CDR(x),CDR(y))
                         FI
                    FI }

tak = λ x y z . { IF less(y,x)
                     THEN tak(tak(CDR(x),y,z),
                              tak(CDR(y),z,x),
                              tak(CDR(z),x,y),
                     ELSE z
                   FI }
```

Für den Aufruf tak((A B C D E),(A B C D),(A B)) ergeben sich folgende Werte:

Optimierungsstufe	Max. Kellertiefe	Laufzeit	# AR's
Keine Optimierung	278	291,2s	65369
Definitonsgemäßer GDV	278	293,0s	65369
Einführung des GDV–Kellers	257	286,7s	65369
Dicke Parameter Opt. mittels GMARK	257	286,7s	65369
GDV–Verweis mittels GMARK	257	317,0s	65369
Parameter–Permutation mittels GPMARK	—	—	—
Effizienter Laufzeit–Keller	191	285,2s	65369
Auswertungs–Strategie Call By Need	112	1,4s	222

⇒ ca. 59,7% Kellerspeicher– und ca. 99,5% Zeitersparnis.

Beispiel 9.12 Ein Fakultäts–Programm auf Listenbasis (vgl. Beispiel „fak.lsp“ im Anhang A.1):

```
mult = λ x y .
            { append = λ z q .
                          { IF ATOM(z)
                              THEN q
                              ELSE CONS(CAR(z),append(CDR(z),q))
                            FI }
```

```
                IF ATOM(x) THEN NIL
                           ELSE IF ATOM(CDR(x))
                                  THEN y
                                  ELSE append(y,mult(CDR(x),y))
                                FI
                FI }

fak = λ n . { IF ATOM(n) THEN (I)
                           ELSE mult(fak(CDR(n)),n)
                FI }
```

Für den Aufruf fak((I I I I I)) ergeben sich folgende Werte:

Optimierungsstufe	Max. Kellertiefe	Laufzeit	# AR's
Keine Optimierung	325	691,3s	168497
Definitonsgemäßer GDV	325	695,3s	168497
Einführung des GDV–Kellers	290	676,2s	168497
Dicke Parameter Opt. mittels GMARK	290	676,2s	168497
GDV–Verweis mittels GMARK	290	747,5s	168497
Parameter–Permutation mittels GPMARK	—	—	—
Effizienter Laufzeit–Keller	208	673,5s	168497
Auswertungs–Strategie Call By Need	158	2,6s	413

⇒ ca. 51,4% Kellerspeicher– und 99,6% Zeitersparnis.

Bemerkung 9.11 Die Angaben zur maximal erreichten Kellertiefe beziehen sich nur auf die AR–, GDV– und PP–Keller (vgl. Bemerkung 8.4). Darin nicht eingeschlossen ist der Bedarf an *Heap–Speicher*. Immer dann, und auch nur dann, wenn die Standardfunktion CONS (bzw. PCONS) ausgeführt wird, wird neuer Speicher im Heap belegt. Falls in Folge der Call By Need–Optimierung NSF–Aufrufe eingespart werden, welche CONS (bzw. PCONS) aufrufen, wird somit auch Speicherplatz im Heap und die zur Belegung des Heaps benötigte Zeit eingespart.

Kapitel 10

Zur Implementation in Pascal

In diesem Kapitel geben wir einen knappen Überblick über das nach Turbo–Pascal (Version 6.0) portierte und modifizierte neue LISP/N–System. Im Anhang 10.2 werden wir den kompletten Lauf eines Beispiel–Programms dokumentieren. Für Implementations–Details zu den in dieser Arbeit vorgestellten Optimierungs–Techniken sei auf die in den Anhängen B.1 und B.2 wiedergegebenen und kommentierten Pascal–Quelltexte von Laufzeitsystem und Compiler verwiesen.

10.1 Zur Implementation allgemein

Grundlage für die in dieser Arbeit vorgenommene Implementation ist das LISP/N–System, welches an der Universität Kiel auf einer Siemens 7.760 unter BS2000 in Siemens–Pascal (Version 3) implementiert wurde [Ho83]. Dabei lag das System als Version 3.1 vom 20.10.85 vor.

Für die Übernahme nach Turbo–Pascal 6.0 unter MS–DOS wurden einige Änderungen am Quellcode notwendig. Hier die wesentlichen:

1. Die einzelnen Laufzeit–Module wurden in die Datei STANDC.DAT integriert.

2. Der Type STRING ist in TP vordefiniert und wurde in STRIN umbenannt.

3. Für Integer–Zahlen wurde einheitlich LONGINT gewählt.

4. Das File–Handling wurde an den ASCII–Zeichensatz angepaßt, GET/PUT–Befehle durch READLN/WRITELN ersetzt und Dateien mit ASSIGN zugewiesen.

5. In TP sind nur lokale Goto–Sprünge erlaubt. Der Streusprung „goto 0“ mußte in das Zielprogramm integriert werden.

6. Zuweisungen von Zeichenketten (STRING) auf ein ARRAY OF CHAR ist nur bei gleicher Länge erlaubt.

7. Pointer werden mit „^“ anstatt „@“ angesprochen.

Dabei wurden die Veränderungen in Kleinschrift vorgenommen und sind dadurch von dem Orginal–Text leichter zu unterscheiden.

Grundsätzlich wurde nicht versucht, eine effiziente Implementation vorzunehmen. Um die Portabilität der Programme zu sichern, wurde weitestgehend auf TP–spezielle Befehle verzichtet. Lediglich zur Laufzeitbestimmung wird im Zielprogramm die TP–Unit DOS geladen und die Prozedur GETTIME angewendet.

Das neue LISP/N–System besteht im wesentlichen aus folgenden Dateien und Verzeichnissen:

1. ANACOMP.PAS: LISP/N–Compiler als TP–Programm. Er kann mit dem Befehl „TPC ANACOMP“ vom TP–Compiler in das lauffähige Programm ANACOMP.EXE übersetzt werden (siehe Bemerkung 10.1).

2. STANDC.DAT: Rumpf–Programm des Laufzeitsystems. Es enthält alle vom Zielcode benutzten Routinen und wird von ANACOMP.EXE gelesen.

3. LISPN <dir>: Unterverzeichnis mit LISP/N–Beispielprogrammen.

4. RUN.BAT: Batch–Datei. Sie enthält die zum Übersetzen und Ablaufen eines LISP/N–Programms notwendigen Befehle. Der Aufruf findet mit RUN <Pgm–Name> statt, wobei <Pgm–Name> ein LISP/N–Programm im aktuellen Verzeichnis oder im Unterverzeichnis LISPN ist.

5. ZPROG.PAS: Pascal–Programm als Ergebnis einer Übersetzung mit ANACOMP. Es kann mit „TPC ZPROG“ vom TP–Compiler in das lauffähige Zielprogramm ZPROG.EXE übersetzt werden (siehe Bemerkung 10.1).

6. LISPFILE.LSP: Die Standardeingabe von ANACOMP. Diese Datei muß das zu übersetzende LISP/N–Programm enthalten.

7. LISTFILE.DAT: Fehlerprotokoll der Übersetzung durch ANACOMP.

8. PASFIL1.DAT und PASFIL2.DAT: Teilergebnisse der Übersetzung durch ANACOMP.

9. ZWCODE.DAT: Zwischencode nach der ersten Compilerphase.

10. GMZWCODE.DAT: Die durch GPMARK markierte und ggf. permutierte Zwischencode–Datei ZWCODE.DAT.

11. KELLER.DAT: Kellerprotokoll–Datei als Option bei ZPROG.

Bemerkung 10.1 Der Turbo–Pascal Compiler TPC.EXE wird mit folgenden Optionen Aufgerufen: /$R+/$D-/$M65520,0,655360. Dadurch wird eine zusätzliche Bereichsprüfung zur Laufzeit durchgeführt, keine Debug–Informationen ins Zielprogramm übernommen und ein maximaler Speicher zur Verfügung gestellt.

Abbildung 10.1: Überblick über den LISP/N–Compiler

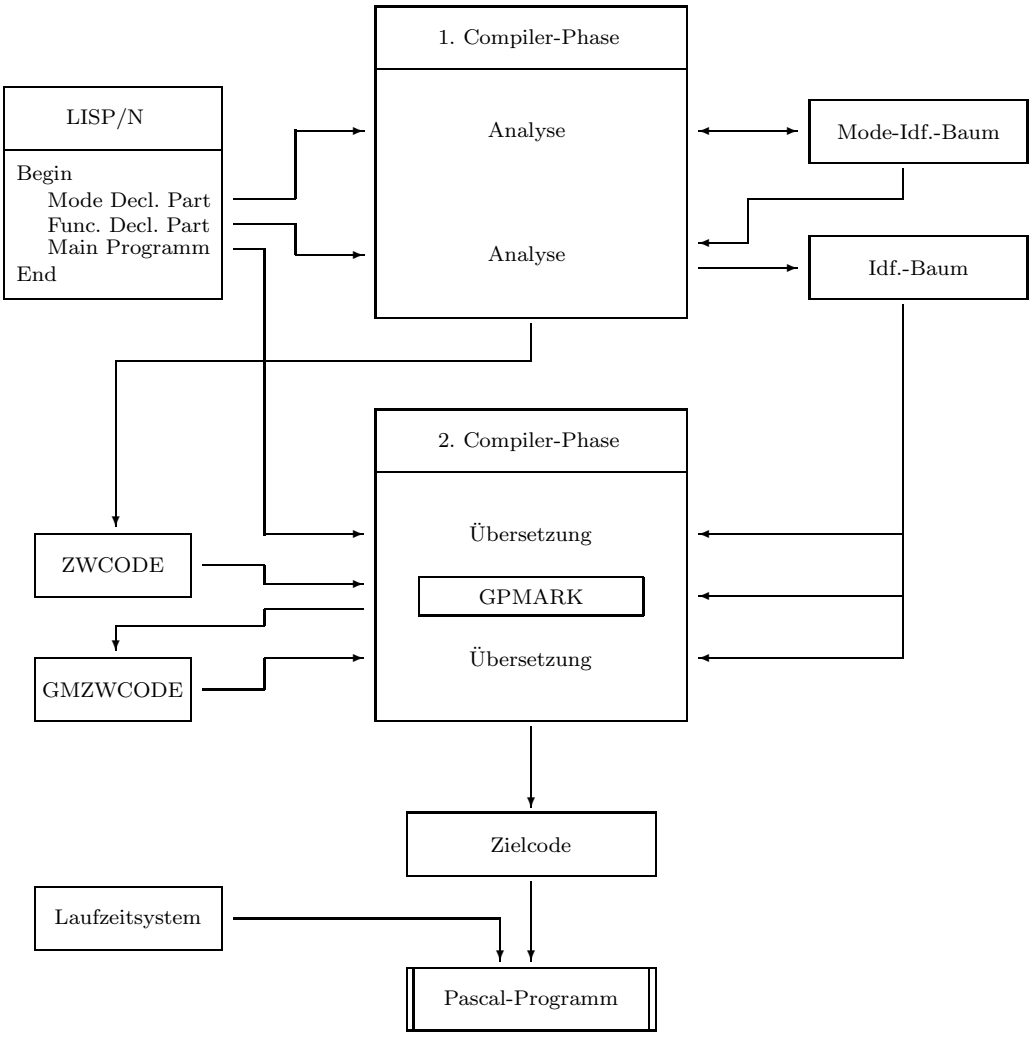
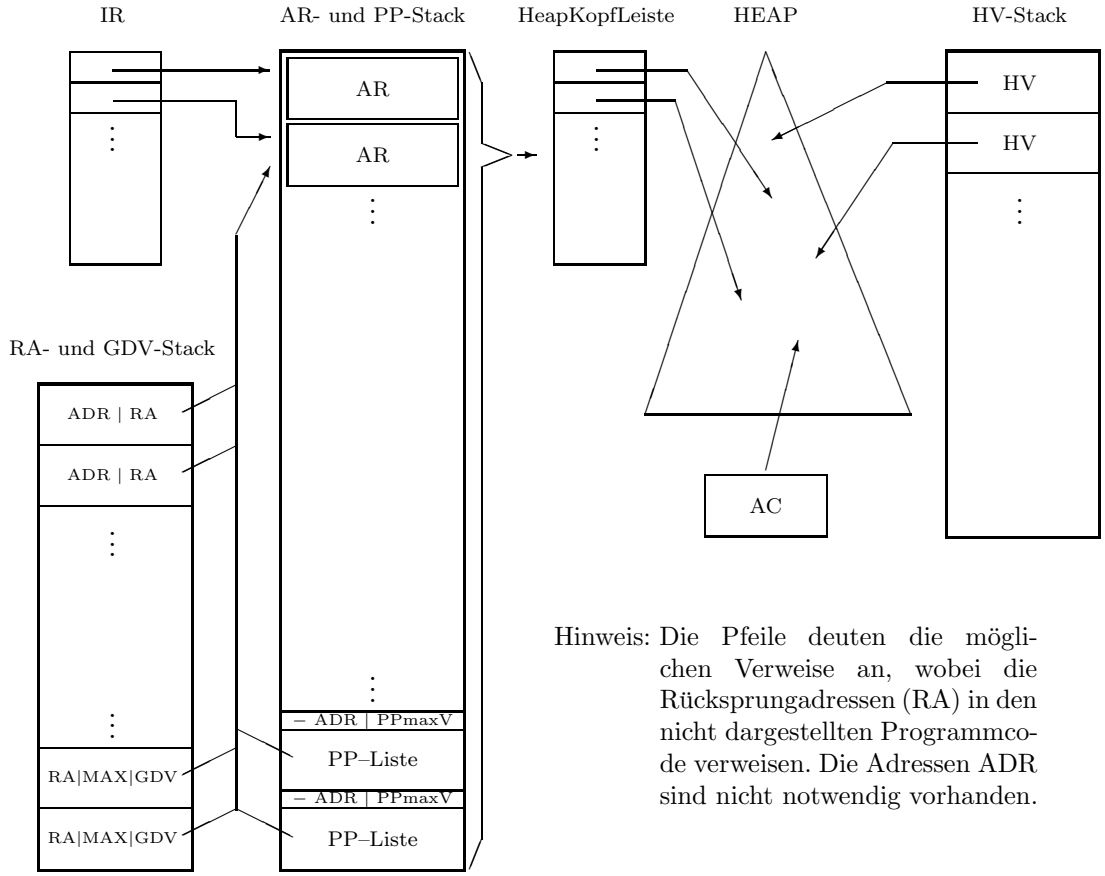


Abbildung 10.2: Speicherbereiche des implementierten Laufzeitsystems



Hinweis: Die Pfeile deuten die möglichen Verweise an, wobei die Rücksprungadressen (RA) in den nicht dargestellten Programmcode verweisen. Die Adressen ADR sind nicht notwendig vorhanden.

10.2 Dokumentation eines Beispiel–Laufs

Für das kurze LISP/N–Programm DEMO.LSP seien hier die einzelnen Dateien exemplarisch aufgeführt, die bei der Bearbeitung anfallen. Die Datei KELLER.DAT ist optional und dient zum Überprüfen der Kellerinhalte nach der Laufzeit.

LISP/N–Eingabeprogramm (LISPFILE.LSP)

```
begin
 mode tmode = func (smode1) smode1;
 mode emode = func () smode1;
 tmode:  twice (smode1: ff) smode1;
              { smode1: p (s-expr: x) s-expr; { ff(ff(x)) }; p };
 emode:  e () smode1; { cdr };
 smode1: h (s-expr: y) s-expr; { cons(cons(y,y),(D)) };
 smode1: g (s-expr: x) s-expr; { cons(h(twice(e())((A B C))),x) };
 g((E))
end
```

Zwischencode nach dem 1. Lauf (ZWCODE.DAT)

```
**FUNC** TWICE    ;        {        **FUNC** P        ;        {
FF       (        FF       (        X        )        )        }
P        }        **FUNC** E        ;        {        CDR      }
**FUNC** H        ;        {        CONS     (        CONS     (
Y        ,        Y        )        ,        (        D        )
)        }        **FUNC** G        ;        {        CONS     (
H        (        TWICE    (        E        (        )        )
(        (        A        B        C        )        )        )
,        X        )        }        G        **MAIN**
(        (        E        )        )
```

Zwischencode nach der Markierung (GMZWCODE.DAT)

```
**FUNC** TWICE    ;        {        **FUNC** P        ;        {
FF       (        (*102*)  FF       (        X        )        )
}        P        }        **FUNC** E        ;        {        CDR
}        **FUNC** H        ;        {        CONS     (        CONS
(        Y        (*1*)    ,        Y        (*0*)    )        ,
(        D        )        )        }        **FUNC** G        ;
{        PCONS_   (        X        (*0*)    ,        H        (
(*100*)  TWICE    (        (*100*)  E        (        )        )
(        (        A        B        C        )        )        )
)        }        **MAIN** G        (        (        E        )
)
```

Zielcode (ZPROG.PAS ohne Laufzeitsystem) Der Inhalt der beiden Dateien PASFIL1.DAT und PASFIL2.DAT mit Kommentaren versehen:

```
INHEAP('(E       )       ',3);      Liste (E) → Heap
INHEAP('(D       )       ',4);      Liste (D) → Heap
INHEAP('(A       B       C       )       ',5);  Liste (A B C) → Heap
AC:=B[3];                           AC:=(E)
LZSPARB(0,1,3);                     Parameter (E) anlegen
LZSARK(0,22,23,0,'R',0); goto 0;    NSF–Aufruf g((E))
23:                                 RA von g((E))
OUT;                                Ausgabe vom AC als Endergebnis: ((((C) C) D) E)
GOTO 2;                             Programm–Ende
19:                                 Code der NSF p:
LZSPARB(2,2,25);                    dicker Para.: ff(x)
GOTO 24;                            Sprung zur Code–Fortsetzung von p
25:                                 Code der Hilfsfkt. zu ff(x):
LZSPARB(3,2,1);                     form. Idf. x als akt. Parameter
LZSFORMIDF(1,1,TYP,ADR);            Typ und ggf. Adr. des form. Idf. ff
IF TYP=0 THEN FEHLER(13);           S–Ausdruck als Funk.–Idf.
if need then T1update:=true;        Call By Need: Typ–1 Update für ff vorbereiten
LZSARK(typ,1,26,1,'R',0); goto 0;   form. Aufruf ff(x)
```

```
26:                                  RA von ff(x)
LZSFEND; goto 0;                     Funktionsende der Hilfsfkt. zu ff(x)
24:                                  Fortsetzung des Codes von p:
LZSFORMIDF(1,1,TYP,ADR);             Typ und ggf. Adr. des form. Idf. ff
IF TYP=0 THEN FEHLER(13);            S–Ausdruck als Funk.–Idf.
if need then T1update:=true;         Call By Need: Typ–1 Update für ff vorbereiten
LZSARK(typ,1,27,1,'R',0); goto 0;    form. Aufruf ff(ff(x))
27:                                  RA von ff(ff(x))
LZSFEND; goto 0;                     Funktionsende von p
18:                                  Code der NSF twice:
LZSARK(0,19,28,1,'R',0); goto 0;     Aufruf der NSF p ohne akt. Para.
28:                                  RA von p
LZSFEND; goto 0;                     Funktionsende von twice
20:                                  Code der NSF e:
LZSARK(0,5,29,0,'R',0); goto 0;      Aufruf der SF CDR ohne akt. Para.
29:                                  RA von CDR
LZSFEND; goto 0;                     Funktionsende von e
21:                                  Code der NSF h:
rlacall:=false;                      Anfang linker Ast
LZSFORMIDF(1,1,TYP,ADR);             Typ und ggf. Adr. des form. Idf. y
IF TYP=0 THEN AC:=B[ADR]             Wert von y ist S–Ausdruck
ELSE BEGIN                           Wert von y vom Typ–1 oder Typ–2
LZSARK(typ,1,30,1,'L',1); goto 0;    linker Ast–NSF–Aufruf durch y
30:                                  RA des Aufrufs durch y
LZSLEFTEND(30);                      linker Ast–Aufruf durch y beendet
if need then T0update;               Call By Need: Typ–0 Update für y
END;
PUSH;                                inneres CONS: Zwischenergebnis in HV–Keller
LZSFORMIDF(1,1,TYP,ADR);             Typ und ggf. Adr. des form. Idf. y
IF TYP=0 THEN AC:=B[ADR]             Wert von y ist S–Ausdruck
ELSE BEGIN                           Wert von y vom Typ–1 oder Typ–2
LZSARK(typ,1,31,1,'L',0); goto 0;    linker Ast–NSF–Aufruf durch y
31:                                  RA des Aufrufs durch y
LZSLEFTEND(31);                      linker Ast–Aufruf durch y beendet
if need then T0update;               Call By Need: Typ–0 Update für y
END;
CONS;                                inneres CONS ausführen
rlacall:=true;                       Rückkehr in äußeren linken Ast
PUSH;                                äußeres CONS: Zwischenergebnis in HV–Keller
AC:=B[4];                            AC:=(D)
CONS;                                äußeres CONS ausführen
LZSFEND; goto 0;                     Funktionsende von h
22:                                  Code der NSF g:
rlacall:=false;                      Anfang linker Ast (von PCONS)
LZSFORMIDF(1,1,TYP,ADR);             Typ und ggf. Adr. des form. Idf. x
IF TYP=0 THEN AC:=B[ADR]             Wert von x ist S–Ausdruck
ELSE BEGIN                           Wert von x vom Typ–1 oder Typ–2
LZSARK(typ,1,32,1,'L',0); goto 0;    linker Ast–NSF–Aufruf durch x
```

```
32:                                  RA des Aufrufs durch x
LZSLEFTEND(32);                      linker Ast–Aufruf durch x beendet
if need then T0update;               Call By Need: Typ–0 Update für x
END;                                 Auswertung von x beendet
rlacall:=true;                       Ende linker Ast (von PCONS)
PUSH;                                Ergebnis des linken Astes in HV–Keller
LZSPARB(2,0,34);                     dicker Para.: twice(e())((A B C))
GOTO 33;                             Sprung zur Code–Fortsetzung von g
34:                                  Hilfs–Code für dicken Para. twice(e())((A B C)):
LZSPARB(2,0,36);                     dicker Para.: e()
GOTO 35;                             Sprung zur Hilfscode–Fortsetzung von twice
36:                                  Hilfs–Code für dicken Para. e():
LZSTRENN;                            leere Parameterliste
LZSARK(0,20,37,0,'R',0); goto 0;     NSF–Aufruf e()
37:                                  RA von e()
LZSFEND; goto 0;                     Funktionsende von Hilfsfkt. zu e()
35:                                  Hilfscode–Fortsetzung von twice(e())((A B C)):
LZSTRENN;                            Trennmarke vor PP–Liste (A B C)
AC:=B[5];                            AC:=(A B C)
LZSPARB(0,2,5);                      Parameter (A B C) anlegen
LZSARK(0,18,38,0,'R',0); goto 0;     NSF–Aufruf twice(e())((A B C))
38:                                  RA von twice(e())((A B C))
LZSFEND; goto 0;                     Funktionsende von Hilfsfkt. zu twice
33:                                  Code–Fortsetzung von g:
LZSARK(0,21,39,0,'R',0); goto 0;     NSF–Aufruf h(twice(e())((A B C)))
39:                                  RA von h(twice(e())((A B C)))
pcons;                               PCONS ausführen
LZSFEND; goto 0;                     Funktionsende von g
```

Laufzeitkeller–Protokoll (KELLER.DAT) Als Option beim Programmlauf kann diese Datei ausgegeben werden. Sie enthält bei jedem NSF–Aufruf den aktuellen Inhalt des AR–, PP– und GDV–Kellers und den Inhalt der Indexregister (SV–Kette). Bei den Einträgen im GDV–Keller werden aus Platzgründen die zusätzlichen Einträge von RA und MaxGDV nicht dargestellt. In dem Beispiel kommt kein Typ–4 Parameter mit Referenz–Verweis vor. Die Linkage ist bei jedem AR beschriftet: SN = statisches Niveau, SV = statischer Vorgänger und BFS = Beginn freier Speicher. Der implementierte Keller hat eine Länge von 1000 Zellen und enthält am Ende den auf den AR–Keller zuwachsenden PP–Keller. Der angegebene Kellerablauf ist der Ablauf des permutierten Programms DEMO.LSP unter Nutzung der Auswertungs–Strategie Call By Need:

```
         1 [SN]  :     -1          Aufruf vom Hauptprogramm
         2 [SV]  :     -1
         3 [BFS] :      4

         4 [SN]  :      0          g((E))
         5 [SV]  :      1
         6 [BFS] :      8
         7       :  30010          (E)

      1000       :     -1          ADR=0 / PPmaxV=1

  1.) SV-Kette   : 1 4
      GDV-Keller : 1
---------------------------------

         1 [SN]  :     -1
         2 [SV]  :     -1
         3 [BFS] :      4

         4 [SN]  :      0          h(Hilfsfkt.())
         5 [SV]  :      1
         6 [BFS] :      8
         7       : 340012          Hilfsfkt. zu twice(e())((A B C))

      1000       :     -1

  2.) SV-Kette   : 1 4
      GDV-Keller : 1
---------------------------------

         1 [SN]  :     -1
         2 [SV]  :     -1
         3 [BFS] :      4

         4 [SN]  :      0
         5 [SV]  :      1
         6 [BFS] :      8
         7       : 340012

         8 [SN]  :      0          Hilfsfkt. zu twice(e())((A B C))
         9 [SV]  :      1
        10 [BFS] :     11

      1000       :     -1

  3.) SV-Kette   : 1 8
      GDV-Keller : 1 4
---------------------------------
```

```
          1 [SN]  :     -1
          2 [SV]  :     -1
          3 [BFS] :      4

          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0          twice(Hilfsfkt.())((A B C))
          9 [SV]  :      1
         10 [BFS] :     12
         11       : 360012          Hilfsfkt. zu e()

        998       :     -1
        999       :  50020          (A B C)
       1000       :     -1

  4.) SV-Kette   : 1 8
      GDV-Keller : 1 4
---------------------------------

          1 [SN]  :     -1
          2 [SV]  :     -1
          3 [BFS] :      4

          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0
          9 [SV]  :      1
         10 [BFS] :     12
         11       : 360012

         12 [SN]  :      1          p((A B C))
         13 [SV]  :      8
         14 [BFS] :     16
         15       :  50020          (A B C)

       1000       :     -1

  5.) SV-Kette   : 1 8 12
      GDV-Keller : 1 4
---------------------------------
```

```
          1 [SN]  :     -1
          2 [SV]  :     -1
          3 [BFS] :      4

          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0
          9 [SV]  :      1
         10 [BFS] :     12
         11       : 360012

         12 [SN]  :      1
         13 [SV]  :      8
         14 [BFS] :     16
         15       :  50020

         16 [SN]  :      0          Hilfsfkt. zu e()(ff(x))
         17 [SV]  :      1
         18 [BFS] :     19

        998       : -11012          ADR=11 / PPmaxV=12
        999       : 250122          Hilfsfkt. zu ff(x)
       1000       :     -1

  6.) SV-Kette   : 1 16
      GDV-Keller : 1 4
---------------------------------

          1 [SN]  :     -1
          2 [SV]  :     -1
          3 [BFS] :      4

          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0
          9 [SV]  :      1
         10 [BFS] :     12
         11       : 360012

         12 [SN]  :      1
         13 [SV]  :      8
         14 [BFS] :     16
```

```
        15       :  50020

        16 [SN]  :      0          e()(ff(x))
        17 [SV]  :      1
        18 [BFS] :     19

       998       : -11012
       999       : 250122
      1000       :     -1

  7.) SV-Kette   : 1 16
      GDV-Keller : 1 4
---------------------------------

         1 [SN]  :     -1
         2 [SV]  :     -1
         3 [BFS] :      4

         4 [SN]  :      0
         5 [SV]  :      1
         6 [BFS] :      8
         7       : 340012

         8 [SN]  :      0
         9 [SV]  :      1
        10 [BFS] :     12
        11       :  50011          ← Typ–1 Update

        12 [SN]  :      1
        13 [SV]  :      8
        14 [BFS] :     16
        15       :  50020

        16 [SN]  :      0          CDR(Hilfsfkt.())
        17 [SV]  :      1
        18 [BFS] :     20
        19       : 250122          Hilfsfkt. zu ff(x)

      1000       :     -1

  8.) SV-Kette   : 1 16
      GDV-Keller : 1 4
---------------------------------

         1 [SN]  :     -1
         2 [SV]  :     -1
         3 [BFS] :      4
```

```
          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0
          9 [SV]  :      1
         10 [BFS] :     12
         11       :  50011

         12 [SN]  :      1
         13 [SV]  :      8
         14 [BFS] :     16
         15       :  50020

         16 [SN]  :      2          Hilfsfkt. zu ff(x)
         17 [SV]  :     12
         18 [BFS] :     19

       1000       :     -1

  9.) SV-Kette   : 1 8 12 16
      GDV-Keller : 1 4
-----------------------------------

          1 [SN]  :     -1
          2 [SV]  :     -1
          3 [BFS] :      4

          4 [SN]  :      0
          5 [SV]  :      1
          6 [BFS] :      8
          7       : 340012

          8 [SN]  :      0          CDR((A B C))
          9 [SV]  :      1
         10 [BFS] :     12
         11       :  50020          (A B C)

       1000       :     -1

 10.) SV-Kette   : 1 8
      GDV-Keller : 1 4
-----------------------------------
```

Anschließend erfolgt an der Adresse 7 ein Typ–0 Update und das Programm kann ohne weitere NSF–Aufrufe fortgesetzt werden. Es endet schließlich mit Ausgabe der Liste ((((C) C) D) E).

Ausblick

In dieser Arbeit haben wir ein verbessertes Compiler–System für die applikative Programmiersprache LISP/N und einem kellerartigem Laufzeitsystem mit statischer Variablen–Bindung vorgstellt. Durch statische Analysetechniken kann nun der GDV–Verweise geziehlt gesetzt werden, ein notwendiges statisches Niveau für dicke Parameter–Hilfsfunktionen ermittelt werden, sowie über Parameter–Permutationen entschieden werden. Dabei haben diese Maßnahmen gemeinsam, daß zwar oft Speicherplatz eingespart werden kann, aber Laufzeitgewinne prinzipiell nicht eintreten können. Durch den neuen PP–Keller und durch die auf drei Einträge reduzierte und zu einem späteren Zeitpunkt geschriebene Linkage wird dann neben Speicherplatz auch Laufzeit eingespart.

Einen bemerkenswerten Anteil an den erziehlten Verbesserungen hat dabei unser Vorschlag zu einer Call By Need–Realisierung. Obwohl noch Situationen „konstruiert“ werden können, in denen dicke Parameter weiterhin mehrmals ausgewertet werden müssen (also weiterhin gemäß Call By Name verfahren wird), werden die wesentlichen, in der Praxis relevanten Aufruf–Situationen, bereits jetzt erfaßt. In Beispiel 9.7 haben wir eine mögliche Situation beschrieben, durch die ein dicker Parameter noch mehrmals ausgewertet werden kann: Die Duplizierung von formalen Identifikatoren auf aktueller Parameterposition, wobei gleichzeitig das Kriterium NK1 nicht erfüllt ist. Die in Bemerkung 9.6 angedeutete Möglichkeit dieses Problem zu lösen scheint relativ aussichtsreich.

Die in Bemerkung 9.3 angedeutete Alternative schließt die Möglichkeit mit ein, daß eine Verschlechterung im Speicherplatzbedarf entstehen könnte. Mögliche Verschlechterungen durch „Optimierungen“ haben wir in dieser Arbeit prinzipiell versucht zu vermeiden. Dennoch bleibt zu untersuchen, inwieweit mögliche Verschlechterungen in Kauf genommen werden können, wenn dafür die meisten praxis–relevanten Situationen optimiert werden. Diese Frage stellt sich ebenfalls bei den Permutations–Kriterien: Häufig wird gerade durch PK2 eine Permutation verhindert, die dann aber dynamisch zu einer Optimierung geführt hätte.

Eine weitere offene Frage ist, wie sich unsere Call By Need–Realisierung im Vergleich zu Call By Value verhält. Ein vollständiges Call By Need wäre vermutlich mindestens so effizient wie Call By Value und würde zusätzlich die gewünschte λ–Semantik gewährleisten.

In LISP/N werden bereits die fünf (mit PCONS sechs) Standardfunktionen gemäß Call By Value gehandhabt. Würden auch NSF–Aufrufe gemäß Call By Value bearbeitet, so würden zum einen wesentlich häufiger GDV–Verweise notwendig werden, zum anderen jedoch könnten dann Parameter–Permutationen für *jeden* mehrstelligen Aufruf geprüft werden. Die vorgestellte GDV–Optimierung und die Optimierung durch Parameter–Permutation würden an Bedeutung gewinnen.

Für einen Leistungs–Vergleich mit am Markt gebräuchlichen LISP–Systemen müßte eine effiziente Implementation des LISP/N–Sytstems vorgenommen werden, wobei der Sprachumfang stark zu erweitern wäre und das Laufzeitsystem sicherlich in Assembler formuliert werden müßte. In diesem Rahmen wäre auch nach einer möglichst effizienten Realisierung des Heaps zu suchen. Für Von Neumann–Rechnerarchitekturen bietet das vorgestellte LISP/N–System eine gute Grundlage für eine konkurenzfähige Implementation.

Interessant wird es, wenn die Übertragung auf andere Hardware–Konzepte geprüft wird: Insbesondere sind dabei die sich zur Zeit stark verbreitenden *parallelen* Rechner–Architekturen zu nennen. Es ist z.B. zu überlegen, ob bei der Auswertung von mehrstelligen SF–Aufrufen bei Call By Name/Need, bzw. bei der Auswertung *jedes* Aufrufs bei Call By Value, die einzelnen Argumente der Aufrufe *gleichzeitig* bearbeitet werden können...

Es sei an dieser Stelle auch auf die Dissertation von Hundehege [Hu92] verwiesen, in der ähnliche und zum Teil wesentlich weiterreichende Konzepte für eine algebraische Spezifikationssprache und einer abstrackten Stack–Maschine vorgestellt werden.

Mit diesem Ausblick wollen wir schließen.

Literaturverzeichnis

[**Be88**] Beckmann, R.: Ein LISP–Interpretierer mit LCC–Optimierung, Implementation und Benchmark Tests; Diplomarbeit; Westfälische Wilhelms–Universität Münster, 1988.

[**Fe86**] Felgentreu, K.-U.: Ein optimierter Static Scope–Interpretierer auf der Basis ALGOL–artiger Laufzeitkeller; Tagungsband der Jahrestagung der Gesellschaft für Informatik; Informatik Fachberichte, Band 126, S. 165−179, Oktober 1986.

[**Fe87**] Felgentreu, K.-U.: Optimierung von Funktionsaufrufen bei Shallow Binding mit statischer Variablenbindung; Dissertation; Westfälische Wilhelms–Universität Münster, 1987.

[**Fe/Li86**] Felgentreu, K.-U.; Lippe, W.-M.: Dynamic Optimization of Covered Tail Recursive Function Calls in Applicative Languages; Proceedings of the ACM Computer Science Conference, S. 293−299, 1986.

[**Ga85**] Gabriel, R.P.: Perfomance and Evaluation of LISP–Systems; Computer Science Series, MIT Press, Cambridge Massachusetts, 1985.

[**Gr/Hi/La67**] Grau, A.A.; Hill, U.; Langmaack, H.: Translation of ALGOL60; Handbook for Automatic Computation; Vol. I, Part b, Springer–Verlag 1967.

[**Ho83**] Honschopp, U.: Implementierung der applikativen Programmiersprache LISP/N unter Berücksichtigung eines speziellen Laufzeitsystems; Diplomarbeit; Institut für Informatik und Praktische Mathematik der Universität Kiel, 1983.

[**Ho/Li/Si83**] Honschopp, U.; Lippe, W.-M.; Simon, F.: Compiling Functional Languages for von Neumann Machines; SIGPLAN Notices, Vol. 18, No. 6, 1983.

[**Hu92**] Hundehege, J.-B.: Effiziente Compilation funktionaler Module einer algebraischen Spezifikationssprache; Dissertation; Westfälische Wilhelms–Universität Münster; erscheint 1992.

[**Ki85**] Kindler, M.: LISP/N Benutzerhandbuch; Institut für Informatik und Praktische Mathematik der Universität Kiel, 1985.

[**Ki87**] Kindler, M.: Korrektheit eines LISP/N–Compilers; Diplomarbeit; Institut für Informatik und Praktische Mathematik der Universität Kiel, 1987.

[**La73**] Langmaack, H.: Übersetzerkonstruktion; Scriptum einer Vorlesung an der Universität des Saarlandes im WS 72/73 und SS 73; Angefertigt von K. Mich und V. Müller.

[**Li/Si79**] Lippe, W.-M.; Simon, F.: LISP/N – Basic Definitions and Properties; Bericht 4/79; Institut für Informatik und Praktische Mathematik der Universität Kiel; 1979.

[**Sch92**] Schumacher, J.: Die Steigerung der Anzahl von skgi–Aufrufen in PureLISP–Programmen durch statische Programmanalyse (?); Diplomarbeit; Westfälische Wilhelms–Universität Münster, erscheint 1992 (?).

Anhang A

Beispiele und ihre Optimierungen

Wir geben in Abschnitt A.1 einige (zum Teil in der Arbeit verwendete) Beispiel–LISP/N–Programme an. In Abschnitt A.2 sind dann die dazu festgestellten Optimierungen für die jeweiligen Optimierungs–Stufen tabellarisch wiedergegeben.

A.1 LISP/N–Beispielprogramme

ack.lsp (vgl. Beispiel 9.10)

```
BEGIN
 MODE S1=func(S-EXPR,S-EXPR)S-EXPR;
 S1 : ACK(S-EXPR:X,S-EXPR:Y)S-EXPR;
      {IF ATOM(X) THEN CONS("I,Y)
          ELSE IF ATOM(Y) THEN ACK(CDR(X),(I))
                  ELSE ACK(CDR(X),ACK(X,CDR(Y)))
               FI
       FI };
 ACK(IN,IN)
END
```

aperm1.lsp (vgl. Beispiel 7.3)

```
begin
 smode2: e (s-expr: v, s-expr: w) s-expr; { v };
 smode1: d (s-expr: z) s-expr;
          { smode1: h (s-expr: y) s-expr;
                     { smode1: g (s-expr: x) s-expr;
                                { cons(y,e(x,"D)) };
                       g("B) };
            h("A) };
 d("C)
end
```

aperm3.lsp (vgl. Beispiel 7.4)

```
begin
 mode smode10 = func (s-expr,s-expr,s-expr,s-expr,s-expr,
                      s-expr,s-expr,s-expr,s-expr,s-expr) s-expr;
 smode1: e (s-expr: z) s-expr;
          { smode10:
              d (s-expr:a,s-expr:b,s-expr:c,s-expr:d,s-expr:e,
                 s-expr:g,s-expr:h,s-expr:i,s-expr:j,s-expr:k) s-expr;
               { cons(a,cons(b,cons(c,cons(d,cons(e,
                 cons(g,cons(h,cons(i,cons(j,k)))))))) };
            if eq(z,"B) then d("A,"B,"C,"D,"E,"F,"G,"H,"I,"J)
                        else "K
            fi };
 smode1: h (s-expr: x) s-expr;
          { smode1: g (s-expr: y) s-expr;
                     { cons(e(x),e(y)) };
            g("B) };
 h("A)
end
```

append.lsp

```
begin
 smode2: append (s-expr: x, s-expr: y) s-expr;
          { smode1: null (s-expr: z) s-expr;
                     { if atom(z) then eq(z,"nil)
                                  else f
                       fi };
            if null(x) then y
                       else cons(car(x),append(cdr(x),y))
            fi };
 append(in,in)
end
```

aufg34.lsp (vgl. Beispiel 6.7)

```
begin
 mode ma = func (smode1) s-expr;
   smode2:
     l1 (s-expr: x, s-expr: y) s-expr;
      { ma:
          l2 (smode1: c) s-expr;
           { smode1:
               l3 (s-expr: y) s-expr;
                 { c(y) };
               cons(l3("A),cons(c(x),cons(x,y))) };
        smode1:
          h (s-expr: x) s-expr;
           { smode1:
               l4 (s-expr: h) s-expr; { cons(h,"B) };
             smode1:
               l5 (s-expr: y) s-expr; { cons(h("C),cons(h("D),cons(x,y))) };

             if atom(x) then l4(y)
                        else l5("E) fi };
```

```
        l2(h) };
   l1(cons("F,"G),"H)
end
```

aufg39.lsp (vgl. Beispiel 7.7)

```
begin
  smode1: p (s-expr: x) s-expr; { cons ( g("B) , c(x) ) };
  smode1: g (s-expr: x) s-expr; { h(x) };
  smode1: h (s-expr: x) s-expr; { cons ( k("D) , x ) };
  smode1: c (s-expr: x) s-expr; { cons ( k("E) , x ) };
  smode1: k (s-expr: x) s-expr; { x };
  p("A)
end
```

aufg40.lsp

```
begin
 smode2:
  p (s-expr: x, s-expr: y) s-expr;
   { smode1: c (s-expr: x) s-expr;
              { cons(g(x,"A),y) };
     smode2: g (s-expr: x, s-expr: y) s-expr;
              { if atom(x) then cons(x,y)
                           else c(car(x)) fi };
     smode1: h (s-expr: x) s-expr;
              { g(car(x),"B) };
     cons(h(x),g(x,"C)) };
 p( ((D.E).F) , "G )
end
```

dckop.lsp (vgl. Beispiel 6.3)

```
begin
 smode1: g (s-expr: x) s-expr;
          { if atom(x) then x
                       else g(car((A B)))
            fi };

 g((X Y))
end
```

dckop1.lsp (vgl. Beispiel 6.4)

```
begin
 smode1: e (s-expr: x) s-expr;
   { smode1: g (s-expr: y) s-expr;
              { if atom(x) then x
                           else cons(e(y),e(cdr(x)))
                fi };
     g("A) };
 e(in)
end
```

demo.lsp Siehe Abschnitt 10.2.

equal.lsp

```
BEGIN
 MODE M1=FUNC(S-EXPR,S-EXPR)S-EXPR;
 M1 : EQUAL(S-EXPR:X,S-EXPR:Y)S-EXPR;
      {IF ATOM(X) THEN IF ATOM(Y) THEN EQ(X,Y) ELSE F FI ELSE
       IF ATOM(Y) THEN F ELSE IF EQUAL(CAR(X),CAR(Y)) THEN EQUAL(CDR(X),CDR(Y))
        ELSE F FI FI FI };
 EQUAL(IN,IN)
END
```

evalq.lsp (aus der Vorlesung „Funktionale und Logische Programmierung“ von Prof. Dr. W.-M. Lippe im WS87 in Münster)

```
begin
 mode smode3 = func (s-expr, s-expr, s-expr) s-expr;
 smode2: evalq (s-expr: fn, s-expr: x) s-expr;
     { apply(fn,x,"nil) };
 smode3: apply (s-expr: fn, s-expr: x, s-expr: a) s-expr;
     { if atom(fn) then if eq(fn,"CAR)  then car(car(x)) else
                        if eq(fn,"CDR)  then cdr(car(x)) else
                        if eq(fn,"ATOM) then atom(car(x)) else
                        if eq(fn,"CONS) then cons(car(x),car(cdr(x))) else
                        if eq(fn,"EQ)   then eq(car(x),car(cdr(x))) else
                        apply(cdr(assoc(fn,a)),x,a) fi fi fi fi fi else
       if eq(car(fn),"LAMBDA) then eval(car(cdr(cdr(fn))),
                                        pairlis(car(cdr(fn)),x,a)) else
       if eq(car(fn),"LABEL) then apply(car(cdr(cdr(fn))),x,
                                        cons(cons(car(cdr(fn)),
                                                  car(cdr(cdr(fn)))),a)) else
       if eq(car(fn),"FUNARG) then apply(car(cdr(fn)),x,car(cdr(cdr(fn)))) else
       apply(eval(fn,a),x,a) fi fi fi fi };
 smode2: eval (s-expr: e, s-expr: a) s-expr;
     { if null(e) then "nil else
       if atom(e) then if eq(e,T) then T else
                      if eq(e,F) then F else
                      cdr(assoc(e,a)) fi fi else
       if atom(car(e)) then if eq(car(e),"QUOTE) then car(cdr(e)) else
                            if eq(car(e),"COND)  then evcon(cdr(e),a) else
                            if eq(car(e),"FUNCTION)
                              then list("FUNARG,car(cdr(e)),a) else
                            apply(car(e),evlis(cdr(e),a),a) fi fi fi else
       apply(car(e),evlis(cdr(e),a),a) fi fi fi };
 smode2: evcon (s-expr: l, s-expr: a) s-expr;
     { if eval(car(car(l)),a) then eval(car(cdr(car(l))),a) else
       evcon(cdr(l),a) fi };
 smode2: evlis (s-expr: m, s-expr: a) s-expr;
     { if null(m) then "nil else
       cons(eval(car(m),a),evlis(cdr(m),a)) fi };
 smode3: list (s-expr: x, s-expr: y, s-expr: z) s-expr;
     { cons(x,cons(y,cons(z,"nil))) };
 smode2: assoc (s-expr: x, s-expr: a) s-expr;
     { if eq(car(car(a)),x) then car(a) else
       assoc(x,cdr(a)) fi };
 smode3: pairlis (s-expr: x, s-expr: y, s-expr: a) s-expr;
```

```
    { if null(x) then a else
      cons(cons(car(x),car(y)),pairlis(cdr(x),cdr(y),a)) fi };
 smode1: null (s-expr: x) s-expr;
    { if atom(x) then if eq(x,"nil) then T else
                      F fi else
      F fi };
 evalq(in,in)
end
```

fak.lsp (vgl. Beispiel 9.12)

```
begin
 smode2: mult (s-expr: x, s-expr: y) s-expr;
          { smode2: append (s-expr: z, s-expr: q) s-expr;
                    { if atom(z) then q
                                 else cons(car(z),append(cdr(z),q))
                      fi };
            if atom(x) then "nil
                       else if atom(cdr(x)) then y
                                            else append(y,mult(cdr(x),y))
            fi              fi };
 smode1: fak (s-expr: n) s-expr;
          { if atom(n) then (I)
                       else mult(fak(cdr(n)),n)
            fi };
 fak(in)
end
```

fib.lsp (vgl. Beispiel 5.3)

```
begin
 smode1:
   fib (s-expr: x) s-expr;
     { if atom(x) then x else
         if atom(cdr(x)) then x
           else cons(fib(cdr(x)),fib(cdr(cdr(x))))
       fi fi };
 fib(in)
end
```

fibpp.lsp (vgl. Beispiel 8.1)

```
begin
 mode smode10 = func (s-expr,s-expr,s-expr,s-expr,s-expr,
                      s-expr,s-expr,s-expr,s-expr,s-expr) s-expr;
 mode funmode = func (s-expr) smode10;
 smode1:
   fib (s-expr: x) s-expr;
     { if atom(x) then x else
         if atom(cdr(x)) then x
           else cons(fib(cdr(x)),fib(cdr(cdr(x))))
       fi fi };
 smode10:
   fun (s-expr:a,s-expr:b,s-expr:c,s-expr:d,s-expr:e,
        s-expr:g,s-expr:h,s-expr:i,s-expr:j,s-expr:k) s-expr;
     { cons(a,cons(b,cons(c,cons(d,cons(e,
```

```
      cons(g,cons(h,cons(i,cons(j,k)))))))) };
 funmode:
   h (s-expr: x) smode10;
     { if atom(fib(x)) then fun
                       else fun
      fi };
 h(in)("A,"B,"C,"D,"E,"F,"G,"H,"I,"J)
end
```

gdv1.lsp (vgl. Beispiel 6.1)

```
begin
 smode1: g (s-expr: x) s-expr; { x };
 smode1: e (s-expr: y) s-expr; { cons(g(y),"B) };
 e("A)
end
```

gdv2.lsp

```
begin
 smode1: g (s-expr: x) s-expr; { x };
 smode1: e (s-expr: y) s-expr; { cons(g(y),"B) };
 e(e(e(e(e("A)))))
end
```

gdvk.lsp (vgl. Beispiel 6.5)

```
begin
 smode2: p (s-expr: x, s-expr: y) s-expr;
           { smode1: h (s-expr: v) s-expr;
                     { cons(g(v),y) };
             smode1: g (s-expr: w) s-expr;
                     { if atom(w) then h(cons("D,"E))
                                  else cons(x,y)
                       fi };
             g(x) };
 p("A,p((B),"C))
end
```

gdvt4.lsp

```
begin
  smode2: e (s-expr: v, s-expr: w) s-expr; { cons(v,w) };
  smode2: h (s-expr: x, s-expr: y) s-expr;
           { smode1: i (s-expr: d) s-expr;
              { smode1: g (s-expr: z) s-expr;
                        { e(cons(x,y),z) };
                g("C) };
             i("D) };
  h(car((A B)),car((B C)))
end
```

handbuch.lsp Siehe LISP/N–Benutzerhandbuch [Ki85].

hanoi.lsp

```
begin
 mode smode4 = func (s-expr,s-expr,s-expr,s-expr) s-expr;
 smode4:
   hanoi1 (s-expr: start, s-expr: ziel, s-expr: hilf, s-expr: hoehe) s-expr;
     { if atom(hoehe) then cons(start,ziel)
       else cons(hanoi1(start,hilf,ziel,cdr(hoehe)),
                 cons(cons(start,ziel),hanoi1(hilf,ziel,start,cdr(hoehe))))
       fi };
 smode1:
   hanoi (s-expr: thoehe) s-expr;
     { hanoi1("A,"B,"C,cdr(thoehe)) };
 hanoi(in)
end
```

lsto.lsp (vgl. Beispiel 6.6)

```
begin
 smode1:
   lsto (s-expr: tree) s-expr;
     { if atom(tree) then tree
                     else cons(lsto(car(tree)),"nil)
       fi };
 lsto(in)
end
```

member.lsp

```
BEGIN
 MODE M1=FUNC(S-EXPR,S-EXPR)S-EXPR;
 M1 : MEMBER(S-EXPR:X,S-EXPR:Y)S-EXPR;
      {IF ATOM(Y) THEN F ELSE IF EQUAL(X,CAR(Y) ) THEN T ELSE MEMBER(X,CDR(Y))
       FI FI };
 M1 : EQUAL(S-EXPR:Y,S-EXPR:X)S-EXPR;
      {IF ATOM(Y) THEN IF ATOM(X) THEN EQ(X,Y) ELSE F FI ELSE
    IF ATOM(X) THEN F ELSE IF EQUAL(CAR(Y),CAR(X)) THEN EQUAL(CDR(Y),CDR(X))
        ELSE F FI FI FI };
 MEMBER(IN,IN)
END
```

need2c.lsp (vgl. Beispiel 9.5)

```
begin
 smode1: g (s-expr: x) s-expr;
          { smode1: e (s-expr: y) s-expr;
                     { smode1: h (s-expr: z) s-expr; { cons(z,z) };
                       cons(h(y),y) };
            cons(e(x),x) };
 g(car((a b)))
end
```

perm.lsp (vgl. Beispiel 7.1)

```
begin
 smode1: g (s-expr: x) s-expr; { x };
 smode1: h (s-expr: y) s-expr;
```

```
          { smode1: e (s-expr: z) s-expr; { cons(g("C),g(z)) };
            e("B) };
 h("A)
end
```

perm1.lsp (vgl. Beispiel 7.6)

```
begin
 smode1: g (s-expr: x) s-expr;
          { smode1: h (s-expr: y) s-expr;
                     { if atom(x) then x
                                  else cons(cons(g(cdr(x)),y),x)
                       fi };
            h(x) };
 g(in)
end
```

perm2.lsp (vgl. Beispiel 7.8)

```
begin
 smode1: g (s-expr: x) s-expr;
          { smode1: h (s-expr: y) s-expr;
                     { if atom(x) then x
                                  else cons(cons(g(cdr(x)),g(cdr(y))),x)
                       fi };
            h(x) };
 g(in)
end
```

perm3.lsp

```
begin
 smode1: g (s-expr: x) s-expr; { cons(x,x) };
 smode1: h (s-expr: y) s-expr;
          { smode1: e (s-expr: z) s-expr; { cons(g("C),g(z)) };
            e("B) };
 h("A)
end
```

reverse.lsp (vgl. Beispiel 5.2)

```
BEGIN
 SMODE1 : REVERSE(S-EXPR : X)S-EXPR;
          { IF ATOM(X) THEN X
                       ELSE CONS(REVERSE(CDR(X)),REVERSE(CAR(X))) FI };
 REVERSE(IN)
END
```

tak.lsp (vgl. Beispiel 9.11)

```
begin
 mode smode3 = func (s-expr,s-expr,s-expr) s-expr;
 smode2: less (s-expr: x, s-expr: y) s-expr;
          { if atom(x) then if atom(y) then F
                                       else T
                            fi
                       else if atom(y) then F
```

```
                                        else less(cdr(x),cdr(y))
                              fi
            fi };
 smode3: tak (s-expr: x, s-expr: y, s-expr: z) s-expr;
          { if less(y,x) then tak(tak(cdr(x),y,z),
                                  tak(cdr(y),z,x),
                                  tak(cdr(z),x,y))
                         else z
            fi };
tak(in,in,in)
end
```

A.2 Die Optimierungen

In diesem Abschnitt dokumentieren wir die Relevanz der einzelnen Optimierungsstufen dieser Arbeit. Hinweise zu der Tabelle finden sich unten.

Erläuterungen zu den Tabellen A.1 und A.2:

- Die Abkürzungen zu den Optimierungsstufen bedeuten:
  - KE : Keine Optimierung; Laufzeitsystem wie in [Ho83] (siehe Kapitel 2).
  - DG : Definitionsgemäßer GDV–Verweis (siehe Abschnitt 6.1).
  - GK : Eigener GDV–Keller (siehe Abschnitt 6.3).
  - DP : Dicke Parameter Optimierung mittels GMARK (siehe Abschnitt 6.2).
  - GM: Der neue GDV–Verweise mittels GMARK (siehe Abschnitt 6.1).
  - PE : Mögliche Parameter–Permutationen mittels GPMARK (siehe Kapitel 7).
  - EK : Ein effizienter Laufzeitkeller (eigener PP–Keller und günstigere HOpt; siehe Kapitel 8).
  - NE : Auswertungs–Strategie Call By Need (siehe Kapitel 9).
- Die Angaben in den Spalten zu den einzelnen Optimierungsstufen bauen von links nach rechts aufeinander auf.
- Die Angaben zur maximalen Kellertiefe beziehen sich in den Spalten „KE“ und „DG“ nur auf den AR–Keller. Ab der Spalte „GK“ werden die Einträge vom neuen GDV–Keller und zusätzlich ab der Spalte „EK“ die Einträge vom neuen PP–Keller berücksichtigt.
- Falls keine Permutationen durchgeführt werden, so steht in der Spalte „PE“ lediglich ein „—“.
- Die Laufzeiten sind auf einem IBM–XT–Kompatibelem mit 8086 CPU (ohne Coprozessor) und einer Taktfrequenz von 9,87 MHz gemessen.

Tabelle A.1: Die maximalen Kellertiefen und einige Laufzeiten

Programm mit Eingabe(n)	Maximale Keller–Tiefe								Einsp.
Optimierungsstufe:	KE	DG	GK	DP	GM	PE	EK	NE	in %
`ack((A B),(C D E))`	211	211	189	189	189	—	137	62	70,6
Laufzeit in Sekunden:	*24,3*	*24,5*	*24,1*	*24,1*	*26,8*	—	*24,2*	*0,6*	*97,5*
`aperm1`	26	26	23	23	23	—	17	17	34,6
`aperm3`	28	28	26	26	26	—	22	22	21,4
`append((A B C),(A B C))`	50	50	46	46	46	—	34	34	32,0
`aufg34`	48	48	44	44	38	—	28	28	41,7
`aufg39`	26	26	25	25	25	11	9	9	65,4
`aufg40`	57	57	53	53	41	—	31	31	45,6
`dckop`	25	25	23	17	17	—	13	13	48,0
`dckop1((A B C D E F G H))`	137	137	119	72	71	—	49	49	64,2
`demo`	64	44	41	41	41	34	23	23	64,1
`equal((A B C),(A B C))`	49	43	40	40	40	—	30	30	38,8
`evalq((LAMBDA (X Y) (CONS (CAR X) Y)),((A B)(C D)))`	103	103	97	97	97	—	73	73	29,1
`fak((I I I I I))`	325	325	290	290	290	—	208	158	51,4
Laufzeit in Sekunden:	*691,3*	*695,3*	*676,2*	*676,2*	*747,5*	—	*673,5*	*2,7*	*99,6*
`fib((I I I I I I I I I I I I))`	168	102	101	101	101	—	73	73	56,5
Laufzeit in Sekunden:	*57,2*	*60,0*	*57,8*	*57,8*	*62,8*	—	*56,3*	*9,4*	*83,6*
`fibpp((I I I I I I I I I I I I))`	449	273	272	272	254	—	84	84	81,3
Laufzeit in Sekunden:	*173,8*	*184,2*	*180,9*	*180,9*	*185,9*	—	*56,3*	*9,4*	*94,6*
`gdv1`	19	19	18	18	11	—	9	9	52,6
`gdv2`	47	47	46	46	11	—	9	9	80,9
`gdvk`	42	42	38	32	26	—	20	20	52,4
`gdvt4`	47	47	43	31	26	—	20	20	57,4
`handbuch(TWICE,(((A))((B))((C))))`	124	124	112	112	108	—	80	65	47,6
`hanoi((I I I I I I I I I))`	172	118	116	116	116	—	92	92	46,5
Laufzeit in Sekunden:	*44,5*	*45,9*	*44,6*	*44,6*	*48,3*	—	*43,1*	*16,9*	*62,0*
`lsto(((((((A)B)C)D)E)F))`	90	60	59	59	53	—	37	37	58,9
`member((A B),(C D (A B)))`	71	59	55	55	55	—	41	41	42,3
`need2c`	32	32	31	31	31	—	23	23	28,0
`perm`	26	26	24	24	24	17	13	13	50,0
`perm1((A B C D E F G H))`	179	137	127	127	127	79	57	57	68,2
`perm2((A B C D E F G H))`	179	137	127	127	127	127	89	89	50,3
Laufzeit in Sekunden:	*32,8*	*34,3*	*33,3*	*33,3*	*35,7*	*35,7*	*32,1*	*10,0*	*69,5*
`perm3`	26	26	24	24	24	17	13	13	50,0
`reverse((A B C D E F G H))`	116	74	73	73	73	—	53	53	54,3
`tak((A B C D E),(A B C D),(A B))`	278	278	257	257	257	—	191	112	59,7
Laufzeit in Sekunden:	*291,2*	*293,0*	*286,7*	*286,7*	*317,0*	—	*285,2*	*1,4*	*99,5*

Tabelle A.2: Die Anzahl der angelegten AR's

Programm mit Eingabe(n)	Anzahl angelegter AR's		Einsp.
Optimierungsstufe:	KE ... EK	NE	in %
`ack((A B),(C D E))`	5348	106	98,0
`aperm1`	4	4	0,0
`aperm3`	5	5	0,0
`append((A B C),(A B C))`	20	11	45,0
`aufg34`	14	12	14,3
`aufg39`	6	6	0,0
`aufg40`	20	14	30,0
`dckop`	4	3	25,0
`dckop1((A B C D E F G H))`	78	42	46,2
`demo`	22	10	54,5
`equal((A B C),(A B C))`	49	19	61,2
`evalq((LAMBDA (X Y) (CONS (CAR X) Y)),((A B)(C D)))`	491	50	89,8
`fak((I I I I I))`	168497	413	99,8
`fib((I I I I I I I I I I I I I))`	14741	1505	89,8
`fibpp((I I I I I I I I I I I I I))`	14743	1507	89,8
`gdv1`	2	2	0,0
`gdv2`	14	14	0,0
`gdvk`	11	8	27,3
`gdvt4`	7	7	0,0
`handbuch(TWICE,(((A))((B))((C))))`	386	53	86,3
`hanoi((I I I I I I I I I I))`	10241	2047	80,0
`lsto(((((((A)B)C)D)E)F))`	34	13	61,8
`member((A B),(C D (A B)))`	62	23	62,9
`need2c`	7	4	42,9
`perm`	4	4	0,0
`perm1((A B C D E F G H))`	118	33	72,0
`perm2((A B C D E F G H))`	8194	1786	78,2
`perm3`	4	4	0,0
`reverse((A B C D E F G H))`	133	33	75,2
`tak((A B C D E),(A B C D),(A B))`	65369	222	99,7

Anhang B

Die Pascal–Quelltexte

B.1 Das Laufzeitsystem (standc.dat)

```
 1
 2  (* Laufzeitsystem zum LISP/N-Compiler ANACOMP                              *)
 3  (*                                                                        *)
 4  (* Version 3.1 , Uwe Honschopp (siehe [Ho83])            Stand: 10-21-85  *)
 5  (*                                                                        *)
 6  (* TP-Version 1.65 , Guido Wessendorf                    Stand: 07-24-92  *)
 7  (*                                                                        *)
 8  (* Testversion: Es koennen verschiedene Optimierungsstufen ausprobiert und  *)
 9  (*              der Laufzeitkellerinhalt protokolliert werden.            *)
10  (*                                                                        *)
11
12
13  CONST
14       BLANKS    ='        ';          (* ZUR VORBESETUNG MIT BLANKS         *)
15       HEAPMAX   = 250      ;          (* LAENGE DER HEAP-KOPFLEISTE         *)
16       HVSTMAX   =1000      ;          (* MAX. EINTRAEGE IM HV-STACK         *)
17       IRMAX     =  20      ;          (* ANZAHL DER INDEXREGISTER           *)
18       LONGSTRMAX= 255      ;          (* MAX. LAENGE VON LISP/N-KONSTANTEN  *)
19       MAXLENGTH =   8      ;          (* MAX. LAENGE VON ATOMAREN S-AUSDR.  *)
20       RAgdvSTMAX=1000      ;          (* MAX. EINTRAEGE IM RA- und gdv-STACK*)
21       STACKMAX  =1000      ;          (* MAX. EINTRAEGE IM STACK            *)
22
23
24  TYPE
25       POATOMEL  =^ATOMEL;                      (*ZEIGER AUF ATOM            *)
26       POHEAPEL  =^HEAPEL;                      (*ZEIGER AUF HEAPELEMENT     *)
27       ATOMSTR   =ARRAY[1..MAXLENGTH] OF CHAR;  (*ATOMARER S-AUSDRUCK        *)
28       ATOMEL    =RECORD                        (*ATOMSPEICHERELEMENT        *)
29                    NAME : ATOMSTR;             (*ATOMNAME                   *)
30                    R,L  : POATOMEL             (*ZEIGER A. NACHF. ATOMELEM. *)
31                   END;
32       HEADREC   =RECORD
33                    AP   : POATOMEL;            (*ZEIGER IN DEN ATOMSPEICHER *)
34                    KP   : POHEAPEL             (*ZEIGER IN DEN KNOTENSPEICHER*)
35                   END;
36       HEAPEL    =RECORD                        (*KNOTENSPEICHERELEMENT      *)
37                    LA,RA: POATOMEL;            (*ZEIGER IN DEN ATOMSPEICHER *)
38                    LK,RK: POHEAPEL             (*ZEIGER IN DEN KNOTENSPEICHER*)
39                   END;
40       LONGSTR   =ARRAY[1..LONGSTRMAX] OF CHAR;  (* FUER LISP/N-KONSTANTANTEN *)
41
42
43  VAR
44       A       : ARRAY[1..STACKMAX] OF longint  ; (* STACK                   *)
45       AC      : HEADREC                        ; (* 'ACCUMULATOR '          *)
46       ADR     : longint                        ; (* ADR. BEI FORM.PARAMETERN *)
47       ARP     : longint                        ; (* ANFANG DES JEW. AR      *)
48       ATOMWP  : POATOMEL                       ; (*ZEIGER A. ATOMSPEICHERWURZ.*)
49       ATOP    : longint                        ; (* ERSTER FREIER PLATZ IN A *)
50       B       : ARRAY[0..HEAPMAX+stackmax] OF HEADREC
51                                                ; (* HEAPKOPFLEISTE          *)
```

```
52        GDV     : longint                         ; (* NAECHTER NEUER GDV        *)
53        GOTOVAR : longint                         ; (* ENTHAELT MARKE BEI SPRUNG *)
54        HVSTACK : ARRAY[1..HVSTMAX] OF HEADREC    ; (* HILFSVARIABLEN - STACK    *)
55        HVSTTOP : longint                         ; (* 1.FREIER PLATZ IN HVSTACK *)
56        IR      : ARRAY[-1..IRMAX] OF longint     ; (* INDEXREGISTER             *)
57        LETDIG  : SET OF CHAR                     ; (* BUCHSTABEN UND ZIFFERN    *)
58        LETTER  : SET OF CHAR                     ; (* BUCHSTABEN                *)
59        RAgdvST : ARRAY[1..RAgdvSTMAX] OF longint; (* RA- und gdv-Stack         *)
60        RASTTOP : longint                         ; (* ERSTER FREIER PL.IN RAST  *)
61        STATNIV : 0..IRMAX                        ; (* JEW. STATISCHES NIVEAU    *)
62        TYP     : longint                         ; (* TYP EINES FORM.PARAMETER  *)
63
64        boffs   : longint;           (* Offset: Heapkopfleiste --> AR-Keller    *)
65        gdvsttop: longint;           (* Erster freier Platz im GDV-Stack        *)
66        pptop   : longint;           (* PP-Kellerspitze                         *)
67        bpp     : longint;           (* Globale Var. ersetzt BPP-Linkage-Eintrag*)
68        rlacall : boolean;           (* Ist RLA-NSF-Aufruf aktiv?               *)
69        T1update: boolean;           (* Call By Need: Typ-1 Ergebnis updaten?   *)
70
71        kell    : text;              (* Fuer die Keller-Protokoll-Datei         *)
72        maxdeep : longint;           (* Fuer Statistik: Maximale Kellertiefe    *)
73        aktdeep : longint;           (* Fuer Statistik: Aktuelle Kellertiefe    *)
74        kellsum : longint;           (* Fuer Statistik: Summe Kellereintraege   *)
75        anzar   : longint;           (* Fuer Statistik: Anzahl der AR's         *)
76        gdvstmax: longint;           (* Fuer Statistik: Maximale GDV-Kellertiefe*)
77        opt     : char;              (* Optimierungs-Stufen: 'K','D' oder 'G'   *)
78        proto   : boolean;           (* Kellerprotokoll schreiben?              *)
79        need    : boolean;           (* Call By Need unterstuetzen?             *)
80
81        h1,m1,s1,hs1: word;          (* Fuer Laufzeitbestimmung (TP): Startzeit *)
82        h2,m2,s2,hs2: word;          (* Fuer Laufzeitbestimmung (TP): Endzeit   *)
83
84
85    (* MODULE DES LAUFZEITSYSTEMS                                              *)
86
87
88    PROCEDURE FEHLER ( N : LONGINT );
89
90    (* DIE LZS-ROUTINE FEHLER WIRD BEI AUFTRETEN VON LAUFZEITFEHLERN AUFGERUFEN. *)
91    (* ES ERFOLGT DIE AUSGABE EINER FEHLERMELDUNG UND ANSCHLIESSEND DER ABBRUCH  *)
92    (* DER PROGRAMMAUSFUEHRUNG.                                                 *)
93
94    VAR
95              B        :      BOOLEAN ;       (* STEUERVARIABLE FUER AUSSTIEG-CMD*)
96
97    BEGIN
98
99     WRITELN;
100    WRITELN('WAEHREND DER PROGRAMMAUSFUEHRUNG IST EIN FEHLER AUFGETRETEN:  ');
101    WRITELN;
102    WRITE(' @@ ',N:3,' :  ');
103
104    CASE N OF
105
106     1  : WRITELN('INHALT VON AC<>T  BZW. <>F IN FUNKTION BOOLTEST!            ');
107     2  : WRITELN('CAR WURDE AUF ATOMAREN S-AUSDRUCK ANGEWANDT!               ');
108     3  : WRITELN('CDR WURDE AUF ATOMAREN S-AUSDRUCK ANGEWANDT!               ');
109     4  : WRITELN('ZWEITES ARGUMENT IN EQ NICHT ATOMAR!                       ');
110     5  : WRITELN('ERSTES ARGUMENT IN EQ NICHT ATOMAR!                        ');
111     7  : WRITELN('UNZULAESSIGES ZEICHEN IN ATOMAREM S-AUSDRUCK!              ');
112     8  : WRITELN('ATOMARER S-AUSDRUCK ZU LANG - MAX.',MAXLENGTH:3,'ZEICHEN!  ');
113     9  : WRITELN('ZUVIELE KONSTANTEN - MAXIMAL Z.ZT.',HEAPMAX:3,'ZULAESSIG!  ');
114    10  : WRITELN('KONSTANTE ZU LANG  - MAXIMAL Z.ZT.',LONGSTRMAX:3,'ZEICHEN! ');
115    11  : WRITELN('EINGABE DES S-AUSDRUCKES NICHT VOLLSTAENDIG!              ');
116    12  : writeln('")" bei punktierter Liste fehlt!                          ');
117    13  : writeln('S-Expr auf formaler Funktionsposition!                    ');
118    14  : WRITELN('UEBERLAUF IM RUECKKEHRADRESSEN (und GDV) - KELLER!        ');
119    15  : WRITELN('UEBERLAUF IM LAUFZEITKELLER!                              ');
120    16  : WRITELN('UEBERLAUF IM HILFSVARIABLENKELLER!                        ');
121    17  : writeln('Ueberlauf im GDV (und RA) - Keller!                       ');
122    18  : writeln('Ueberlauf der Heapkopfleiste!                             ');
123    ELSE  WRITELN(' FEHLERNUMMER: ',N:3,' NICHT GEFUNDEN!                    ');
124
125    END;
126
```

```
127    writeln;
128    WRITELN('DAS PROGRAMM IST NICHT AUSFUEHRBAR. ES FOLGT PROGRAMMABBRUCH!');
129
130    (* AN DIESER STELLE KANN DUMP-ROUTINE EINGEFUEHRT WERDEN.                  *)
131
132    halt                                (* TP-Pascal: Ausstieg aus dem Programm *)
133
134   END;   (* OF PROCEDURE FEHLER  *)
135
136
137   PROCEDURE ATOM;
138
139   (* LZS-ROUTINE ZUR AUSFUEHRUNG DER LISP/N-STANDDARDFUNKTION ATOM .          *)
140   (* DER S-AUSDRUCK AUF DEN ATOM ANGEWENDET WERDEN SOLL WIRD IN AC GEHALTEN.  *)
141   (* DAS ERGEBNIS WIRD IN AC ABGELIEFERT.                                    *)
142
143   BEGIN
144
145    IF (AC.AP<>NIL)AND(AC.KP=NIL) THEN  AC:=B[1]              (* B[1]=^TRUE    *)
146                                  ELSE  AC:=B[2]              (* B[2]=^FALSE   *)
147
148   END;       (* OF PROCEDURE ATOM   *)
149
150
151   PROCEDURE CAR;
152
153   (* LZS-ROUTINE ZUR AUSFUEHRUNG DER LISP/N-STANDARDFUNKTION CAR .            *)
154   (* DER S-AUSDRUCK AUF DEN CAR ANGEWANDT WERDEN SOLL WIRD IN AC GEHALTEN.    *)
155   (* DAS ERGEBNIS WIRD IN AC ABGELIEFERT.                                    *)
156
157   BEGIN
158
159    IF AC.AP<>NIL THEN FEHLER(2);      (* ATOMARES ARGUMENT:CAR NICHT DEFINIERT *)
160    IF AC.KP^.LA<>NIL THEN             (* ERGEBNIS IST ATOMAR                   *)
161                       BEGIN
162                        AC.AP:=AC.KP^.LA;
163                        AC.KP:=NIL;
164                       END
165                      ELSE             (* ERGEBNIS IST NICHT ATOMAR             *)
166                       BEGIN
167                        AC.KP:=AC.KP^.LK;
168                        AC.AP:=NIL;
169                       END
170
171   END;           (* OF PROCEDURE CAR  *)
172
173
174   PROCEDURE CDR;
175
176   (* LZS-ROUTINE ZUR AUSFUEHRUNG DER LISP/N-STANDARDFUNKTION CDR.             *)
177   (* DER S-AUSDRUCK AUF DEN CDR ANGEWANDT WERDEN SOLL WIRD IN AC GEHALTEN.    *)
178   (* DAS ERGEBNIS WIRD IN AC ABGELIEFERT.                                    *)
179
180   BEGIN
181
182    IF AC.AP<>NIL THEN FEHLER(3);      (* ATOMARES ARGUMENT: CDR NICHT DEFINIERT*)
183    IF AC.KP^.RA<>NIL THEN             (* ERGEBNIS IST ATOMAR                   *)
184                       BEGIN
185                        AC.AP:=AC.KP^.RA;
186                        AC.KP:=NIL;
187                       END
188                      ELSE             (* ERGEBNIS SIT NICHT ATOMAR             *)
189                       BEGIN
190                        AC.KP:=AC.KP^.RK;
191                        AC.AP:=NIL;
192                       END
193
194   END;         (* OF PROCEDURE CDR  *)
195
196
197   PROCEDURE CONS;
198
199   (* LZS-ROUTINE ZUR AUSF^UHRUNG DER LISP/N-STANDARDFUNKTION CONS.            *)
200   (* DAS ERSTE ARGUMENT WIRD IN POP(HVSTACK) GEHALTEN;DAS ZWEITE IN AC.       *)
201   (* DAS ERGEBNIS WIRD IN AC ABGELIEFERT.                                    *)
```

```
202
203  VAR
204                  AC1    :  HEADREC;        (* HV -UM ERSTES  ARGUMENT ZU HALTEN*)
205                  AC2    :  HEADREC;        (* HV -UM ZWEITES ARGUMENT ZU HALTEN*)
206
207      PROCEDURE INITEL(VAR P : POHEAPEL);
208      (* DIE PROZEDUR LEGT EINEN NEUEN KNOTEN P IM HEAP AN.                    *)
209
210      BEGIN
211
212       NEW(P);                        (* GENERIERUNG EINES NEUEN KNOTEN IM HEAP*)
213       P^.LA:=NIL;                    (* VORBESETZUNG DES KNOTENS              *)
214       P^.RA:=NIL;
215       P^.LK:=NIL;
216       P^.RK:=NIL;
217
218      END;    (* OF PROCEDURE INITEL*)
219
220
221  BEGIN
222
223   AC2:=AC;                                  (* LADEN DES ZWEITEN ARGUMENTES   *)
224   HVSTTOP:=HVSTTOP-1;                       (* ac1:=pop; pop vom HV-Stack     *)
225   AC1:=HVSTACK[HVSTTOP];                    (* LADEN DES ERSTEN  ARGUMENTES   *)
226   AC.AP:=NIL;
227   INITEL(AC.KP);                            (* ERZEUGEN EINES NEUEN KNOTEN    *)
228   IF AC1.AP<>NIL THEN AC.KP^.LA:=AC1.AP     (* CAR-TEIL EINTRAGEN             *)
229                  ELSE AC.KP^.LK:=AC1.KP;
230   IF AC2.AP<>NIL THEN AC.KP^.RA:=AC2.AP     (* CDR-TEIL EINTRAGEN             *)
231                  ELSE AC.KP^.RK:=AC2.KP
232
233  END;    (* OF PROCEDURE CONS *)
234
235
236  procedure pcons;
237
238  (* LZS-Routine zur Ausfuehrung der LISP/N-Standardfunktion PCONS.            *)
239  (* Abarbeitung vom permutierten CONS: Das erste Argument steht im AC und das *)
240  (* zweite Argument an der Spitze vom HV-Stack. Ansonsten identisch zu CONS.  *)
241
242  var
243                  ac1    :  headrec;        (* hv -um erstes  Argument zu halten*)
244                  ac2    :  headrec;        (* hv -um zweites Argument zu halten*)
245
246      procedure initel(var p : poheapel);
247      (* Die Prozedur legt einen neuen Knoten p im Heap an.                    *)
248
249      begin
250
251       new(p);                        (* Generierung eines neuen Knoten im Heap*)
252       p^.la:=nil;                    (* Vorbesetzung des Knotens              *)
253       p^.ra:=nil;
254       p^.lk:=nil;
255       p^.rk:=nil;
256
257      end;    (* of procedure initel *)
258
259  begin
260
261   ac1:=ac;                                  (* Laden des zweiten Argumentes   *)
262   hvsttop:=hvsttop-1;                       (* ac2:=pop; pop vom HV-Stack     *)
263   ac2:=hvstack[hvsttop];                    (* Laden des ersten  Argumentes   *)
264   ac.ap:=nil;
265   initel(ac.kp);                            (* Erzeugen eines neuen Knoten    *)
266   if ac1.ap<>nil then ac.kp^.la:=ac1.ap     (* car-teil eintragen             *)
267                  else ac.kp^.lk:=ac1.kp;
268   if ac2.ap<>nil then ac.kp^.ra:=ac2.ap     (* cdr-teil eintragen             *)
269                  else ac.kp^.rk:=ac2.kp
270
271  end;    (* of procedure pcons *)
272
273
274  PROCEDURE EQ;
275
276  (* LZS-ROUTINE ZUR AUSFUEHRUNG DER LISP/N-STANDARDFUNKTION EQ.               *)
```

```
277  (* DAS ERSTE ARGUMENT WIRD IN POP(HVSTACK) GEHALTEN; DAS ZWEITE IN AC.       *)
278  (* DAS ERGEBNIS WIRD IN AC ABGELIEFERT.                                      *)
279
280  VAR
281                  AC1   : HEADREC;         (* HV - UM ERSTES ARGUMENT ZU HALTEN*)
282
283  (**    DIE FUNKTION POP HOLT DEN OBERSTEN EINTRAG AUS DEM HILFSVARIABLENSTACK.
284                                                                            **)
285
286  BEGIN
287
288   IF AC.AP=NIL THEN FEHLER(4);   (*ZWEITES ARGUMENT NICHT ATOMAR:EQ NICHT DEF.*)
289   HVSTTOP:=HVSTTOP-1;            (* ac1:=pop; pop vom HV-Stack                *)
290   AC1:=HVSTACK[HVSTTOP];         (* LADEN DES ERSTEN ARGUMENTES               *)
291   IF AC1.AP=NIL THEN FEHLER(5);  (*ERSTES ARGUMENT NICHT ATOMAR :EQ NICHT DEF.*)
292   IF AC1.AP=AC.AP THEN AC:=B[1]  (* B[1] =^ TRUE                              *)
293                   ELSE AC:=B[2]  (* B[2] =^ FALSE                             *)
294
295  END;        (* OF PROCEDURE EQ *)
296
297
298  PROCEDURE INHEAP(st: string; BTOP: LONGINT);
299
300  (* DIE LZS-ROUTINE LEGT VOM UEBERSETZER ODER DER ROUTINE INP ANGELIEFERTE    *)
301  (* LISP/N-KONSTANTEN IM HEAP AB.DIE KONSTANTE WIRD IM PARAMETER K UEBERGEBEN.*)
302  (* DER PARAMETER BTOP ENTHAELT DIE ZUGEHOERIGE ADRESSE DER HEAPKOPFLEISTE.   *)
303
304  VAR
305          ATOMSYMBOL :  ATOMSTR;         (*HAELT NAME EINES ATOMES             *)
306          J,N        :  LONGINT;         (*LAUFINDEX FUER JEWEILIGE KONSTANTE K*)
307          PA         :  POATOMEL;        (*ZEIGER IN DEN ATOMSPEICHER          *)
308          PK         :  POHEAPEL;        (*ZEIGER AUF NAECHSTEN KNOTEN         *)
309          k          :  longstr;         (* Array zur Aufnahme von ST...       *)
310
311
312
313      PROCEDURE NEXTJ;
314      (* DIE PROZEDUR SETZT DEN ZEIGER J AUF DAS NAECHSTE ZEICHEN VON K UND    *)
315      (* TESTET DABEI,OB DIE MAX. LAENGE FUER KONSTANTEN UEBERSCHRITTEN WIRD.  *)
316
317      BEGIN
318
319       IF J<LONGSTRMAX THEN J:=J+1
320                       ELSE FEHLER(10)
321
322      END;    (* OF PROCEDURE NEXT *)
323
324
325      PROCEDURE READATOM(VAR ATOMSYM:ATOMSTR);
326      (* DIE PROZEDUR LIESST EIN ATOM                                          *)
327
328       VAR
329              I  :  LONGINT;                                  (* ZAEHLER      *)
330
331       BEGIN
332
333        FOR I:=1 TO MAXLENGTH DO ATOMSYM[I]:=' ';
334        IF (K[J] IN LETTER) THEN ATOMSYM[1]:=K[J]
335                            ELSE FEHLER(7);
336        NEXTJ;
337        I:=2;
338        WHILE (K[J] IN LETDIG) DO
339                                BEGIN
340                                 IF I>MAXLENGTH THEN FEHLER(8);
341                                 ATOMSYM[I]:=K[J];
342                                 NEXTJ;
343                                 I:=I+1
344                                END;
345        J:=J-1
346
347       END;     (* OF PROCEDURE READATOM  *)
348
349       PROCEDURE INITEL(VAR P : POHEAPEL);
350       (* DIE PROZEDUR GENERIERT EINEN NEUEN KNOTEN IM HEAP .                   *)
351       BEGIN
```

```
352
353         NEW(P);
354         P^.LA:=NIL;                              (* VORBESETZUNG MIT PASCAL-NIL *)
355         P^.RA:=NIL;
356         P^.LK:=NIL;
357         P^.RK:=NIL
358
359       END;    (* OF PROCEDURE INITEL*)
360
361
362       PROCEDURE INATOMTREE(ATOMSYM:ATOMSTR;AP:POATOMEL;VAR PA:POATOMEL);
363       (* DIE PROZEDUR SUCHT EIN ATOM IM ATOMSPEICHER DES HEAPS.IST ES NOCH    *)
364       (* NICHT VORHANDEN,SO WIRD ES AN DER RICHTIGEN STELLE EINSORTIERT.IN    *)
365       (* PA WIRD EIN VERWEIS AUF DIESES ATOM ABGELIEFERT.                     *)
366
367       BEGIN
368
369         IF AP^.NAME=ATOMSYM
370          THEN
371           PA:=AP
372          ELSE
373           IF AP^.NAME>ATOMSYM
374            THEN
375             IF AP^.L=NIL THEN
376                           BEGIN
377                            NEW(AP^.L);
378                            PA:=AP^.L;
379                            PA^.NAME:=ATOMSYM;
380                            PA^.L:=NIL;
381                            PA^.R:=NIL
382                           END
383                          ELSE
384                            INATOMTREE(ATOMSYM,AP^.L,PA)
385             ELSE                                          (* AP^.NAME<ATOMSYM*)
386              IF AP^.R=NIL THEN
387                           BEGIN
388                            NEW(AP^.R);
389                            PA:=AP^.R;
390                            PA^.NAME:=ATOMSYM;
391                            PA^.L:=NIL;
392                            PA^.R:=NIL
393                           END
394                          ELSE
395                            INATOMTREE(ATOMSYM,AP^.R,PA);
396
397       END;      (* OF PROCEDURE INATOMTREE  *)
398
399
400       FUNCTION LOOKAHEAD : CHAR;
401       (* DIE FUNKTION FINDET DAS NAECHSTE ZEICHEN <> BLANK OHNE ES WEGZULESEN.*)
402         VAR
403           I       :     LONGINT;                      (* HILFSZAEHLER    *)
404
405           PROCEDURE NEXTI;
406
407           BEGIN
408
409             IF I<LONGSTRMAX THEN I:=I+1
410                             ELSE FEHLER(10)
411
412           END;  (* OF NEXTI  *)
413
414         BEGIN
415
416          I:=J;
417          NEXTI;
418          WHILE K[I]=' ' DO NEXTI;
419          IF K[I]=')' THEN J:=I; LOOKAHEAD:=K[I]
420
421         END;    (* OF FUNCTION LOOKAHEAD  *)
422
423
424         PROCEDURE CONSTR(P : POHEAPEL);
425         (* DIE PROZEDUR BRICHT DEN S-AUSDRUCK REKURSIV AUF UND LEGT IHN IM HEAP*)
426         (* AB.                                                                  *)
```

```
427
428         VAR
429              PA : POATOMEL;                       (*ZEIGER IN DEN ATOMSPEICHER  *)
430
431         BEGIN
432          WHILE K[J]=' ' DO NEXTJ;
433          IF (K[J] IN LETTER) THEN                (* ATOM GEFUNDEN              *)
434                          BEGIN
435                           READATOM(ATOMSYMBOL);
436                           INATOMTREE(ATOMSYMBOL,ATOMWP,PA);
437                           P^.LA:=PA;
438                          END
439                         ELSE
440                          IF K[J]='(' THEN        (* LISTE BZW. LEERE LISTE=NIL *)
441                                       BEGIN
442                                        IF LOOKAHEAD=')' THEN     (*LEERE LISTE *)
443                                                         P^.LA:=B[0].AP
444                                                        ELSE
445                                                         BEGIN
446                                                          INITEL(PK);
447                                                          P^.LK:=PK;
448                                                          NEXTJ;
449                                                          CONSTR(PK);
450                                                         END;
451                                       END
452                                      ELSE FEHLER(7);
453        NEXTJ;
454        WHILE K[J]=' ' DO NEXTJ;
455        IF K[J]='.' THEN                          (* DOTTED PAIR GEFUNDEN       *)
456                  BEGIN
457                   NEXTJ;
458                   WHILE K[J]=' ' DO NEXTJ;
459                   IF (K[J] IN LETTER) THEN        (* ATOM GEFUNDEN             *)
460                                       BEGIN
461                                        READATOM(ATOMSYMBOL);
462                                        INATOMTREE(ATOMSYMBOL,ATOMWP,PA);
463                                        P^.RA:=PA;
464                                       END
465                                      ELSE
466                                       IF K[J]='('
467                                          THEN
468                                            IF LOOKAHEAD=')' THEN  (*LEERE LISTE*)
469                                                             P^.RA:=B[0].AP
470                                                            ELSE
471                                                             BEGIN
472                                                              INITEL(PK);
473                                                              P^.RK:=PK;
474                                                              NEXTJ;
475                                                              CONSTR(PK);
476                                                             END
477                                          ELSE FEHLER(7);
478                   NEXTJ;
479                   WHILE K[J]=' ' DO NEXTJ;
480                   IF K[J]<>')' THEN FEHLER(12);              (*ENDE DOTTED PAIR*)
481                  END
482                 ELSE                                        (*KEIN DOTTED PAIR*)
483                  IF K[J]=')' THEN
484                              P^.RA:=B[0].AP                 (*BESETZEN MIT NIL*)
485                             ELSE
486                              IF (K[J] IN LETTER)OR(K[J]='(')
487                                  THEN
488                                   BEGIN
489                                    INITEL(PK);
490                                    P^.RK:=PK;
491                                    CONSTR(PK);
492                                   END
493                                  ELSE FEHLER(7)
494
495        END;  (* OF PROCEDURE CONSTR  *)
496
497
498   BEGIN      (*  OF INHEAP  *)
499
500    for n:=1 to length(st) do k[n]:=st[n]; (* Kopieren vom String st ins Array k*)
501    for n:=length(st)+1 to longstrmax do k[n]:=' ';
```

```
502   J:=1;
503   IF BTOP>HEAPMAX THEN FEHLER(9);
504   B[BTOP].AP:=NIL;
505   B[BTOP].KP:=NIL;
506   IF K[1]='"' THEN                                    (* ATOMARER S-AUSDRUCK  *)
507                BEGIN
508                 J:=J+1;
509                 READATOM(ATOMSYMBOL);
510                 INATOMTREE(ATOMSYMBOL,ATOMWP,PA);
511                 B[BTOP].AP:=PA
512                END
513               ELSE                                    (* LISTE               *)
514                BEGIN
515                 IF LOOKAHEAD=')' THEN                 (* LEERE LISTE         *)
516                                  B[BTOP]:=B[0]
517                                 ELSE                  (* NICHTLEERE LISTE    *)
518                                  BEGIN
519                                   J:=J+1;
520                                   INITEL(PK);
521                                   B[BTOP].KP:=PK;
522                                   CONSTR(PK);
523                                  END
524                END;
525
526   (* Offset zum AR-Keller: erste moegliche Adresse ist 6. Falls die Anzahl    *)
527   (* der Konstanten 6 uebersteigt, wird der Offset entsprechend erhoeht.      *)
528
529   if btop-6>boffs then boffs:=btop-6;
530
531   gettime(h1,m1,s1,hs1);                (* TP-Pascal: Fuer Laufzeitbestimmung *)
532
533 END;   (* OF INHEAP  *)
534
535
536 PROCEDURE INIT;
537
538 (* DIE LZS-ROUTINE INIT SORGT FUER DIE VORBESETZUNGEN VON                     *)
539 (*         -   B[0]..B[2]     :HEAPKOPFLEISTE  :  B0=^NIL/B1=^TRUE/B2=^FALSE *)
540 (*         -   A[1]..A[3]     :STACK :  AR FUER MAINPROGRAM                   *)
541 (*                             (keine Bpp- und keine GDV-Zelle mehr!         *)
542 (*         -   IR[-1]         :INDEXREGISTER FUER MAINPROGRAM                 *)
543 (*         -              GLOBALE VARIABLE                                    *)
544 (*             GDV            :GENERALISIERTER DYNAMISCHER VORGAENGER        *)
545 (*             RASTTOP        :ERSTE FREIE ZELLE IM RUECKKEHRADRESSENSTACK   *)
546 (*             HVSTTOP        :ERSTE FREIE ZELLE IM HILFSVARIABLENSTACK      *)
547 (*             ARP            :MDL-MOMENTANES DYNAMISCHES NIVEAU             *)
548 (*             STATNIV        :MOMENTANES STATISCHES NIVEAU                  *)
549 (*             ATOP           :ERSTE FREIE ZELLE IM LAUFZZEITSTACK A         *)
550 (*
551 (*             gdvsttop       :erste frei Zelle im GDV-Stack                 *)
552 (*             pptop          :erste freie Zelle am Ende vom Laufzeitstack A *)
553 (*             bpp            :Begin Pending Parameter, ersetzt Linkage-Zelle*)
554 (*             boffs          :Offset der Heapkopfleiste zum AR-Keller.      *)
555 (*             T1update       :Liegt ein Typ-1 Update an?                    *)
556
557 VAR
558           I     :    LONGINT;              (*                    LAUFINDEX  *)
559           key   :    char;                 (*                    Hilfsvar.  *)
560
561 BEGIN
562  writeln;   (* Testversion: Verschieden Optimierungsstufen + Kellerprotokoll *)
563  writeln
564    ('GDV-Keller + dicke Parameter-Optimierung + pending Parameter-Keller +');
565  repeat
566    write('GDV-Optimierung: [K]eine / [D]efinitionsgemaess / [G]mark ? ');
567    readln(opt);
568    if opt='k' then opt:='K'; if opt='d' then opt:='D'; if opt='g' then opt:='G'
569  until opt in ['K','D','G'];
570  repeat
571    write('Soll Call By Need unterstuetzt werden:      [J]a / [N]ein ? ');
572    readln(key);
573    if key='j' then key:='J'; if key='n' then key:='N'
574  until key in ['J','N']; need:=key='J';
575  repeat
576    write('Soll Laufzeitkeller protokolliert werden:   [J]a / [N]ein ? ');
```

```
577       readln(key);
578       if key='j' then key:='J'; if key='n' then key:='N'
579     until key in ['J','N']; proto:=key='J';
580     if proto then begin assign(kell,'keller.dat'); rewrite(kell) end;
581     writeln; maxdeep:=0; anzar:=0; kellsum:=0; gdvstmax:=0;
582
583     NEW(ATOMWP);                              (*VORBESETZUNG B[0] MIT NIL       *)
584     B[0].AP:=ATOMWP;
585     B[0].KP:=NIL;
586     ATOMWP^.NAME:='NIL     ';                 (*EINTRAGEN VON NIL IN D. ATOMBAUM*)
587     NEW(ATOMWP^.L);                           (*VORBESETZUNG B[2] MIT F         *)
588     B[2].AP:=ATOMWP^.L;
589     B[2].KP:=NIL;
590     B[2].AP^.NAME:='F       ';                (*EINTRAGEN VON F IN DEN ATOMBAUM *)
591     B[2].AP^.L:=NIL;
592     B[2].AP^.R:=NIL;
593     NEW(ATOMWP^.R);                           (*VORBESETZUNG B[1] MIT T         *)
594     B[1].AP:=ATOMWP^.R;
595     B[1].KP:=NIL;
596     B[1].AP^.NAME:='T       ';                (*EINTRAGEN VON T IN DEN ATOMBAUM *)
597     B[1].AP^.L:=NIL;
598     B[1].AP^.R:=NIL;
599
600     boffs:=0;                                 (* Offset Heapkopfleiste/AR-Keller*)
601
602     a[stackmax]:=-1;                          (* Max PP-Keller-Verw.: Erstes AR *)
603     pptop:=stackmax-1;                        (*Erster freier Platz im PP-Keller*)
604     bpp:=0;                                   (* Begin Pending Parameter        *)
605
606     RASTTOP:=1;                               (*ERSTER FREIER PLATZ IM RA-STACK *)
607     HVSTTOP:=1;                               (*ERSTER FREIER PLATZ IM HV-STACK *)
608
609     ragdvst[ragdvstmax]:=1001;                (*erster GDV zeigt auf Hauptprogr.*)
610                                               (*und ist zunaechst maximaler GDV *)
611     gdvsttop:=ragdvstmax-1;                   (*erster freier Platz im GDV-Stack*)
612
613     T1update:=false;                          (* Glob. Var. fuer Call By Need   *)
614
615                                               (*LINKAGE FUER DAS MAINPROGRAM    *)
616     A[1]:=-1;                                 (*STATISCHES NIVEAU D. MAINPROGRAM*)
617     A[2]:=-1;                                 (* STATISCHER VERWEIS             *)
618     A[3]:=4;                                  (* B(EGIN) F(REIER) S(PEICHER)    *)
619
620     ARP:=1;                                   (* MOMENTANES DYNAMISCHES NIVEAU  *)
621
622     GDV:=1;                                   (* GDV ZEIGT AUF AR ZU MAINPROGRAM*)
623
624     IR[-1]:=1;
625
626     STATNIV:=0;
627
628     ATOP:=7;                                  (*ERSTER FREIER PLATZ IM AR-STACK *)
629                                               (* 3 ZELLEN FREI FUER LINKAGE     *)
630     LETTER:=['A'..'I']+['J'..'R']+['S'..'Z'];
631     LETDIG:=LETTER +['0'..'9'];
632
633     gettime(h1,m1,s1,hs1)                     (* TP: Fuer Laufzeitbestimmung    *)
634
635   END;    (* OF PROCEDURE INIT *)
636
637
638   PROCEDURE INP(BTOP:LONGINT);
639
640   (* DIE LZS-ROUTINE LIESST S-AUSDRUECKE VON INPUT.DER S-AUSDRUCK WIRD IN DER  *)
641   (* VARIABLEN K ABGELEGT UND DANACH WIE EINE PROGRAMMKONSTANTE MIT INHEAP     *)
642   (* IM HEAP EINGETRAGEN. BTOP GIBT DIE ZUGEHOERIGE ADRESSE IN DER HEAPKOPF-   *)
643   (* LEISTE AN.                                                               *)
644
645   VAR
646             K  : LONGSTR;             (*ZUR ZWISCHENSPEICHERUNG DES S-AUSDRUCKS*)
647             I  : LONGINT;             (*ZAEHLER                               *)
648
649    PROCEDURE READK;
650
651    BEGIN
```

```
652
653     IF I<LONGSTRMAX THEN I:=I+1
654                     ELSE FEHLER(10);
655     IF EOLN(INPUT) THEN FEHLER(11);     (* BEI EINGABE VOM BILDSCHIRM UNNOETIG *)
656     READ(INPUT,K[I]);
657     IF K[I]=' ' THEN                    (* MAX. EIN BLANK ZWISCHEN ZWEI ZEICHEN*)
658                  BEGIN
659                   IF K[I-1]=' ' THEN I:=I-1 ;
660                  END
661                 ELSE                              (* UMWANDELN KLEIN IN GROSSB.*)
662                  IF K[I] IN ['a'..'i','j'..'r','s'..'z'] THEN
663                           K[I]:=CHR( ORD(K[I]) - 32 )   (* ASCII - Zeichensatz *)
664
665    END;    (* OF READK   *)
666
667    PROCEDURE INP1;
668
669    BEGIN
670
671     WHILE K[I]<>')' DO
672                     BEGIN
673                      READK;
674                      WHILE K[I]='(' DO
675                                      IF K[I]='(' THEN
676                                                    BEGIN
677                                                     INP1;
678                                                     READK
679                                                   END;
680                     END
681    END;     (* OF INP1  *)
682
683
684   BEGIN
685
686    WRITELN(OUTPUT,
687           'BITTE S-AUSDRUCK EINGEBEN. MAXIMALE LAENGE EINES ATOMS:',MAXLENGTH:2);
688    WRITELN(OUTPUT,'EINEM EINZELNEM ATOM MUSS " VORANGESTELLT WERDEN. ');
689    FOR I:=1 TO LONGSTRMAX DO K[I]:=' ';
690    I:=1;
691    reset(input);                                (* Loeschen des Eingabe-Puffers *)
692    WHILE K[1]=' ' DO READ(INPUT,K[1]);          (* NUR BEI BILKDSCHIRMEINGABE   *)
693
694    WHILE (K[1]<>'(')AND(K[1]<>'"')DO
695                                   BEGIN
696                                    WRITELN(OUTPUT,
697                                        'S-AUSDRUCK MUSS MIT ( BZW. " BEGINNEN');
698                                    reset(input); READ(INPUT,K[1]);
699                                   END;
700    writeln;
701    IF K[1]='(' THEN INP1                                    (* LISTE          *)
702                ELSE WHILE NOT(EOLN(INPUT)) DO READK;        (* ATOM           *)
703    INHEAP(K,BTOP)
704
705   END;               (* OF PROCEDURE INP  *)
706
707
708   PROCEDURE LZSTOPAR(I:LONGINT);
709
710   (* DIE LZS-ROUTINE ERHOEHT ATOP UM I .FALLS pptop ERREICHT WIRD, SO ERFOLGT  *)
711   (* EINE FEHLERMELDUNG.                                                       *)
712
713   BEGIN
714
715    IF ((ATOP+I)<=pptop)    THEN ATOP:=ATOP+I   (* KELLERSPITZE UM I HOCHSETZEN *)
716                            ELSE FEHLER(15);    (* UEBERLAUF DES LAUFZEITKELLERS*)
717
718    END;   (* OF LZSTOPAR *)
719
720
721   procedure incpp(i: longint);
722
723   (* Die LZS-Routine erniedrigt pptop um i, falls ATOP erreicht wird, so       *)
724   (* erfolgt eine Fehlermeldung.                                               *)
725
726     begin
```

```
727       if pptop-i>=atop then pptop:=pptop-i     (* PP-Spitze um I herabsetzen   *)
728                        else fehler(15);        (* Ueberlauf des Laufzeitkellers*)
729     end;  (* of incpp *)
730
731
732   PROCEDURE LZSTRENN;
733
734   (* DIE LZS-ROUTINE TRAEGT EINE TRENNMARKE AN DER SPITZE DES LAUFZEITKELLERS  *)
735   (* EIN.                                                                      *)
736
737   BEGIN
738
739    A[ATOP]:=500000000;                                  (* TRENNMARKE SETZEN   *)
740    LZSTOPAR(1)                                          (* ATOP:=ATOP+1;       *)
741
742   END;            (* OF PROCEDURE LZSTRENN  *)
743
744
745   procedure statistik;
746     begin
747       writeln;
748       writeln('Maximale Kellertiefe: ',maxdeep:6,'    (AR+PP+GDV-Keller)');
749       writeln('Max. GDV-Kellertiefe: ',gdvstmax:6);
750       write(  'Durchschnittliche " : ');
751       if anzar=0 then writeln(0:6)
752                  else writeln(kellsum/anzar:9:2);
753       writeln('Angelegte AR''s      : ',anzar:6);
754       if not proto then
755         begin                                          (* Laufzeitbestimmung *)
756           write('Laufzeit in Sekunden: ');
757           writeln((h2*3600+m2*60+s2+hs2/100)-(h1*3600+m1*60+s1+hs1/100):9:2)
758         end;
759       if proto then close(kell); writeln
760     end;          (* of procedure statistik *)
761
762
763   procedure kellerausgabe;    (* Bei prot=true wird Keller-Protokoll erstellt. *)
764     var i,l,arend: longint;
765     begin
766       l:=1; arend:=4;
767       for i:=1 to a[arp+2]-1 do
768         begin
769           case l of
770             1: writeln(kell,'          ',i:5,' [SN]  :',a[i]:7);
771             2: writeln(kell,'          ',i:5,' [SV]  :',a[i]:7);
772             3: begin
773                  writeln(kell,'          ',i:5,' [BFS] :',a[i]:7);
774                  arend:=a[i]
775                end;
776             else writeln(kell,'          ',i:5,'       :',a[i]:7)
777           end;
778           l:=l+1;
779           if i=arend-1 then begin writeln(kell); l:=1 end
780         end;
781       for i:=pptop+1 to stackmax do
782         writeln(kell,'          ',i:5,'       :',a[i]:7);
783       if stackmax>pptop then writeln(kell);
784       write(kell,'     ',anzar:3,'.) SV-Kette   : ');
785       for i:=-1 to a[arp] do write(kell,ir[i],' ');
786       writeln(kell);
787       write(kell,'           GDV-Keller : ');
788       for i:=ragdvstmax downto gdvsttop+1 do write(kell,ragdvst[i] mod 1000,' ');
789       writeln(kell); writeln(kell);
790       writeln(kell,'     ----------------------------------');
791       writeln(kell)
792     end;
793
794
795   PROCEDURE LZSARK(TYP,STAADR,RA,STANIV: longint; GDVSTAT: CHAR; gmniv: longint);
796
797   (*  DIE PROZEDUR LEGT EIN ACTIVATION RECORD BEI AUFRUF EINER NICHTSTANDARD-  *)
798   (*  FUNKTION AN. SIEHE KAPITEL IV.4.K2 .                                     *)
799   (*  Moegliche pending Parameter werden auf den PP-Stack (er waechst im LZS-  *)
800   (*  Keller von oben nach unten) gelegt bzw. heruntergeholt. Die maximalen    *)
801   (*  statischen Verweise aus dem PP-Stack werden nur einmal berechnet und     *)
```

```
802   (*  in negativer Form anstatt der Trennmarke '500000000' auf dem PP-Stack    *)
803   (*  abgelegt. Die Linkagezelle BPP und die Endemarke '600000000' entfallen.  *)
804   (*  Ebenso entfaellt die Linkagezelle GDV. Sie wird in den GDV-Stack verlegt.*)
805   (*  Der Verweis GDV wird jetzt je nach Optimierungsstufe behandelt...        *)
806   (*  Der nach LZSARK erforderliche Streusprung 'goto 0' wird im Zielcode      *)
807   (*  hinter jeden Aufruf von LZSARK angehaengt.(Keine globalen Goto-Spruenge) *)
808   (*  Bei einer HOpt werden nur noch die akt. Parameter Verschoben. Die        *)
809   (*  Linkage und evtl. zu aktuellen Parametern werdende pending Parameter     *)
810   (*  werden sofort an die richtige Stelle geschrieben.                        *)
811   (*  Call By Need wird unterstuezt...                                         *)
812
813
814   VAR
815           ARPH           :   LONGINT;   (* ANFANG DES AUFZUBAUENDEN AR        *)
816           I,J,t,maxgdv   :   LONGINT;   (* HILFSVARIABLE                      *)
817           MAX            :   LONGINT;   (* HAELT VERWEIS F. SPEICHERPLATZFREIG*)
818           STAV           :   LONGINT;   (* STATISCHER VERWEIS BEI TYP=1       *)
819           ppmaxv         :   longint;   (* max. stat. Verweis aus PP-Keller   *)
820           pptoap         :   boolean;   (* PP als AP uebernehmen?             *)
821
822
823    PROCEDURE PUSHRA(RA : LONGINT );
824    (* DIE LZS-ROUTINE LEGT DIE RUECKKEHRADRESSE RA AN DER SPITZE DES RUECK-    *)
825    (* KEHRADRESSENSTACK AB.                                                    *)
826
827    BEGIN
828
829      RAgdvST[RASTTOP]:=RA;                            (* PUSH(RA)             *)
830      IF RASTTOP<gdvsttop THEN RASTTOP:=RASTTOP+1
831                          ELSE FEHLER(14)              (* UEBERLAUF DES RA-STACK *)
832
833    END;     (* OF PROCEDURE PUSHRA  *)
834
835
836    procedure pushgdv(ragdv: longint);
837    (* Legt einen GDV-Verweis an der Spitze des neuen GDV-Kellers ab.           *)
838
839    begin
840      ragdvst[gdvsttop]:=ragdv;                        (* push(gdv)            *)
841      if gdvsttop>rasttop then gdvsttop:=gdvsttop-1
842                          else fehler(17)              (* Ueberlauf im gdv-Stack *)
843
844    end; (* of procedure pushgdv *)
845
846
847   BEGIN                                          (* OF PROCEDURE LZSARK       *)
848
849    ARPH:=A[ARP+2];                               (* BFS DES DYN. VORGANGERS   *)
850
851    max:=1;     (* Initialisierung zur Bestimmung des maximalen stat. Verweises *)
852
853
854    (* Ermitteln des GDV:                                                       *)
855    (*                                                                          *)
856    (* Die Variable GMNIV liefert das maximale statische Niveau im rlk vom      *)
857    (* Aufruf.                                                                  *)
858    (*                                                                          *)
859    (* GDV-Verweise stehen nicht mehr in der Linkage eines AR sondern zusammen  *)
860    (* mit der jeweiligen RA-Adresse und dem maximalen GDV-Verweis im neuen     *)
861    (* GDV-Keller.                                                              *)
862    (*                                                                          *)
863    (* Die globale Variable RLACALL zeigt an, ob ein RLA-Aufruf noch aktiv ist. *)
864    (* Es wird ein mehrfaches Setzen des GDV in einem linken Ast verhindert und *)
865    (* so ein definitionsgemaesser GDV erreicht.                                *)
866
867    case opt of                                 (* Optimierungsstufe           *)
868
869    'K': if gdvstat='L' then                    (* Keine Optimierung.          *)
870           pushgdv(gdv);                        (* push ( - / - / GDV )        *)
871
872    'D': if gdvstat='L' then                    (* Def.-gemaesser GDV-Optimierung *)
873           if not rlacall then                  (* 1. RLA-Aufruf: GDV setzen   *)
874             begin
875               pushgdv(1000000*ra+gdv);         (* push ( RA / - / GDV )       *)
876               rlacall:=true                    (* RLA-Aufruf nun aktiv        *)
```

```
877             end;
878
879     'G': if gdvstat='L' then                (* Def.-gemaesser GDV + GMARK-Opt.  *)
880            if (not rlacall) and (gmniv>0) then
881                                             (* 1. LA-Aufruf und Idf's im rlk    *)
882             begin                           (* push ( RA / MaxGDV / neuer GDV ) *)
883               i:=ragdvst[gdvsttop+1]; maxgdv:=(i div 1000) mod 1000;
884               if ir[gmniv-1]>maxgdv         (* max (MaxGDV,neuer GDV) ermitteln *)
885                 then pushgdv(1000000*ra+1001*ir[gmniv-1])
886                 else pushgdv(1000000*ra+1000*maxgdv+ir[gmniv-1]);
887               rlacall:=true                 (* RLA-Aufruf nun aktiv             *)
888             end
889     end;
890
891
892     (* Ueberpruefung des GDV-Verweises                                          *)
893
894     if opt='G' then maxgdv:=(ragdvst[gdvsttop+1] div 1000) mod 1000
895                else maxgdv:=ragdvst[gdvsttop+1] mod 1000;
896
897     if maxgdv>max then max:=maxgdv;
898
899
900     (* Der maximale stat. Verweis aus dem PP-Keller im oberten Kellereintrag:   *)
901
902     if need
903       then ppmaxv:=-a[pptop+1] mod 1000
904       else ppmaxv:=-a[pptop+1];
905
906
907     (* Falls pending Parameter vorhanden sind und pending Parameter zu aktuellen*)
908     (* Parametern werden duerfen (nicht bei DP-Aufrufen (Typ=2) u. Aufrufen mit *)
909     (* leerer aktueller Parameterliste als einzige Parameterliste) und noch     *)
910     (* keine aktuellen Parameter eingetragen sind, wird die erste PP-Gruppe als *)
911     (* aktuelle Parameterliste ins AR kopiert.                                  *)
912     (* Zunaechst wird nur festgestellt, ob kopiert werden muss:                 *)
913
914     pptoap:=((pptop<stackmax-1) and (atop=arph+3) and (typ<2));
915
916
917     (* Falls neue PP vorhanden: verschieben in PP-Keller + neuen ppmaxv ermitt. *)
918
919     if (typ=2) and (atop>arph+3) then bpp:=arph+3; (* DP: alle Parameter zu PP's*)
920
921     if bpp>0 then                                  (* PP vorhanden             *)
922       begin
923         i:=atop-1;
924         while i>=bpp do                            (* Schleife ueber alle PP's  *)
925           begin
926             while (a[i]<>500000000) and (i>=bpp) do(* Schleife ueber 1 PP-Liste *)
927               begin
928                  t:=a[i] mod 10;                   (* Typ des pending Parameter *)
929                  case t of
930                    0: a[pptop]:=a[i];              (* S-Ausdruck -> kopieren    *)
931                  1,2: begin                        (* Typ1 oder Typ2:           *)
932                         j:=(a[i] div 10) mod 1000; (* stat.Verw. (SV) ermitteln *)
933                         if j>ppmaxv then ppmaxv:=j;(* ppmaxv := max (SV,ppmaxv) *)
934                         a[pptop]:=a[i]             (* kopieren                  *)
935                       end;
936                    5: begin                        (* kurzfristiger Typ-5:      *)
937                         (* Ein Typ-2 Parameter ist als aktueller Parameter      *)
938                         (* weitergereicht worden: Ueberpruefung ob die zugeh.   *)
939                         (* Parameterzelle durch GDV-Verweis geschuetzt ist:     *)
940                         (* Falls ja:   Typ-4 Referenz Parameter erzeugen,       *)
941                         (* Falls nein: Typ-2 Parameter kopieren.                *)
942
943                         j:=a[i] div 10;            (* Referenz-Adresse ADR      *)
944                         if j>a[maxgdv+2]           (* Kriterium NK1:            *)
945                           then begin               (*   ADR > (BFS vom GDV)     *)
946                                  a[pptop]:=a[j];   (* Typ-5 ueberschr. mit Typ-2*)
947                                  j:=(a[j] div 10) mod 1000; (* stat. Verw. (SV) *)
948                                  if j>ppmaxv       (* ppmaxv := max (SV,ppmaxv) *)
949                                    then ppmaxv:=j
950                                end
951                           else a[pptop]:=a[i]-1    (* Aus Typ-5 wird Typ-4 und  *)
```

```
952                     end                             (* nicht max.Verw. bestimmen!*)
953                   end; i:=i-1; pptop:=pptop-1       (* naechster Parameter...    *)
954               end;
955             a[pptop]:=-ppmaxv; incpp(1);            (* neue Trennmarke setzen    *)
956             if a[i]=500000000 then i:=i-1           (* alte Trennmarke ueberlesen*)
957           end;
958         atop:=i+1                                   (* neue AR-Kellerspitze      *)
959       end;
960
961     if a[arph+3]=500000000 then atop:=arph+3;       (* Trennmarke entfernen      *)
962
963
964     (* Ueberpruefen der statischen Verweise aus dem PP-Keller                    *)
965
966     if ppmaxv>max then max:=ppmaxv;
967
968
969     (* Ablegen der RA im RA-STACK.                                               *)
970     (* Call By Need: Protokollieren der absoluten Stack-Adresse (ADR) der        *)
971     (* spaeter zu aktualisierenden Typ-2 Zelle(n) zu dicken Parametern.          *)
972     (* Um sicherzustellen, dass die zu aktualisierende Zelle noch nicht frueh-   *)
973     (* zeitig freigegeben wurde, wird folgendes getestet:                        *)
974     (*   Typ-0 Update: Die Zelle wird durch einen GDV-Verweis geschuetzt;        *)
975     (*   Typ-1 Update: Die Zelle wird durch einen GDV-Verweis ODER durch einen   *)
976     (*                 statischen Verweis in den pending Parametern geschuetzt.  *)
977     (* Gilt T1update=true, so ergibt die Auswertung des dicken Parameters einen  *)
978     (* Funktionsidentifikator (Typ-1) und die ADR wird im PP-Keller zusammen mit*)
979     (* der Information PPmaxV gespeichert.                                       *)
980     (* Gilt T1update=false, so ergibt die Auswertung des dicken Parameters einen*)
981     (* S-Ausdruck (Typ-0) und die ADR wird zusammen mit der RA gespeichert.      *)
982
983     if need                                 (* Kriterium NK2 ueberpruefen:      *)
984       then if (typ=2) and (a[max+2]>adr)    (* ADR < (BFS von max(GDV,PPmaxV)) *)
985              then if T1update
986                     then begin                                  (* Typ-1 Ergebnis *)
987                            pushra(ra);                          (* RA kellern     *)
988                            a[pptop+1]:=-1000*adr+a[pptop+1];    (* - ADR | PPmaxV *)
989                            T1update:=false                      (* Ruecksetzen    *)
990                          end
991                     else pushra(1000*adr+ra)                    (* Typ-0 Ergebnis *)
992                                                                 (*    ADR | RA    *)
993              else begin                                         (* kein Update    *)
994                     pushra(ra);                                 (* RA kellern     *)
995                     T1update:=false                             (* Ruecksetzen    *)
996                   end
997       else pushra(ra);                                          (* RA kellern     *)
998
999
1000    (*  BEI TYP=1(FORMALER AUFRUF) BERECHNEN VON STANIV,STAADR UND STAV          *)
1001    (*  EINTRAG IN PARAMETERZELLE: 1.STELLE:TYP/2-4.:STAT.VERW./5-10:STARTADR    *)
1002
1003    IF typ>0                                    (* FORMALER FUNKTIONSAUFRUF  *)
1004      THEN BEGIN
1005             I:=IR[STANIV-1]+2+STAADR;          (* ABSOLUTE ADRESSE          *)
1006             if (a[i] mod 10)=4
1007               then i:=a[i] div 10;
1008             STAADR:=A[I] div 10000;            (* STARTADRESSE AB 5. STELLE *)
1009             STAV:=(A[I] div 10) MOD 1000;      (* STAT. VERW. IN 2.-4.STELLE*)
1010             STANIV:=A[STAV]+1;                 (* STATISCHES NIVEAU         *)
1011             ir[staniv-1]:=stav;                (* IR zum SV besetzen        *)
1012             for i:=staniv-2 downto 0 do        (* Umladen der Indexregister *)
1013               ir[i]:=a[ir[i+1]+1]
1014           END
1015      else stav:=ir[staniv-1];                  (* stav enthlt stat. Verw.  *)
1016
1017
1018    (* Ueberpruefung der statischen Verweise aus aktuellen Parametern           *)
1019
1020    if not pptoap then (* Nicht noetig, falls PP zu aktuellen Parametern werden *)
1021      if need
1022        then for i:=arph+3 to atop-1 do         (* Schleife ueber alle Para. *)
1023               begin
1024                 t:=a[i] mod 10;                (* Typ des Parameters        *)
1025                 case t of
1026                 1,2: begin                     (* Typ1 oder Typ2:           *)
```

```
1027                      j:=(a[i] div 10) mod 1000; (* stat.Verw. (SV) ermitteln *)
1028                      if j>max then max:=j       (* max := max ( SV , max )   *)
1029                    end;
1030                  5: begin                       (* kurzfristiger Typ-5:      *)
1031                      (* Ein Typ-2 Parameter ist als aktueller Parameter      *)
1032                      (* weitergereicht worden: Ueberpruefung ob die zugeh.   *)
1033                      (* Parameterzelle durch GDV-Verweis geschuetzt ist:     *)
1034                      (* Falls ja:   Typ-4 Referenz-Parameter erzeugen,       *)
1035                      (* Falls nein: Typ-2 Parameter kopieren.                *)
1036
1037                      j:=a[i] div 10;            (* Referenz-Adresse ADR      *)
1038                      if j>a[maxgdv+2]           (* Kriterium NK1:            *)
1039                        then begin               (*    ADR > (BFS vom GDV)    *)
1040                               a[i]:=a[j];       (* Typ-5 ueberschr. mit Typ-2*)
1041                               j:=(a[j] div 10) mod 1000; (* stat. Verw. (SV) *)
1042                               if j>max          (* max := max ( SV , max)    *)
1043                                 then max:=j
1044                             end
1045                        else a[i]:=a[i]-1        (* Aus Typ-5 wird Typ-4 und  *)
1046                    end                          (* nicht max.Verw. bestimmen!*)
1047                end
1048              end
1049       else FOR I:=ARPH+3 TO ATOP-1 DO           (* Kein Call By Need:        *)
1050              IF A[I] MOD 10>0 THEN              (* FUNKTIONSPARAMETER        *)
1051                BEGIN
1052                 J:=(A[I] div 10) MOD 1000;      (* 2.-4. STELLE: STAT. VERW. *)
1053                 IF J>MAX THEN MAX:=J            (* NEUES MAXIMUM             *)
1054                END;
1055
1056
1057    (* Ueberpruefung des Verweises auf den statischen Vorgaenger               *)
1058
1059    if stav>max then max:=stav;
1060
1061
1062    IF A[MAX+2]<ARPH THEN   (* BFS VON MAX < ARPH, D.H. SPEICHERPLATZBEREINIGUNG*)
1063      BEGIN                                     (* "HOpt"                    *)
1064        J:=A[MAX+2];                            (* BFS ZU MAX:NEUER AR-ANFANG*)
1065        FOR I:=ARPH+3 TO ATOP-1 DO
1066          A[J+I-ARPH]:=A[I];                    (* VERSCHIEBEN               *)
1067        ATOP:=ATOP-ARPH+J;                      (* NEUER WERT F. LINKAGEZ.BFS*)
1068        ARPH:=J
1069      END;
1070
1071
1072    (* Kopieren von pending Parametern ins AR als aktuelle Parameter. Diese     *)
1073    (* Manahme wird erst nach evtl. Durchfuehrung einer HOpt durchgefuehrt!    *)
1074    (* AUCH IM IF-PART WERDEN PENDING-PARAMETER ANGEFUEGT.DIESE WERDEN ABER     *)
1075    (* NICHT DURCH EINEN AUFRUF IM IF-PART VERBRAUCHT. (DA DER UBERSETZER NUR   *)
1076    (* ECHTE PROGRAMME AKZEPTIERT.) SIE MUESSEN JEDOCH FUER DIE ZUGEHOERIGEN    *)
1077    (* AUFRUFE IM THEN BZW. ELSE-PART GERETTET WERDEN.                          *)
1078    (*                                                                          *)
1079    (* Call By Need (T1update): Falls PPmaxV>1000 gilt, so ist der (mit einer   *)
1080    (* aktuellen Parameterliste zu versorgende) Funktionsausdruck das Ergebnis  *)
1081    (* einer dicken Parameter-Auswertung und wird an die Adresse adr als Typ-1  *)
1082    (* Parameter zusammen mit dem Verweis auf ein dynamisches Niveau des        *)
1083    (* statischen Vorgaenger geschrieben.                                       *)
1084    (* Dies wird NUR DANN gemacht, wenn "ein dynamisches Niveau des statischen  *)
1085    (* Vorgaengers" unterhalb der zu aktualisierenden Zelle liegt:              *)
1086    (* Nur dann koennen bei einem spaeteren Aufruf die Indexregisterkette       *)
1087    (* korekt umgeladen werden!                                                 *)
1088
1089    if pptoap then                          (* Abfrage siehe oben...           *)
1090      begin
1091        pptop:=pptop+1;
1092        if need then                        (* Call By Need:                   *)
1093          begin
1094            i:=-a[pptop] div 1000;          (* Ref.-Adresse beim Eintrag PPmaxV*)
1095            if i>0 then                     (* Falls ADR vorhanden:            *)
1096              begin                         (* Kriterium NK3 ueberpruefen:     *)
1097               j:=ir[staniv-1];
1098               if j<i then                  (* stat. Verweis < ADR             *)
1099                 a[i]:=staadr*10000+j*10+1  (* Falls ja: Typ-1 Update          *)
1100              end
1101          end;
```

```
1102        while a[pptop+1]>0 do                (* Kopieren bis Trennmarke         *)
1103          begin                              (* Trennmarke ist negatives ppmaxv *)
1104            pptop:=pptop+1;
1105            a[atop]:=a[pptop];               (* Kopie: PP-Keller -> AR          *)
1106            atop:=atop+1
1107          end
1108      end;
1109
1110    (* Aktuelles IR besetzen (erfolgt erst NACH moegl. HOpt):                    *)
1111
1112    IR[STANIV]:=ARPH;
1113
1114
1115    (* Schreiben der neuen Linkage (erfolgt erst NACH moegl. HOpt):              *)
1116
1117    A[ARPH]   :=STANIV;                             (* Statischen Niveau        *)
1118    A[ARPH+1] :=STAV;                               (* Statischer Verweis       *)
1119    A[ARPH+2] :=ATOP;                               (* Begin Freier Speicher    *)
1120
1121
1122    (* ABSCHLUSSHANDLUNGEN                                                      *)
1123
1124    ARP:=ARPH;            (* VERWEIS AUF DAS GERADE ANGELEGTE AR NUN IN ARP     *)
1125    GDV:=ARP;             (** DIE GLOBALE VAR. GDV ZEIGT JETZT AUF DIESES AR.  **)
1126                          (** DIE UEBERNAHME DES GDV AUS DEM DYN. VORGAENGER   **)
1127                          (** IN DER SITUATION "NORMALER" AUFRUF WIRD DURCH    **)
1128                          (** GDVSTAT='R' GESICHERT.(DORT WIRD DANN GERADE AUF **)
1129                          (** DEN VORHERIGEN GDV-WERT ZURUECKGESETZT.)         **)
1130                          (** AM FUNKTIONSENDE WIRD EBENFALLS DER ALTE ZUSTAND **)
1131                          (** WIEDERHERGESTELLT.                              **)
1132    LZSTOPAR(3);          (*  ATOP:=ATOP+3         3 FREIE ZELLEN FUER LINKAGE **)
1133    STATNIV:=STANIV;      (* MOMENTANES STATISCHES NIVEAU BEI AUSF. DER FUNKTION *)
1134    GOTOVAR:=STAADR;      (* LADEN DER STARTADRESSE                           *)
1135    bpp:=0;               (* Begin Pending Parameter mit 0 vorbesetzen        *)
1136
1137
1138    (* Fuer Statistik und Kellerprotokoll:                                      *)
1139
1140    aktdeep:=a[arp+2]+stackmax-pptop-1+ragdvstmax-gdvsttop;
1141    if aktdeep>maxdeep then maxdeep:=aktdeep;
1142    kellsum:=kellsum+aktdeep; anzar:=anzar+1;
1143    if ragdvstmax-gdvsttop>gdvstmax then gdvstmax:=ragdvstmax-gdvsttop;
1144    if proto then kellerausgabe
1145
1146  END;          (* OF PROCEDURE LZSARK *)
1147
1148
1149  PROCEDURE LZSFORMIDF(STATNIV,RELADD : longint;VAR TYP ,ADR: longint);
1150
1151  (* DIE LZS-ROUTINE LIEFERT ZUR LAUFZEIT DEN AKTUELLEN TYP UND GGF. DIE       *)
1152  (* REALATIVADRESSE EINES FORMALEN IDENTIFIKATORS .                           *)
1153  (* Call By Need: Falls der Typ eines Parameters gleich 4 ist, so liegt ein   *)
1154  (* Referenz-Verweis vor. Dann erfolgt ein Zugriff an die Referenz-Adresse!   *)
1155  (* Der VAR-Parameter ADR enthaelt bei einem Typ-0 Parameter wie ueblich die  *)
1156  (* Heap-Adresse, bei Typ-2 Parametern jedoch die absolute Keller-Adresse des *)
1157  (* Parameters. Diese Adresse wird dann evtl. im RA-Keller bzw. im PP-Keller  *)
1158  (* fuer einen spaeteren Update protokolliert.                                *)
1159
1160   VAR           H     : LONGINT;                  (* HILFSVARIABLE           *)
1161
1162   BEGIN
1163
1164     h:=IR[STATNIV-1]+2+RELADD;                    (* ABSOLUTE ADRESSE IM STACK*)
1165     TYP:=a[h] MOD 10;                             (* 1. STELLE : TYP          *)
1166     if typ=4                                      (* Referenz-Parameter:      *)
1167       then begin
1168              h:=a[h] div 10;                      (* neue abs. Stack-Adresse  *)
1169              typ:=a[h] mod 10                     (* neuer Typ                *)
1170            end;
1171     IF TYP=0                                      (* KONSTANTE                *)
1172       THEN ADR:=a[h] DIV 10000                    (* 5.-10.STELLE:HEAPADRESSE *)
1173       else adr:=h                                 (* Typ1,2: absolute Adresse *)
1174                                                   (* nur fuer Typ2 relevant...*)
1175
1176   END;    (* OF PROCEDURE LZSFORMIDF *)
```

```
1177
1178
1179  PROCEDURE LZSLEFTEND(ra: longint);
1180
1181  (* DIE LZS-ROUTINE SETZT IN DER SITUATION "LINKER AST BEENDET" DEN GDV       *)
1182  (* ZURUECK UND LAEDT DIE INDEXREGISTER UM.                                  *)
1183  (* Der GDV wird nur zurueckgesetzt, falls der Aufruf beendet ist, der auch  *)
1184  (* den GDV gesetzt hat (Kriterium: RA's stimmen ueberein) !                 *)
1185  (* In nicht optimierter Version (opt='K') wird immer zurueckgesetzt, weil bei*)
1186  (* jedem Aufruf im linken Ast der GDV neu gesetzt wurde.                    *)
1187  (* Der GDV-Verweis steht nicht mehr in der Linkage, sondern im GDV-Keller   *)
1188  (* zusammen mit der Rueckkeraddresse des fuer den GDV-Eintrag verantwotlichen*)
1189  (* Aufrufs und dem jeweils aktuellen max. GDV-Keller-Verweis...             *)
1190
1191  VAR    STATNIV   : longint;
1192
1193  BEGIN
1194
1195   if (ragdvst[gdvsttop+1] div 1000000=ra) or (opt='K') then   (* Abfrage s.o. *)
1196     begin
1197       GDV:=ragdvst[gdvsttop+1] mod 1000;  (* GDV zuruecksetzen               *)
1198       IR[a[gdv]]:=GDV;                    (* Umladen des IR zum stat. Niveau  *)
1199       IF a[gdv]>0 THEN                    (* Umladen nur bis zum IR[0] noetig *)
1200         FOR STATNIV:=A[GDV]-1 DOWNTO 0 DO (* Umladen weiterer Indexregister   *)
1201           IR[STATNIV]:=A[IR[STATNIV+1]+1];
1202       gdvsttop:=gdvsttop+1;               (* Obersten GDV-Eintrag loeschen    *)
1203       rlacall:=false          (* Es kann wieder ein GDV im RLA gesetzt werden *)
1204     end
1205
1206  END; (* OF PROCEDURE LZSLEFTEND    *)
1207
1208
1209  PROCEDURE LZSPARB(TYP,STANIV,ADR : longint );
1210
1211  (* DIE PROZEDUR TRAEGT EINEN AKTUELLEN PARAMETER AN DER SPITZE DES LAUFZEIT- *)
1212  (* KELLERS A EIN. SIEHE KAPITEL IV.4.K1 .                                   *)
1213  (* Es wird nicht mehr die Linkagezelle BPP, sondern die globale Variable BPP *)
1214  (* aktualisiert.                                                            *)
1215  (* Call By Need: Falls ein formaler Identifikator als aktueller Parameter   *)
1216  (* in das AR eingetragen werden soll, wird ueberprueft, ob der ermittelte   *)
1217  (* Parameter vom Typ-2 ist: Falls ja, wird ein kurzfristiger Typ-5 einge-   *)
1218  (* richtet, weil in dieser Routine LZSPARB die MaxGDV-Information als       *)
1219  (* Entscheidungs-Kriterium fuer einen moeglichen Referenz-Verweis noch nicht *)
1220  (* vorliegt. In LZSARK wird diese Kriterium ueberprueft und entweder aus dem *)
1221  (* Typ-5 Parameter mit dem schon vorhandenen Referenz-Verweis ein Typ-4     *)
1222  (* Parameter gemacht, oder der Parameter von der Referenz-Adresse           *)
1223  (* nachtraeglich kopiert.                                                   *)
1224
1225  var h: longint;                  (* Hilfsvariable                          *)
1226
1227  BEGIN
1228
1229   IF (A[ATOP-1]=500000000) and (a[arp+2]+3<atop)
1230         THEN                      (* ES WURDE BEREITS EINE TRENNMARKE GESETZT*)
1231          IF bpp=0 THEN            (* BPP NOCH NICHT BESETZT                  *)
1232                    bpp:=ATOP;     (* BPP MIT ATOP BESETZEN                   *)
1233
1234   CASE TYP OF
1235     0 : A[ATOP]:=ADR*10000+STANIV*10;                  (* KONSTANTE:          *)
1236                                                        (* HEAPADR/STA.NIV/0   *)
1237     1 : A[ATOP]:=ADR*10000+(IR[STANIV-1]*10)+1;        (* FUNKTIONSIDENTIFIK.:*)
1238                                                        (* STARTADR/STA.VERW./1*)
1239     2 : A[ATOP]:=ADR*10000+(IR[STANIV-1]*10)+2;        (* F-IDF. OHNE PARAM.: *)
1240                                                        (* STARTADR./STA.VERW/2*)
1241     3:                                                 (* FORMALER IDENTIFIK.:*)
1242                                                        (*   AKTUELLER WERT    *)
1243        if need                                         (* Call By Need:       *)
1244          then begin
1245                 h:=ir[staniv-1]+2+adr;                 (* Abs. Stack-Adresse  *)
1246                 if (a[h] mod 10)=2                     (* Falls Typ-2, dann   *)
1247                   then a[atop]:=10*h+5                 (* prov. Typ-5 erzeug. *)
1248                   else a[atop]:=a[h]                   (* Sonst: kopieren     *)
1249               end
1250          else A[ATOP]:=A[IR[STANIV-1]+2+ADR]           (* kein Call By Need   *)
1251    END;
```

```
1252
1253    LZSTOPAR(1);                                          (* ATOP:=ATOP+1        *)
1254
1255   END;  (* OF PROCEDURE  LZSPARB *)
1256
1257
1258    PROCEDURE OUT;
1259
1260    (* LZS-ROUTINE ZUR AUSGABE VON S-AUSDRUECKEN AUF OUTPUT. DER VERWEIS AUF    *)
1261    (* DEN AUSZUGEBENDEN S-AUSDRUCK WIRD IM AC GEHALTEN.                        *)
1262
1263
1264     PROCEDURE WRITEATOM(NAME : ATOMSTR);
1265
1266      VAR
1267          I : 1..MAXLENGTH;                     (* LAUFVARIABLE *)
1268
1269      BEGIN
1270        I := 1;
1271        WHILE (I < MAXLENGTH) AND (NAME[I] <> ' ')
1272         DO
1273          BEGIN
1274            WRITE(NAME[I]);
1275            I := I + 1
1276          END
1277       (*OD*);
1278        IF (I = MAXLENGTH) AND (NAME[I] <> ' ')
1279         THEN
1280           WRITE(NAME[I])
1281      (*FI*)
1282      END; (* OF PROCEDURE WRITEATOM *)
1283
1284
1285     PROCEDURE WRITELIST(P : POHEAPEL);
1286
1287      BEGIN
1288        (***   AUSGABE DES CAR   ***)
1289        IF P^.LA <> NIL
1290         THEN (* DER CAR IST ATOMAR *)
1291           WRITEATOM(P^.LA^.NAME)
1292         ELSE (* DER CAR IST STRUKTURIERT *)
1293           BEGIN
1294             WRITE('(');
1295             WRITELIST(P^.LK);
1296             WRITE(')')
1297           END
1298      (*FI*);
1299
1300        (***   AUSGABE DES CDR   ***)
1301        IF P^.RK <> NIL
1302         THEN (* DER CDR IST STRUKTURIERT *)
1303           BEGIN
1304             WRITE(' ');
1305             WRITELIST(P^.RK)
1306           END
1307         ELSE (* DER CDR IST ATOMAR *)
1308           IF P^.RA^.NAME <> 'NIL     '
1309            THEN (* DOTTED PAIR *)
1310              BEGIN
1311                WRITE(' . ');
1312                WRITEATOM(P^.RA^.NAME)
1313              END
1314         (*FI*)
1315      (*FI*)
1316      END; (* OF PROCEDURE WRITELIST *)
1317
1318
1319     BEGIN (* OUT *)
1320       gettime(h2,m2,s2,hs2);              (* TP-Pascal: Fuer Laufzeitbestimmung *)
1321       IF AC.AP <> NIL
1322        THEN (* ATOMARER S-AUSDRUCK *)
1323          WRITEATOM(AC.AP^.NAME)
1324        ELSE (* STRUKTURIERTER S-AUSDRUCK *)
1325          BEGIN
1326            WRITE('(');
```

```
1327            WRITELIST(AC.KP);
1328            WRITE(')')
1329          END
1330     (*FI*);
1331       WRITELN
1332     END;       (* OF PROCEDURE OUT *)
1333
1334
1335   PROCEDURE PUSH;
1336
1337    (* DIE LZS-ROUTIN TRAGT DEN INHALT VON AC IN DEN HILFSVARIABLENKELLER EIN.  *)
1338
1339   BEGIN
1340
1341    HVSTACK[HVSTTOP]:=AC;                          (* EINTRAGEN DES AC-INHALTES *)
1342    IF HVSTTOP<HVSTMAX THEN HVSTTOP:=HVSTTOP+1     (* NEUE KELLERSPITZE         *)
1343                       ELSE FEHLER(16);            (* UEBERLAUF DES HV-STACKS   *)
1344
1345   END;          (* OF PROCEDURE PUSH  *)
1346
1347
1348   FUNCTION BOOLTEST : BOOLEAN;
1349
1350   (* DIE LZS-ROUTINE PRUEFT OB DER AC AUF DEN SPEZIELLEN S-AUSDRUCK T BZW. F   *)
1351   (* ZEIGT.BOOLTEST NIMMT DANN DEN WERT TRUE BZW. FALSE AN. ANDERNFALLS ERFOLGT*)
1352   (* EINE LAUFZEITFEHLERMELDUNG.                                               *)
1353
1354   BEGIN
1355
1356    IF (ac.ap=b[1].ap) and (ac.kp=b[1].kp)                    (* ac=b[1] ?    *)
1357               THEN BOOLTEST:=TRUE                            (* B[1]=^ TRUE  *)
1358               ELSE IF (ac.ap=b[2].ap) and (ac.kp=b[2].kp)    (* ac=b[2] ?    *)
1359                                THEN BOOLTEST:=FALSE          (* B[2]=^ FALSE *)
1360                                ELSE FEHLER(1);
1361
1362   END;    (* OF FUNCTION BOOLTEST *)
1363
1364
1365   PROCEDURE LZSFEND;
1366
1367   (* DIE LZS-ROUTINE SORGT FUER DEN RUECKSPRUNG AM ENDE DER AUSFUEHRUNG EINER  *)
1368   (* NICHTSTANDARDFUNKTION.                                                    *)
1369
1370   BEGIN
1371
1372    RASTTOP:=RASTTOP-1;                        (* POP(RASTTOP)                *)
1373    if need
1374      then GOTOVAR:=RAgdvST[RASTTOP] mod 1000  (* LADEN DER RUECKKEHRADRESSE    *)
1375      else GOTOVAR:=RAgdvST[RASTTOP]           (* dabei ausblenden einer moegl. *)
1376                                               (* Absolut-Adresse (CallByNeed)  *)
1377   END;           (* OF PROCEDURE LZSFEND   *)
1378
1379
1380   procedure T0update;
1381
1382   (* Call by Need: Falls zusammen mit der Ruecksprungadresse eine absolute     *)
1383   (* Keller-Adresse ADR gespeichert wurde, wird ein Typ-0 Update durchgefuehrt:*)
1384   (* Die RA gehoert zur Auswertung eines dicken Parameters und das Ergebnis ist*)
1385   (* ein S-Ausdruck. Er steht im AC als Verweis in den Heap und wird als Typ-0 *)
1386   (* Parameter an die Adresse ADR geschrieben.                                 *)
1387
1388     var i: longint;                         (* Hilfsvariable                 *)
1389
1390     begin
1391       i:=ragdvst[rasttop] div 1000;         (* Absolutadresse der Typ-2 Zelle  *)
1392       if i>0 then                           (* ADR im RA-Keller vorhanden      *)
1393         begin                               (* Typ-0 Update:                   *)
1394           a[i]:=(i+boffs)*10000+statniv*10; (* Typ-0 Parameter schreiben       *)
1395           b[i+boffs].ap:=ac.ap;             (* Heapkopfleiste zeigt auf den    *)
1396           b[i+boffs].kp:=ac.kp              (* AC-Inhalt im Heap               *)
1397         end
1398     end;
1399
1400
1401   BEGIN  (* OF MAINPROGRAM   *)
```

```
1402
1403    INIT;                          (* Vorbesetzungen                              *)
1404    GOTO 1;                        (* Beginn des Hauptprogramm-Codes             *)
1405
1406
1407   3:                              (* CODE FUER DIE LISP/N-STANDARDFUNKTION ATOM *)
1408    LZSFORMIDF(1,1,TYP,ADR);       (* ERSTER PARAMETER IM AR ZU IR[1-1],RELADD 1 *)
1409    IF TYP=0 THEN AC:=B[ADR]
1410             ELSE BEGIN
1411                    LZSARK(1,1,9,1,'R',0); goto 0;
1412                  END;
1413   9: ATOM;                         (* AUSFUEHRUNG VON ATOM                       *)
1414    LZSFEND; goto 0;
1415
1416
1417   4:                               (* CODE FUER DIE LISP/N-STANDARDFUNKTION CAR *)
1418    LZSFORMIDF(1,1,TYP,ADR);        (* ERSTER PARAMETER IM AR ZU IR[1-1],RELADD 1*)
1419    IF TYP=0 THEN AC:=B[ADR]
1420             ELSE BEGIN
1421                   LZSARK(1,1,10,1,'R',0); goto 0;
1422                  END;
1423   10: CAR;                         (* AUSFUEHRUNG VON CAR                        *)
1424    LZSFEND; goto 0;
1425
1426
1427   5:                              (* CODE FUER DIE LISP/N-STANDARFUNKTION CDR   *)
1428    LZSFORMIDF(1,1,TYP,ADR);       (* ERSTER PARAMETER IM AR ZU IR[1-1],RELADD 1 *)
1429    IF TYP=0 THEN AC:=B[ADR]
1430             ELSE BEGIN
1431                   LZSARK(1,1,11,1,'R',0); goto 0;
1432                  END;
1433   11: CDR;                         (* AUSFUEHRUNG VON CDR                        *)
1434    LZSFEND; goto 0;
1435
1436
1437   6:                              (* CODE FUER DIE LISP/N-STANDARDFUNKTION CONS *)
1438    rlacall:=false;                (* Eintritt in den linken Ast                 *)
1439    LZSFORMIDF(1,1,TYP,ADR);       (* ERSTER PARAMETER IM AR ZU IR[1-1],RELADD 1 *)
1440    IF TYP=0 THEN AC:=B[ADR]
1441             ELSE BEGIN
1442                   LZSARK(1,1,12,1,'L',1); goto 0;  (* GDV ist immer AR zu SN 1 *)
1443   12:             LZSLEFTEND(12);
1444                  END;
1445    rlacall:=true;                 (* Austritt aus dem linken Ast                *)
1446    PUSH;                          (* ABLEGEN DES 1. ARGUMENTES IM HV-STACK      *)
1447    LZSFORMIDF(1,2,TYP,ADR);       (* ZWEITER PARAMETER IM AR ZU IR[1-1],RELADD 2*)
1448    IF TYP=0 THEN AC:=B[ADR]
1449             ELSE BEGIN
1450                   LZSARK(1,2,13,1,'R',0); goto 0;
1451                  END;
1452   13: CONS;                        (* AUSFUEHRUNG VON CONS                       *)
1453    LZSFEND; goto 0;
1454
1455
1456   8:                              (* Code fuer die LISP/N-Standardfunktion PCONS*)
1457    rlacall:=false;                (* Eintritt in den linken Ast                 *)
1458    lzsformidf(1,1,typ,adr);       (* erster parameter im ar zu ir[1-1], reladd 1*)
1459    if typ=0 then ac:=b[adr]
1460             else begin
1461                   lzsark(1,1,16,1,'L',1); goto 0;  (* GDV ist immer AR zu SN 1 *)
1462   16:             lzsleftend(16);
1463                  end;
1464    rlacall:=true;                 (* Austritt aus dem linken Ast                *)
1465    push;                          (* ablegen des 1. argumentes im hv-stack      *)
1466    lzsformidf(1,2,typ,adr);       (* zweiter parameter im ar zu ir[1-1],reladd 2*)
1467    if typ=0 then ac:=b[adr]
1468             else begin
1469                   lzsark(1,2,17,1,'R',0); goto 0;
1470                  end;
1471   17: pcons;                    (* ausfuehrung von pcons                       *)
1472    lzsfend; goto 0;
1473
1474
1475   7:                               (* CODE FUER DIE LISP/N-STANDDARDFUNKTION EQ  *)
1476    rlacall:=false;                 (* Eintritt in den linken Ast                 *)
```

```
1477    LZSFORMIDF(1,1,TYP,ADR);      (* ERSTER PARAMETER IM AR ZU IR[1-1],RELADD 1 *)
1478    IF TYP=0 THEN AC:=B[ADR]
1479             ELSE BEGIN
1480                   LZSARK(1,1,14,1,'L',1); goto 0;  (* GDV ist immer AR zu SN 1 *)
1481   14:             LZSLEFTEND(14);
1482                  END;
1483    rlacall:=true;               (* Austritt aus dem linken Ast               *)
1484    PUSH;                        (* ABLEGEN DES 1. ARGUMENTES IM HV-STACK     *)
1485    LZSFORMIDF(1,2,TYP,ADR);      (* ZWEITER PARAMETER IM AR ZU IR[1-1],RELADD 2*)
1486    IF TYP=0 THEN AC:=B[ADR]
1487             ELSE BEGIN
1488                   LZSARK(1,2,15,1,'R',0); goto 0;
1489                  END;
1490   15: EQ;                       (* AUSFUEHRUNG VON EQ                        *)
1491    LZSFEND; goto 0;
1492
1493
1494   0: CASE GOTOVAR OF            (* STREUSPRUNG ZUR SIMULATION VON DYNAMISCHEN *)
1495                                 (* SPRUENGEN IN PASCAL. GOTOVAR ENTHAELT DIE  *)
1496                                 (* DIE ANZUSPRINGENDE MARKE .                 *)
```

B.2 Der Compiler (anacomp.pas)

```
 1
 2  (* ANALYSE UND UEBERSETZUNG VON LISP/N-PROGRAMMEN IN PASCAL                *)
 3  (* METHODE    : REKURSIVER ABSTIEG.                                       *)
 4  (* ZIELSPRACHE: PASCAL                                                    *)
 5  (* VERSION 3.1                                          STAND: 10-21-85  *)
 6  (* Uwe Honschopp (siehe [Ho83])                                           *)
 7  (*                                                                        *)
 8  (* TP-Version 1.65 , Guido Wessendorf                   Stand: 07-24-92  *)
 9
10
11  PROGRAM ANACOMP
12   (input,OUTPUT,LISPFILE,LISTFILE,ZWCODE,PASFIL1,PASFIL2,STANDC,ZPROG,gmzwcode);
13
14
15  (* ****************** MARKENDEKLARATION ********************************** *)
16
17  LABEL
18       8888;                                    (* AUSSPRUNG BEI LEEREM PROGR*)
19
20
21
22  (* ******************  KONSTANTENDEKLARATION  ***************************** *)
23
24  CONST
25       VERSION    = ' 3.1  VOM 21.10.85';      (* VERSIONSMELDUNG           *)
26       tpversion  = '1.65 vom 24.7.92  ';      (* Versionsmeldung TP-Version*)
27       SYMLENG    = 8;                         (* MAXIMALE SYMLAENGE        *)
28       EMPTYSTRING= '        ';                (* LEERES SYM                *)
29       MAXNEST    = 5;                         (* MAX. FUNKT.SCHACHTEL.TIEFE*)
30       ANZSTLLN   = 4;                         (* ANZAHL DER STELLEN IM     *)
31                                               (* INDEX EINES IDENTIFIKATORS*)
32                                               (* PRO SCHACHTELUNGSTIEFE;   *)
33                                               (* MAX. ZAHL PAR. FUNKTIONEN:*)
34                                               (*  (10^ANZSTLLN)-1          *)
35       INDLENGTH  =20;                         (* MAXNEST * ANZSTLLN        *)
36       PACKLENG   =29;                         (* SYMLENG+INDLENGTH+1       *)
37
38
39
40  (* ******************  TYPDEKLARATION   ********************************** *)
41  TYPE
42       strin      =ARRAY[1..SYMLENG]  OF CHAR;    (* SYMBOLE                  *)
43       NUMB       =ARRAY[1..INDLENGTH]  OF CHAR;  (* NUMMERN FUER IDENTIFIKAT. *)
44       PACKAR     =ARRAY[1..PACKLENG] OF CHAR;    (* IDENTIFIKATOREN M. NUMMERN*)
45       EOZ        =1..2;                          (* TYPANGABE EINS O. ZWEI    *)
46       POMIDFREC  =^MIDFREC;                      (* ZEIGER AUF MODEIDF.RECORDS*)
47       POIDF      =^IDF;                          (* ZEIGER AUF IDF.RECORDS    *)
48       POMR       =^MODEREC;                      (* ZEIGER AUF MODERECORDS    *)
49       SCHACHTINT =-1..MAXNEST;                   (* SCHACHTELUNGSTIEFE        *)
50
51       IDF=RECORD                                 (* INFORMATION UEB. FUNKTION-*)
52                                                  (* BZW. PARAMETERIDENTIFIKAT.*)
53                NAMENR        : PACKAR;          (* AUS NAME UND NR. GENERIER-*)
54                                                  (* TER NEUER NAME            *)
55                MODEIDF       : strin;           (* MODE DES IDENTIFIKATORS   *)
56                MODEP         : POMR  ;          (* ANFANG D. ZUGEH.MODELISTE *)
57                RESULTMOD     : strin ;          (* RESULTMODE DES MODES      *)
58                STATNIV       : SCHACHTINT;      (* STATISCHES NIVEAU DES IDF.*)
59                LLINK,RLINK   : POIDF;           (* NACHFOLGER IM IDF.BAUM    *)
60          CASE   TYP          : EOZ        OF    (* 1=FUNKTIONSIDF 2=PARAMETER*)
61           1:  ( STARTADR     : longint);        (* STARTADRESSE DER FUNKTION *)
62           2:  ( RELADD       : longint);        (* RELATIVADRESSE D. PARAMET.*)
63
64              END;
65
66
67       MIDFREC=RECORD                             (* INFORMATION UEBER MODEIDF.*)
68
69                 NAME         : strin;           (* NAME DES IDENTIFIKATORS   *)
70                 DECL         : 0..1 ;           (* 0=^FORWARD-DECKLARATION   *)
71                 MODEP        : POMR;            (* ANFANG DER MODE-LISTE     *)
72                 RESULTMOD    : strin;           (* RESULTMODE DES MODES      *)
```

```
 73                LLINK,RLINK  : POMIDFREC         (* NACHFOLGER IM MODEBAUM    *)
 74             END;
 75
 76
 77        MODEREC=RECORD                           (* ELEMENTE DER MODELISTE    *)
 78
 79                MODEIDF      : strin;            (* MODENAME                  *)
 80                NEXTP        : POMR              (* NACHFOLGER IN MODE-LISTE  *)
 81              END;
 82
 83
 84        ABSRELTYP = (ABSOLUT,RELATIV);           (* SCHALTER IN CHANGEIND     *)
 85
 86
 87   (* ***************  VARIABLENDEKLARATION  ******************************** *)
 88
 89   VAR
 90
 91        CHZ          : longint;                  (* SPALTENZAEHLER F. FEHLER  *)
 92        CHZH         : longint;                  (* WIE CHZ/FUER POS. B. EOLN *)
 93        DIGIT        : SET OF CHAR;              (* MENGE DER ZIFFERN         *)
 94        ENDSYMS      : SET OF CHAR;              (* MENGE DER BEGRENZ. SYMBOLE*)
 95        HNUMBER      : NUMB;                     (* HILFSZAEHLER FUER NUMBER  *)
 96        I            : longint;                  (* HILFSVARIABLE/LAUFINDEX   *)
 97        ISSYM        : strin;                    (* JEWEILIGES SYMBOL         *)
 98        ISNEXTSYM    : strin;                    (* JEW. NAECHSTES SYMBOL     *)
 99        KOMMENTARTEST: BOOLEAN;                  (* KLAMMER, DIE KEINEN       *)
100                                                 (* KOMMENTAR EINLEITET UNTER-*)
101                                                 (* SUCHT                     *)
102        KONSTZ       : longint;                  (* PROGRAMMKONST. ZAEHLER    *)
103        LABELNR      : longint;                  (* NAECHSTE FREIE MARKE      *)
104        LETDIG       : SET OF CHAR;              (* MENGE D. ZIFFERN & BUCHST.*)
105        LETTER       : SET OF CHAR;              (* MENGE DER BUCHSTABEN      *)
106        LETTERBLANK  : SET OF CHAR;              (* MENGE DER BUCHST. & BLANK *)
107        LISPFILE     : TEXT;                     (* QUELLCODE                 *)
108        LISTFILE     : TEXT;                     (* ENTH. ANALYSE-PROYOKOLL   *)
109        MIDFRECP     : POMIDFREC;                (* AUFZUFUELLENDES MODE-REC. *)
110        MWURZELP     : POMIDFREC;                (* ZEIGER A. MODE-BAUM-WURZEL*)
111        NEXTCHAR     : CHAR;                     (* JEWEILS GELESENE ZEICHEN  *)
112        NUMBER       : NUMB;                     (* ZAEHLER F. VAR.-NUMMERIER.*)
113        OUTCOUNT     : longint;                  (* ZAEHLER FUER PROTOKOLL    *)
114        PASFIL1      : TEXT ;                    (* ERZEUGTE INHEAP-BEFEHLE   *)
115        PASFIL2      : TEXT ;                    (* ERZEUGTER PASCAL-ZIELCODE *)
116        SCHACHT      : longint;                  (* JEW. SCHACHTELUNGSTIEFE   *)
117        STANDC       : TEXT;                     (* ENTHAELT STANDARD(LZS)CODE*)
118        SYMZ         : longint;                  (* ZAEHLER FUER ZWCODE       *)
119        WURZELP      : POIDF;                    (* ZEIGER AUF IDF-BAUM-WURZEL*)
120        ZPROG        : TEXT;                     (* ENTHAELT ZIELPROGRAMM     *)
121        ZWCODE       : TEXT;                     (* ZWISCHENCODE NACH 1. LAUF *)
122
123        gmniv,dpniv  : longint;                  (* GMARK/DckPara-Markierung  *)
124        gmzwcode     : text;                     (* Zwischencode nach GMARK   *)
125        zeile        : string;                   (* Fuer Ein/Ausgabe Funktion *)
126
127
128   (* ************************************************************************  *)
129   (* *************  GLOBAL BENUTZTE PROZEDUREN/FUNKTIONEN ******************  *)
130   (* ************************************************************************  *)
131
132
133   procedure ende;      (* Schliessen aller Files und Ausstieg aus dem Programm *)
134     begin
135       close(zprog); close(zwcode); close(pasfil1); close(pasfil2);
136       close(listfile); close(standc); halt
137     end;
138
139
140   PROCEDURE FEHLER(FEHLERNUMMER:longint);
141   (* DIE PROZEDUR ERZEUGT FEHLERMELDUNGEN AUF DEM BILDSCHIRM UND DER DATEI     *)
142   (* LISTFILE.NACH EINEM FEHLER WIRD DIE ANALYSE/UEBERSETZUNG ABGEBROCHEN.     *)
143
144   VAR       J   : longint;                       (* LAUFINDEX                 *)
145             B   :  BOOLEAN;                      (* HV FUER ABBRUCH-KOMMANDO  *)
146
147
```

```
148   BEGIN
149    WRITELN(' NO EXECUTABLE CODE!  ');
150    WRITELN(' ISSYM=''',ISSYM:SYMLENG,'''  ISNEXTSYM=''',ISNEXTSYM:SYMLENG,'''');
151    WRITELN(' FEHLERPROTOKOLL AUF FILE LISTFILE!!! ');
152    WRITELN(LISTFILE,' ');
153    FOR J:=0 TO (CHZH-8) DO WRITE(LISTFILE,' ');   (* VORSCHUB  *)
154    CHZ:=0;
155    WRITE(LISTFILE,'@');                          (* KENNZEICHNUNG FEHLERSTELLE*)
156    CASE FEHLERNUMMER OF
157
158        01 : WRITELN(LISTFILE,' "BEGIN" EXPECTED ');
159        02 : WRITELN(LISTFILE,' IDENTIFIER EXPECTED');
160        03 : WRITELN(LISTFILE,' "=" EXPECTED ');
161        04 : WRITELN(LISTFILE,' "FUNC" EXPECTED');
162        05 : WRITELN(LISTFILE,' "(" EXPECTED ');
163        06 : WRITELN(LISTFILE,' ";" EXPECTED ');
164        07 : WRITELN(LISTFILE,' ILLEGAL RESULT-MODE ');
165        08 : WRITELN(LISTFILE,' ILLEGAL MODE ');
166        09 : WRITELN(LISTFILE,' ")" EXPECTED ');
167        10 : WRITELN(LISTFILE,' IDF. BESTEHT AUS MEHR ALS ',SYMLENG:2,' ZEICHEN');
168        11 : WRITELN(LISTFILE,' ":" EXPECTED ');
169        12 : WRITELN(LISTFILE,' "," EXPECTED ');
170        13 : WRITELN(LISTFILE,' "{" EXPECTED ');
171        14 : WRITELN(LISTFILE,' "}" EXPECTED ') ;
172        15 : WRITELN(LISTFILE,' IDENTIFIER IS STANDARDIDENTIFIER');
173        16 : WRITELN(LISTFILE,' IDENTIFIER DECLARED TWICE');
174        17 : WRITELN(LISTFILE,' MODE OF BOOLEXPR <> "S-EXPR" ');
175        18 : WRITELN(LISTFILE,' IDENTIFIER NOT DECLARED');
176        19 : WRITELN(LISTFILE,' MODE DECLARED OTHERWISE ');
177        20 : WRITELN(LISTFILE,' EOF - NOT YET EXPECTED');
178        21 : WRITELN(LISTFILE,' RESULTMODE DECLARED OTHERWISE');
179        22 : WRITELN(LISTFILE,' TOO MANY NESTED FUNCTIONS     ');
180        23 : WRITELN(LISTFILE,' "}" NOT YET EXPECTED');
181        24 : WRITELN(LISTFILE,' TOO MANY PARALLEL FUNCTIONS ');
182        25 : WRITELN(LISTFILE,  SCHACHT-1:3,' "]" MISSING IN PROGRAMM ');
183        26 : WRITELN(LISTFILE,' "END" EXPECTED ');
184        27 : WRITELN(LISTFILE,' ILLEGAL SYMBOL IN S-EXPR-EXPR');
185        28 : WRITELN(LISTFILE,' ILLEGAL SYMBOL IN BOOLEXPR');
186        29 : WRITELN(LISTFILE,' "END" IS NOT LAST SYMBOL  ');
187        30 : WRITELN(LISTFILE,' ACT. PAR. MODE DOES NOT AGREE WITH DECLARATION');
188        31 : WRITELN(LISTFILE,' "THEN" EXPECTED ');
189        32 : WRITELN(LISTFILE,' "ELSE" EXPECTED ');
190        33 : WRITELN(LISTFILE,' TYPE OF ELSE-PART DOES NOT AGREE WITH THEN-PART');
191        34 : WRITELN(LISTFILE,' ILLEGAL SYMBOL IN THEN-PART ');
192        35 : WRITELN(LISTFILE,' ILLEGAL SYMBOL IN ELSE-PART ') ;
193        36 : WRITELN(LISTFILE,' "FI" EXPECTED ');
194        37 : WRITELN(LISTFILE,' RESULTMODE IS NOT S-EXPR ');
195        38 : WRITELN(LISTFILE,' NUMBER OF ACT. PAR. DOES NOT AGREE WITH DECL.') ;
196        39 : WRITELN(LISTFILE,' FUNC.MODE IS VOID AND EXPRESSION IS NOT EMPTY');
197        40 : WRITELN(LISTFILE,' MODE OF FUNC.ST. IN BOOL.EXPR. IS NOT S-EXPR   ');
198        41 : WRITELN(LISTFILE,' SYMBOL IS NOT ATOMIC S-EXPR ');
199        42 : WRITELN(LISTFILE,' RESULTMODE NOT DECLARED');
200        43 : WRITELN(LISTFILE,' RESULTMODE S-EXPR - NO ACT. PAR. LIST EXPECTED');
201        44 : WRITELN(LISTFILE,' RESULTMODE IS VOID - NO ACT. PAR. LIST EXPECTED');
202        45 : WRITELN(LISTFILE,' FUNC.MODE IS NOT S-EXPR ');
203        47 : WRITELN(LISTFILE,' MODE OF EXPRESSION<>RESULTMODE OF FUNCTION  ');
204        48 : WRITELN(LISTFILE,' MODE IS NOT S-EXPR');
205        49 : WRITELN(LISTFILE,' EXPRESSION IS EMPTY AND RESULTMODE<>VOID ');
206        51 : WRITELN(LISTFILE,' ILLEGAL DOT            ');
207        52 : WRITELN(LISTFILE,' FUNC.EXPR. IS NOT ALLOWED IN MAIN PROG.');
208        53 : WRITELN(LISTFILE,' MODE OF MAINPROGRAM <> S-EXPR / VOID ');
209        54 : WRITELN(LISTFILE,' MODE IS NOT S-EXPR OR VOID          ');
210        55 : writeln(listfile,' Syntaxfehler: ), }, THEN oder ELSE fehlt... ')
211      ELSE  WRITELN(LISTFILE,' ERRORNUMBER ',FEHLERNUMMER:3,' NOT FOUND');
212
213    END;  (* OF CASE *)
214
215
216    WRITELN(LISTFILE,' ');
217    WRITELN(LISTFILE,'**********  ANALYSE WURDE ABGEBROCHEN! **********');
218
219    ende                                          (* ABBRUCH VON ANACOMP.      *)
220
221   END;  (* OF PROCEDURE FEHLER *)
222
```

```
223
224
225   PROCEDURE CHANGEIND(VAR IND : NUMB;       (* DER ZU AENDERNDE INDEX         *)
226                       STELLE : longint;     (* DIE ZU AENDERNDE STELLE IN IND *)
227                       AENDERUNG : longint;  (* SIEHE ABSREL                   *)
228                       ABSREL : ABSRELTYP);  (* SCHALTER:                      *)
229                                             (* = ABSOLUT:                     *)
230                                             (*   SETZE DIE DURCH AENDERUNG    *)
231                                             (*   GEGEBENE ZAHL AN STELLE;     *)
232                                             (* = RELATIV:                     *)
233                                             (*   SETZE DIE SUMME AUS          *)
234                                             (*   AENDERUNG UND DER AN STELLE  *)
235                                             (*   STEHENDEN ZAHL AN STELLE     *)
236
237   VAR
238     ZAHL   : longint;          (* longint-REPRAESENTATION DER AN STELLE IN    *)
239                                (* IND BEFINDLICHEN ZAHL                       *)
240     FAKTOR : longint;          (* HILFSGROESSE BEIM UMWANDELN NACH longint    *)
241     I      : longint;          (* LAUFVARIABLE                                *)
242
243   BEGIN
244     IF ABSREL = RELATIV
245      THEN
246        BEGIN
247          ZAHL := 0;
248          FAKTOR := 1;
249          FOR I := ANZSTLLN*STELLE DOWNTO ANZSTLLN*(STELLE-1) + 1
250           DO
251             BEGIN
252               ZAHL := ZAHL + (ORD(IND[I]) - ORD('0')) * FAKTOR;
253               FAKTOR := FAKTOR * 10
254             END;
255          ZAHL := ZAHL + AENDERUNG
256        END
257      ELSE
258        ZAHL := AENDERUNG
259   (*FI*);
260
261    FOR I := ANZSTLLN*STELLE DOWNTO ANZSTLLN*(STELLE-1) + 1
262     DO
263       BEGIN
264         IND[I] := CHR(ZAHL MOD 10 + ORD('0'));
265         ZAHL := ZAHL DIV 10
266       END
267   (*OD*);
268
269    IF ZAHL > 0 THEN FEHLER(24)
270
271   END; (* OF PROCEDURE CHANGEIND *)
272
273
274
275  FUNCTION IDENTIFIER : BOOLEAN;
276  (* DIE FUNKTION STELLT FEST OB DAS VORLIEGENDE SYMBOL (ISSYM) EIN            *)
277  (* NICHTSTANDARDIDENTIFIKATOR IST.                                          *)
278
279    VAR I : longint;                              (* LAUFVARIABLE            *)
280        IS: strin;                                (* GLEICH ISSYM            *)
281
282    BEGIN
283    (*  <IDENTIFIER>::= <LETTER> [<LETTER./<DIGIT>]   *)
284
285     IS:=ISSYM;
286     I:=1;
287     IDENTIFIER:=TRUE;
288                                                  (* STANDARDIDENTIFIKATORTEST *)
289     IF (IS='BEGIN   ') OR (IS='END     ') OR (IS='IF      ')  THEN FEHLER(15);
290     IF (IS='ELSE    ') OR (IS='FI      ') OR (IS='F       ')  THEN FEHLER(15);
291     IF (IS='IN      ') OR (IS='T       ') OR (IS='ATOM    ')  THEN FEHLER(15);
292     IF (IS='FUNC    ') OR (IS='MODE    ') OR (IS='VOID    ')  THEN FEHLER(15);
293     IF (IS='CONS    ') OR (IS='EQ      ') OR (IS='CDR     ')  THEN FEHLER(15);
294     IF (IS='CAR     ') OR (IS='THEN    ') OR (IS='NIL     ')  THEN FEHLER(15);
295     if (is='PCONS_  ')                                        then fehler(15);
296
297     IF NOT(IS[1] IN LETTER) THEN IDENTIFIER:=FALSE
```

```
298                            ELSE                  (*1. ZEICHEN IST BUCHSTABE   *)
299                             BEGIN
300                              I:=2;
301                              REPEAT
302                               IF NOT(IS[I] IN LETDIG) THEN IDENTIFIER:=FALSE;
303                               I:=I+1
304                              UNTIL I=(SYMLENG+1);
305                             END;
306
307    END;  (* OF FUNCTION IDENTIFIER *)
308
309
310
311   PROCEDURE NEXTSYM(VAR ISSYM,ISNEXTSYM:strin;VAR NEXTCHAR:CHAR;CODE:BOOLEAN);
312   (* DIE PROZEDUR LIESST JEWEILS EIN SYMBOL VORRAUS UND LEGT ES IN ISNEXTSYM   *)
313   (* AB.DAS ZU BEARBEITENDE SYMBOL STEHT DANN IN ISSYM.FALLS CODE=TRUE WIRD    *)
314   (* DIESES SYMBOL(ISSYM) AUF DEN FILE ZWISCHENCODE(ZWCODE) GESCHRIEBEN .      *)
315   (* DIE VARIABLE NEXTCHAR ENTHAELT NACH DEM EINLESEN VON ISNEXTSYM DAS AUF    *)
316   (* ISNEXTSYM FOLGENDE ZEICHEN,SOFERN NICHT EOF VON LISPFILE ERREICHT IST.    *)
317
318   LABEL
319         1;                                      (* AUSSTIEG BEI VON QUELLCODE*)
320                                                 (* ODER PROGRAMMENDE         *)
321   VAR
322      I            :  longint;                   (* LAUFINDEX                 *)
323      EOLNINF      :  BOOLEAN;                   (* TRUE,FALLS EOLN(LISPFILE) *)
324
325
326    PROCEDURE RD(VAR EOLNINF:BOOLEAN);
327    (* DIE PROZEDUR LIESST UND PROTOKOLLIERT DAS JEW. NAECHSTE ZEICHEN.         *)
328    (* EOLNINF WIRD TRUE ,FALLS EOLN(LISPFILE) NACH DEM EINLESEN.               *)
329
330     BEGIN
331
332      EOLNINF := FALSE;
333      IF EOF(LISPFILE) THEN NEXTCHAR:=' '          (* SONDERFALL EOF(LISPFILE)  *)
334              else if eoln(lispfile)               (* Ueberlesen vom Zeilenende *)
335                     then begin readln(lispfile); nextchar:=' '; eolninf:=true end
336                     ELSE BEGIN
337                           READ(LISPFILE,NEXTCHAR);
338                                                  (*UMWANDLUNG KLEIN IN GROSSB.*)
339                           IF NEXTCHAR IN ['a'..'i','j'..'r','s'..'z'] THEN
340                              NEXTCHAR:=CHR(ORD(NEXTCHAR)-32);   (* ascii-Chars *)
341                           WRITE(LISTFILE,NEXTCHAR);
342                           IF EOLN(LISPFILE) THEN BEGIN
343                                                  EOLNINF:=TRUE;
344                                                  readln(lispfile); (* statt get *)
345                                                  CHZH:=CHZ;
346                                                  CHZ:=0;
347                                                  WRITELN(LISTFILE,' ');
348                                                 END
349                                           ELSE
350                                                BEGIN
351                                                 CHZ:=CHZ+1;
352                                                 CHZH:=CHZ
353                                                END;
354                           IF EOF(LISPFILE) THEN
355                                           BEGIN
356                                            ISNEXTSYM[I+1]:=NEXTCHAR;
357                                           END;
358                        END;
359
360     END;  (*  OF PROCEDURE RD  *)
361
362
363    PROCEDURE BLANK;
364    (* DIE PROZEDUR UEBERLIESST BLANKS                                          *)
365
366     BEGIN
367
368      IF NEXTCHAR=' ' THEN
369                       REPEAT
370                        RD(EOLNINF);
371                       UNTIL (NEXTCHAR<>' ') OR (EOF(LISPFILE));
372
```

```
373     END;   (*  OF PROCEDURE BLANK *)
374
375
376
377    PROCEDURE COMMENT;
378     (* BENUTZTE GLOBALE VARIABLEN :   KOMMENTARTEST : BOOLEAN   *)
379     (*                                NEXTCHAR      : CHAR      *)
380     (*                                EOLNINF       : BOOLEAN   *)
381     (*                                LISPFILE      : TEXT      *)
382
383     VAR
384       FERTIG : BOOLEAN;
385
386     BEGIN
387       KOMMENTARTEST := FALSE;
388       IF (NEXTCHAR = '(') AND NOT EOLNINF
389        THEN
390          BEGIN
391            RD(EOLNINF);
392            IF NEXTCHAR = '*'
393             THEN
394               BEGIN (* SKIPCOMMENT *)
395                 FERTIG := FALSE;
396                 REPEAT
397                   RD(EOLNINF);
398                   IF (NEXTCHAR = '*') AND NOT EOLNINF
399                    THEN
400                      BEGIN
401                        RD(EOLNINF);
402                        FERTIG := NEXTCHAR = ')'
403                      END
404                 UNTIL FERTIG OR EOF(LISPFILE);
405                 RD(EOLNINF)
406               END
407             ELSE
408               KOMMENTARTEST := TRUE
409         (*FI*)
410         END
411     (*FI*)
412     END; (* OF PROCEDURE COMMENT *)
413
414
415
416    BEGIN                                          (* OF PROCEDURE NEXTSYM      *)
417
418     EOLNINF := FALSE;
419     ISSYM:=ISNEXTSYM;
420     ISNEXTSYM:=EMPTYSTRING;
421     IF ISSYM='END     '                            (* SONDERFALL PROGRAMMENDE   *)
422      THEN BEGIN
423            IF NOT(EOF(LISPFILE)) THEN FEHLER(29); (* "END" NICHT LETZTES SYMB. *)
424            ISNEXTSYM:='END     ';                 (* REDUNDANT                 *)
425            GOTO 1;                                (* KEIN WEITERES EINLESEN    *)
426          END;
427     IF EOF(LISPFILE) THEN GOTO 1;                 (* KEIN WEITERER CODE VORH.  *)
428     I:=1;
429     REPEAT
430       BLANK;
431       REPEAT
432         COMMENT
433       UNTIL (NEXTCHAR <> '(') OR KOMMENTARTEST OR EOF(LISPFILE)
434     UNTIL (NEXTCHAR <> ' ') OR KOMMENTARTEST OR EOF(LISPFILE);
435     IF KOMMENTARTEST
436      THEN (* SCHON EIN ZEICHEN ZU WEIT GELESEN *)
437        BEGIN
438          ISNEXTSYM[1] := '(';
439          I := 2
440        END
441      ELSE
442        IF NEXTCHAR IN ENDSYMS THEN
443              BEGIN
444               ISNEXTSYM[1] := NEXTCHAR;
445               I := 2;
446               RD(EOLNINF)
447              END
```

```
448             ELSE
449              BEGIN
450               REPEAT
451                ISNEXTSYM[I]:=NEXTCHAR;
452                RD(EOLNINF);
453                IF I<SYMLENG THEN I:=I+1
454                             ELSE WRITELN(LISTFILE,' FEHLER-LAENGE ');
455                                                  (* REST DES SYMBOLS IGNORIERT*)
456                                                  (* ALT. FEHLER(10) -->ABBRUCH*)
457              UNTIL EOLNINF OR (NEXTCHAR IN ENDSYMS) ;
458             END;
459    IF EOLNINF AND NOT(NEXTCHAR IN ENDSYMS) and not(isnextsym[1] in endsyms)
460      THEN
461        BEGIN
462          ISNEXTSYM[I]:=NEXTCHAR;
463          RD(EOLNINF)
464        END;
465    IF CODE THEN
466              BEGIN
467               IF SYMZ=0 THEN WRITE(ZWCODE,ISSYM)
468                         ELSE WRITE(ZWCODE,' ',ISSYM);
469               SYMZ:=SYMZ+1;
470              END;
471    IF SYMZ>7 THEN                               (*7 SYMBOLE IN ZWCODE-ZEILE  *)
472               BEGIN
473                WRITELN(ZWCODE);
474                SYMZ:=0
475               END;
476   1:IF EOF(LISPFILE) AND (ISNEXTSYM<>'END     ') THEN FEHLER(20)
477
478    END;   (* OF PROCEDURE NEXTSYM *)
479
480
481
482   PROCEDURE NEXTCODESYM( VAR ISSYM,ISNEXTSYM : strin);
483   (* DIE PROZEDUR LIESST JEW. EIN SYMBOL AUF ZWCODE VORRAUS. DIESES SYMBOL DER *)
484   (* FESTEN LAENGE SYMLENG (I.A. 8 - VGL. KONSTANTENDEKLARATION) WIRD IN DER   *)
485   (* VARIABLEN ISNEXTSYM ABGELEGT.DIE VARIABLE ISSYM ENTHAELT DANN DAS JEW. ZU *)
486   (* BEARBEITENDE SYMBOL.                                                     *)
487
488
489   VAR
490      I            :  longint;                  (* LAUFINDEX                 *)
491      NEXTCHAR     :  CHAR;                     (* JEW. GELESENES ZEICHEN    *)
492
493   BEGIN
494     ISSYM:=ISNEXTSYM;
495     ISNEXTSYM:=EMPTYSTRING;
496     if eoln(zwcode) then readln(zwcode)          (* Lesen CR bei Zeilenende   *)
497                     else READ(ZWCODE,NEXTCHAR);  (*LESEN D. TRENNSYMBOLS      *)
498     if not(eof(zwcode)) then
499       begin
500         FOR I:=1 TO SYMLENG DO
501                              BEGIN
502                               READ(ZWCODE,NEXTCHAR);
503                               ISNEXTSYM[I]:=NEXTCHAR;
504                              END;
505         WRITE(LISTFILE,isnextsym);      (* PROTOKOLLIERUNG DES gelesenen SYMB. *)
506         IF OUTCOUNT<9 THEN                       (* 10 SYMBOLE JE LISTINGZEILE*)
507                        OUTCOUNT:=OUTCOUNT+1
508                       ELSE
509                        BEGIN
510                         OUTCOUNT:=0;
511                         WRITELN(LISTFILE,' ')
512                        END
513       end
514     END;   (* OF PROCEDURE NEXTCODESYM *)
515
516
517   PROCEDURE INIT;
518   (* DIE PROZEDUR SORGT FUER VORBESETZUNGEN VOR DEM AUFRUF DES ERSTEN LAUF.    *)
519
520   VAR         I       :     longint;           (* LAUFVARIABLE              *)
521
522   BEGIN
```

```
523
524   assign(lispfile,'lispfile.lsp');         (* Zuordnungen zu externen Dateien *)
525   assign(zwcode,'zwcode.dat'); assign(listfile,'listfile.dat');
526   assign(pasfil1,'pasfil1.dat'); assign(pasfil2,'pasfil2.dat');
527   assign(zprog,'zprog.pas'); assign(standc,'standc.dat');
528
529   RESET(LISPFILE);                                (* ENTHAELT DEN QUELLCODE    *)
530   REWRITE(LISTFILE); REWRITE(ZWCODE);             (* F. PROTOKOLL/ZWISCHENCODE *)
531   REWRITE(PASFIL1); REWRITE(PASFIL2);             (* F. ERZEUGTEN PASCAL-CODE  *)
532   rewrite(zprog); reset(standc);
533   WRITELN(LISTFILE,'FEHLERPROTOKOLL: '); writeln(listfile);
534   WRITEln(ZWCODE);
535   CHZ:=0; symz:=0;                                (* 0 ZEICHEN GELESEN         *)
536   DIGIT:=['0'..'9'];
537   ENDSYMS:=[' ',',',':','(',')',';','=','{','}','.','"'];
538   ISNEXTSYM:=EMPTYSTRING;
539   ISSYM:=EMPTYSTRING;
540   KONSTZ:=2;                                      (* KONST.: 0=^NIL,1=^T,2=^F  *)
541                                                   (* AB PLATZ 3 FREIE PLAETZE  *)
542   LABELNR:=3;                                     (*0..2:STANDARDMARKEN IM CODE*)
543   LETTER:=['A'..'I']+['J'..'R']+['S'..'Z'];
544   LETTERBLANK:=LETTER+[' '];
545   LETDIG:=LETTERBLANK+DIGIT;                      (* LETTERBLANK,DA SYMBOLE MIT*)
546                                                   (* BLANKS AUFGEFUELLT WERDEN.*)
547   MIDFRECP:=NIL;
548   MWURZELP:=NIL;
549   NEXTCHAR:=' ';
550   FOR I := 1 TO INDLENGTH DO NUMBER[I] := ' ' (*OD*);
551   FOR I := 1 TO MAXNEST DO CHANGEIND(HNUMBER,I,1,ABSOLUT) (*OD*);
552   CHANGEIND(NUMBER,1,1,ABSOLUT);
553   OUTCOUNT := 0;
554   SCHACHT := 1;                                 (* STAT. NIVEAU D. HAUPTPROGR. *)
555   WURZELP := NIL;
556
557
558   NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);        (* 1.SYMBOL IN ISNEXTSYM     *)
559   NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);        (* 1.SYMBOL IN ISSYM         *)
560
561
562 END;   (* OF PROCEDURE INIT  *)
563
564
565
566 PROCEDURE INIT2;
567 (* DIE PROZEDUR SORGT FUER VORBESETZUNGEN VOR DER UEBERSETZUNG DES           *)
568 (* ZWISCHENCODES.                                                            *)
569
570 VAR               I    :   longint;             (* LAUFVARIABLE              *)
571
572
573 BEGIN
574
575   WRITELN(ZWCODE);
576   RESET(ZWCODE);                                  (* ENTH. ZWISCHENCODE        *)
577   REWRITE(LISTFILE);
578   WRITELN(LISTFILE,'UEBERSETZUNG DES ZWISCHENCODES ');
579   WRITELN(LISTFILE,'FEHLERPROTOKOLL: '); writeln(listfile); (* Leerzeile einf.*)
580   ISNEXTSYM:=EMPTYSTRING;
581   ISSYM:=EMPTYSTRING;
582   FOR I := 1 TO INDLENGTH DO NUMBER[I] := ' ' (*OD*);
583   FOR I := 1 TO MAXNEST DO CHANGEIND(HNUMBER,I,1,ABSOLUT) (*OD*);
584   CHANGEIND(NUMBER,1,1,ABSOLUT);
585   NEXTCODESYM(ISSYM,ISNEXTSYM);                   (* 1.ZWCODESYM. IN ISNEXTSYM *)
586   SCHACHT := 1; outcount:=0
587
588
589 END;   (* OF PROCEDURE INIT2 *)
590
591
592
593 (* ************************************************************************)
594 (* ****  1. LAUF:ANALYSE DES MODEDEKLARATIONSTEILS UND DES FUNKTIONS-    *****)
595 (* ****          DEKLARATIONSTEILS OHNE DIE AUSDRUECKE IM RUMPF.         *****)
596 (* ****          ERZEUGUNG EINES ZWISCHENCODES AUF DEM FILE ZWCODE FUER  *****)
597 (* ****          DIE UEBERSETZUNG DES FUNKTIONSDEKLARATIONTEIL IM 2.LAUF *****)
```

```
598  (* ************************************************************************)
599
600
601
602
603  (* ************************************************************************  *)
604  (* ***********   M O D E    D E C L A R A T I O N    P A R T   ***********  *)
605  (* *****  ANALYSE UND ABLAGE DER MODEIDENTIFIKATOREN IM MODEBAUM  *********  *)
606  (* ************************************************************************  *)
607
608
609
610
611
612  PROCEDURE MODEDECLPART;
613
614
615   PROCEDURE NEWMIDFREC(NAME:strin;VAR MIDFRECP : POMIDFREC);
616   (* DIE PROZEDUR NEWMIDFREC LEGT BEI DEKLARATION EINES MODEIDENTIFIKATORS    *)
617   (* EIN RECORD VOM TYP MIDFREC AN.                                           *)
618   (* DER PARAMETER NAME ENTHAELT DEN NAMEN DES IDENTIFIKATORS.DER PARAMETER   *)
619   (* HAELT AM ENDE DER PROZEDUR DEN VERWEIS AUF DAS NEU ANGELEGTE MODERECORD, *)
620   (* UM SPAETER DIE MODELISTE AUFZUFUELLEN.                                   *)
621   (* DIE MODERECORDS WERDEN MIT HILFE DER PROZEDUR NEWLINK IN LEXIKOGRAPH.    *)
622   (* ORDNUNG IN EINEM BAUM ABGESPEICHERT.AUF DIE WURZEL DIESES  BAUMES ZEIGT  *)
623   (* STETS MWURZELP.                                                          *)
624
625
626    PROCEDURE NEWLINK(NAME :strin;VAR MIDFRECP:POMIDFREC; O:POMIDFREC);
627    (* DIE PROZEDUR NEWLINK ORDNET AN "RICHTIGER" STELLE IM MODEBAUM EIN.      *)
628    (* NAME,MIDFRECP WIE IN NEWMIDFREC.O ZEIGT AUF JEW. RECORD IM BAUM         *)
629    (* WAEHREND SUCHENS NACH STELLE, WO NEUES RECORD EINGETRAGEN WERDEN SOLL.  *)
630
631
632    BEGIN
633     IF NAME<>O^.NAME THEN
634       IF NAME > O^.NAME THEN
635          IF O^.RLINK=NIL THEN
636                          BEGIN
637                           NEW(O^.RLINK);
638                           MIDFRECP:=O^.RLINK;
639                           O^.RLINK^.NAME:=NAME;
640                           O^.RLINK^.DECL:=1;
641                           O^.RLINK^.MODEP:=NIL;
642                           O^.RLINK^.RLINK:=NIL;
643                           O^.RLINK^.LLINK:=NIL;
644                          END
645                         ELSE NEWLINK(NAME,MIDFRECP,O^.RLINK)
646                        ELSE
647                         IF O^.LLINK=NIL THEN
648                             BEGIN
649                              NEW(O^.LLINK);
650                              MIDFRECP:=O^.LLINK;
651                              O^.LLINK^.NAME:=NAME;
652                              O^.LLINK^.DECL:=1;
653                              O^.LLINK^.MODEP:=NIL;
654                              O^.LLINK^.LLINK:=NIL;
655                              O^.LLINK^.RLINK:=NIL;
656                             END
657                              ELSE NEWLINK(NAME,MIDFRECP,O^.LLINK)
658                     ELSE
659                      IF O^.DECL=0 THEN          (* NAME BEREITS FORWARD DEKL.*)
660                                   BEGIN
661                                    O^.DECL:=1;
662                                    MIDFRECP:=O;
663                                   END
664                                  ELSE          (* NAME BEREITS DEKLARIERT   *)
665                                   FEHLER(16);
666
667    END; (*OF PROCEDURE NEWLINK *)
668
669   BEGIN                                        (* OF NEWMIDFREC             *)
670
671    IF MWURZELP=NIL THEN
672                     BEGIN
```

```
673                     NEW(MWURZELP);
674                     MIDFRECP:=MWURZELP;
675                     MWURZELP^.NAME:=NAME;
676                     MWURZELP^.DECL:=1;
677                     MWURZELP^.MODEP:=NIL;
678                     MWURZELP^.LLINK:=NIL;
679                     MWURZELP^.RLINK:=NIL
680                    END
681                   ELSE NEWLINK(NAME,MIDFRECP,MWURZELP)
682
683    END;  (* OF PROCEDURE NEWMIDFREC  *)
684
685
686
687
688    PROCEDURE NEWMODE(MODENAME:strin);
689    (* DIE PROZEDUR ORDNET DEN MODEIDENTIFIKATOR(MODENAME)AM ENDE DER JEWEILIGEN*)
690    (* MODELISTE EIN. AUF DEN ANFANG DIESER LISTE ZEIGT MIDFRECP^.MODEP.        *)
691
692
693    VAR
694       H      :  POMR;                              (* ZEIGER IN DIE MODELISTE   *)
695       H1     :  POMR;                              (* ZEIGER AUF NEUEN RECORD   *)
696
697
698    BEGIN
699
700     H:=MIDFRECP^.MODEP;                            (* H ZEIGT AUF ANFANG MODEL. *)
701     IF H=NIL THEN                                  (* MODELISTE NOCH LEER       *)
702               BEGIN
703                NEW(H);
704                MIDFRECP^.MODEP:=H;
705                H^.MODEIDF:=MODENAME;
706                H^.NEXTP:=NIL
707               END
708              ELSE
709               BEGIN
710                WHILE H^.NEXTP<>NIL DO H:=H^.NEXTP;(* SUCHEN VOM MODELISTENENDE *)
711                NEW(H1);                            (* ANLEGEN EINES NEUEN REC.  *)
712                H^.NEXTP:=H1;
713                H1^.MODEIDF:=MODENAME;
714                H1^.NEXTP:=NIL
715               END
716
717    END; (* OF PROCEDURE NEWMODE *)
718
719
720
721
722    PROCEDURE SEARCH(NS:strin);
723    (* DIE PROZEDUR SUCHT DEN MODENAMEN(MODENAME) IM MODEBAUM.IST ER NICHT VOR- *)
724    (* HANDEN ERFOLGT EINE FORWARD-DEKLARATION.                                 *)
725
726    VAR
727        FOUND   :  BOOLEAN;                         (* =TRUE,F. MODENAME GEFUNDEN*)
728        H       :  POMIDFREC;                       (* ZEIGER IN DEN MODEBAUM    *)
729
730
731    BEGIN
732
733     FOUND:=FALSE;
734     H:=MWURZELP;                                   (* ZEIGT AUF MODEBAUMWURZEL  *)
735     WHILE NOT FOUND DO
736         BEGIN
737          IF H^.NAME=NS THEN
738                           FOUND:=TRUE
739                         ELSE
740                          IF H^.NAME>NS THEN
741                            BEGIN
742                             IF H^.LLINK=NIL THEN   (* IDENTIFIKATOR NOCH NICHT  *)
743                                 BEGIN              (* DEKL.: FORWARD-DEKLARAT.! *)
744                                  NEW(H^.LLINK);
745                                  H:=H^.LLINK;
746                                  H^.NAME:=NS;
747                                  H^.MODEP:=NIL;
```

```
748                                H^.LLINK:=NIL;
749                                H^.RLINK:=NIL;
750                                H^.DECL:=0;
751                                FOUND:=TRUE;
752                               END
753                              ELSE H:=H^.LLINK;
754                           END
755                                    ELSE        (* H^.NAME < NS              *)
756                           BEGIN
757                            IF H^.RLINK=NIL THEN   (* FORWARD-DEKLARATION       *)
758                               BEGIN
759                                NEW(H^.RLINK);
760                                H:=H^.RLINK;
761                                H^.NAME:=NS;
762                                H^.MODEP:=NIL;
763                                H^.LLINK:=NIL;
764                                H^.RLINK:=NIL;
765                                H^.DECL:=0;
766                                FOUND:=TRUE;
767                               END
768                             ELSE H:=H^.RLINK;
769                           END
770         END
771
772    END;  (* OF PROCEDURE SEARCH  *)
773
774
775
776
777    PROCEDURE DECLTEST;
778
779    (* DIE PROZEDUR DECLTEST PRUEFT AM ENDE DES MODEDECLARATIONPARTS , OB FUER  *)
780    (* ALLE "FORWARD" DEKLARIERTEN MODEIDENTIFIKATOREN EINE DEKLARATION FOLGTE. *)
781    (* WAR DIESES NICHT DER FALL ,SO WERDEN IM PROTOKOLL DIE NICHT DEKLARIERTEN *)
782    (* MODENAMEN ANGEGEBEN(FEHLER NR. 18).                                      *)
783
784
785    VAR
786           DECLERROR  :  BOOLEAN;                  (*=TRUE,FALLS DEKLAR. FEHLT  *)
787
788
789     PROCEDURE SITREE(H:POMIDFREC;VAR DECLERROR : BOOLEAN);
790     (* DIE PROZEDUR DURCHSUCHT DEN MODEBAUM.WIRD EIN NAME GEFUNDEN DER NUR     *)
791     (* FORWARD DEKLARIERT IST ( .DECL=0 ),SO WIRD DECLERROR TRUE GESETZT UND   *)
792     (* DER NAME PROTOKOLLIERT.                                                 *)
793
794
795     BEGIN
796
797      IF H<> NIL THEN
798                  BEGIN
799                   IF H^.DECL=0 THEN               (* NUR FORWARD DEKLARIERT    *)
800                       BEGIN
801                        WRITELN(LISTFILE,' ');     (* PROTOKOLL. DES FEHLERS    *)
802                        WRITELN(LISTFILE,' MODEIDF.: ',H^.NAME);
803                        DECLERROR:=TRUE;
804                       END;
805                   SITREE(H^.LLINK,DECLERROR);     (*WEITERSUCHEN IM LINKEN  AST*)
806                   SITREE(H^.RLINK,DECLERROR);     (*WEITERSUCHEN IM RECHTEN AST*)
807                  END;
808
809     END;     (* OF PROCEDURE SITREE *)
810
811    BEGIN                                          (* OF PROCEDURE DECLTEST     *)
812
813     DECLERROR:=FALSE;
814     SITREE(MWURZELP,DECLERROR);                   (* DURCHSUCHEN D. MODEBAUMES *)
815                                                   (* BEGINN AN DER WURZEL      *)
816     IF DECLERROR=TRUE THEN FEHLER(18);
817
818    END;      (* OF PROCEDURE DECLTEST *)
819
820
821
822
```

```
823    PROCEDURE STANDARDMODES;
824    (* DIE PROZEDUR STANDARDMODES DEKLARIERT DIE MODES                          *)
825    (*         SMODE1=FUNC(S-EXPR)S-EXPR;                                      *)
826    (*         SMODE2=FUNC(S-EXPR,S-EXPR)S-EXPR;                               *)
827
828    BEGIN
829
830     NEWMIDFREC('SMODE1  ',MIDFRECP);              (* DEKLARATION VON SMODE1    *)
831     NEWMODE('S-EXPR  ');
832     MIDFRECP^.RESULTMOD:='S-EXPR  ';
833
834     NEWMIDFREC('SMODE2  ',MIDFRECP);              (* DEKLARATION VON SMODE2    *)
835     NEWMODE('S-EXPR  ');
836     NEWMODE('S-EXPR  ');
837     MIDFRECP^.RESULTMOD:='S-EXPR  ';
838
839    END;      (* OF PROCEDURE STANDARMODES  *)
840
841
842
843    PROCEDURE MODEDECL;
844
845     PROCEDURE MIDF(I:longint);
846     (* <MIDF>::=<IDENTIFIER>/SMODE1/SMODE2                                     *)
847     (* I STEUERT IM FEHLERFALL DIE FEHLERMELDUNG UND BEWIRKT DIE JEW. NOETIGEN *)
848     (* ZUSATZHANDLUNGEN IM MODEBAUM.                                           *)
849     (* I=2:MODEIDF. IN <MODEDECL>  /=7: RESULTMODEIDF./=8 MODEIDF IN <MODELIST>*)
850
851
852     BEGIN
853
854      IF (ISSYM<>'SMODE1  ')AND(ISSYM<>'SMODE2  ')
855          THEN IF NOT(IDENTIFIER) THEN FEHLER(I);
856      IF I=2 THEN NEWMIDFREC(ISSYM,MIDFRECP)       (* EINTARG IN DEN MODENBAUM  *)
857             ELSE IF I=7 THEN SEARCH(ISSYM)        (* TEST OB DEKLARIERT,FALLS  *)
858                                                   (* NICHT:FORWARD-D. IN SEARCH*)
859                         ELSE IF I=8 THEN
860                               BEGIN
861                                SEARCH(ISSYM);     (* WIE BEI I=7               *)
862                                NEWMODE(ISSYM)     (* EINTRAG IN DIE MODELISTE  *)
863                               END
864
865     END;  (* OF PROCEDURE MIDF *)
866
867
868
869     PROCEDURE STRUCTMODE;
870
871      PROCEDURE RESULTMODE;
872
873      BEGIN
874      (* <RESULTMODE>::=<MIDF.>/VOID/S-EXPR                                     *)
875
876       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
877       IF NOT((ISSYM='VOID    ')OR(ISSYM='S-EXPR  ')) THEN MIDF(7);
878       MIDFRECP^.RESULTMOD:=ISSYM;
879
880      END;   (*  OF PROCEDURE RESULTMODE  *)
881
882
883
884      PROCEDURE MODELIST;
885
886       PROCEDURE MODE;
887
888       BEGIN
889       (*   <MODE>::=<MIDF>/S-EXPR                                            *)
890
891        IF ISSYM<>'S-EXPR  ' THEN MIDF(8)           (* 8=^ IDF. IN MODELISTE    *)
892                             ELSE NEWMODE(ISSYM)
893
894       END;  (*  OF PROCEDURE MODE  *)
895
896      BEGIN
897      (*  <MODELIST>::=<EMPTY>/<MODE>[,<MODE>]                                 *)
```

```
898
899       IF ISSYM<>')       ' THEN                  (* MODELISTE NICHT LEER      *)
900           BEGIN
901            MODE;
902            NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
903            WHILE ISSYM=',       ' DO
904             BEGIN
905              NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
906              MODE;
907              NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
908             END
909           END
910
911      END;   (* OF PROCEDURE MODELIST *)
912
913
914     BEGIN                                        (* OF PROCEDURE STRUCTMODE   *)
915     (*  <STRUCT. MODE>::=FUNC(<MODELIST>)<RESULT MODE>                          *)
916
917      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
918      IF ISSYM<>'FUNC     ' THEN FEHLER(4);
919      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
920      IF ISSYM<>'(        ' THEN FEHLER(5);
921      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
922      MODELIST;
923      IF ISSYM<>')        ' THEN FEHLER(9);
924      RESULTMODE;
925
926     END; (*  OF PROCEDURE STRUCTMODE *)
927
928
929    BEGIN                                         (* OF PROCEDURE MODEDECL     *)
930    (*  <MODE DECL.>::=MODE <MIDF>=<STRUCT.MODE>                                *)
931    (*  DAS SYMBOL "MODE" WURDE BEREITS VOR DEM AUFRUF VON MODE ERKANNT.        *)
932
933     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
934     MIDF(2);                                     (* 2=^ AUFRUF AUS MODEDECL.  *)
935     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
936     IF ISSYM<>'=        ' THEN FEHLER(3);
937     STRUCTMODE;
938
939    END;  (* OF PROCEDURE MODEDECL *)
940
941
942   BEGIN                                          (* OF PROCEDURE MODEDECLPART *)
943   (*  <MODE DECL. PART>::=<EMPTY>/<MODE DECL.>;[<MODE DECL.>;]                  *)
944
945    STANDARDMODES;                               (*DEKL. VON SMODE1 UND SMODE2*)
946    NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
947    WHILE ISSYM='MODE     ' DO                   (*WEITERE MODEDEKLARATION    *)
948     BEGIN
949      MODEDECL;
950      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
951      IF ISSYM<>';        ' THEN FEHLER(6);
952      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
953     END;
954     DECLTEST;                                   (* TEST OB ALLE FORWARD-DEKL.*)
955                                                 (* MODEIDF. AUCH DEKLARIERT. *)
956   END;  (*  OF PROCEDURE MODEDECLPART  *)
957
958
959
960
961   (* ********************************************************************** *)
962   (* **********  F U N C T I O N   D E C L A R A T I O N   P A R T   ********* *)
963   (* **********  IM 1. LAUF NUR ANALYSE DER DEKLARATIONEN OHNE       ********* *)
964   (* **********  DIE AUSDRUECKE IN DEN RUEMPFEN .                    ********* *)
965   (* **********  ERZEUGUNG VON ZWISCHENCODE AUF DEM FILE ZWCODE.     ********* *)
966   (* **********  ABLAGE DER DEFINIEREND VORKOMMENDEN IDENTIFIKATOREN ********* *)
967   (* **********  IM IDENTIFIKATORBAUM .                              ********* *)
968   (* ********************************************************************** *)
969
970
971
972
```

```
 973  PROCEDURE FUNCDECLPART;
 974
 975  VAR
 976        MIDFRECP    :   POMIDFREC;              (* VERWEIS IN DEN MODEBAUM   *)
 977
 978
 979  PROCEDURE NEWID(ID:strin;T:EOZ;MODE:strin;NR:NUMB;MRP:POMIDFREC;ADR:longint);
 980
 981  (* DIE PROZEDUR VERSIEHT FUNKTIONSIDENTIFIKATOREN(TYP=1) BZW. FORMALE        *)
 982  (* PARAMETERIDENTIFIKATOREN (TYP=2) MIT EINER NUMMER(NR) UND TRAEGT SIE IN   *)
 983  (* EINEN IDENTIFIKATORBAUM IN LEXIKOGRAPH. ORDNUNG EIN.AUF DIE WURZEL DES    *)
 984  (* BAUMES ZEIGT STETS WURZELP.  PARAMETER VON NEWID:                         *)
 985  (* ID   : NAME DES IDENTIFIKATORS      T    : TYP (1 BZW. 2)                 *)
 986  (* MODE : MODE DES IDENTIFIKATORS      NR   : NUMMER(EINDEUTIGE UNTERSCHEID. *)
 987  (* MRP  : ZEIGER IN DEN MODEBAUM AUF DEN EINTRAG VON MODE                    *)
 988  (* ADR  : RELATIVADRESSE DES FORMALEN PARAMETERS VOM TYP=2 .                 *)
 989
 990
 991  VAR
 992       NAMENR  :  PACKAR;                       (* MIT NR VERSEHENER IDF.NAME*)
 993
 994    PROCEDURE PACK(IDFNAME : strin;NR : NUMB;VAR NAMENR : PACKAR);
 995    (* DIE PROZEDUR PACK VERSIEHT DIE IDENTIFIKATOREN MIT EINER NUMMER ZU IHRER*)
 996    (* EINDEUTIGEN UNTERSCHEIDUNG.DAS VERFAHREN IST IN KAPITEL ???? BESCHRIEBEN*)
 997
 998
 999    VAR
1000     I,J   :  longint;                          (* LAUFVARIABLE              *)
1001
1002    BEGIN
1003
1004     FOR I:=1 TO PACKLENG DO NAMENR[I]:=' ';    (* VORBESETZEN               *)
1005     I:=1;
1006     WHILE IDFNAME[I]<>' ' DO
1007      BEGIN
1008       NAMENR[I]:=IDFNAME[I];                   (* 1.TEIL: IDENTIFIKATORNAME *)
1009       I:=I+1
1010      END;
1011     NAMENR[I]:='*';                            (* 2.TEIL: EIN TRENNZEICHEN  *)
1012     J:=1;
1013     I:=I+1;
1014     WHILE NR[J]<>' ' DO
1015      BEGIN
1016       NAMENR[I]:=NR[J];                        (* 3.TEIL: DIE NUMMER        *)
1017       I:=I+1;
1018       J:=J+1;
1019      END
1020
1021
1022    END;   (* OF PROCEDURE PACK *)
1023
1024    PROCEDURE NEWLINK(NAMENR:PACKAR;VAR O:IDF);
1025    (* FALLS BEREITS MINDESTENS EIN EINTRAG IM IDF.BAUM VORHANDEN IST (WURZELP *)
1026    (* <>NIL) , SO ORDNET DIE PROZEDUR DEN IDENTIFIKATOR AN "RICHTIGER" STELLE *)
1027    (* IN DEN BAUM EIN.   PARAMETER VON NEWLINK :                              *)
1028    (* NAMENR : DER MIT NUMMER VERSEHENE IDFNAME   O : JEW. STELLE IM BAUM     *)
1029
1030    BEGIN
1031
1032     IF NAMENR<>O.NAMENR THEN
1033       IF NAMENR > O.NAMENR THEN
1034         IF O.RLINK=NIL THEN                    (* RECHTER NACHFOLGER FREI   *)
1035                        BEGIN
1036                         NEW(O.RLINK);          (* ANLEGEN EINES IDF-RECORDS *)
1037                         O.RLINK^.TYP:=T;
1038                         O.RLINK^.MODEIDF:=MODE;
1039                         O.RLINK^.STATNIV:=SCHACHT-1;
1040                         IF MRP<>NIL THEN       (* MRP=NIL:Z.B. MODE=S-EXPR  *)
1041                          BEGIN
1042                           O.RLINK^.MODEP:=MRP^.MODEP;
1043                           O.RLINK^.RESULTMOD:=MRP^.RESULTMOD;
1044                          END;
1045                         IF T=2 THEN O.RLINK^.RELADD:=ADR
1046                                ELSE O.RLINK^.STARTADR:=LABELNR;
1047                         O.RLINK^.NAMENR:=NAMENR;
```

```
1048                          O.RLINK^.RLINK:=NIL;
1049                          O.RLINK^.LLINK:=NIL;
1050                         END
1051                        ELSE                        (* WEITERSUCHEN RECHTER AST  *)
1052                         NEWLINK(NAMENR,O.RLINK^)
1053                      ELSE
1054                        IF O.LLINK=NIL THEN       (* LINKER NACHFOLGER FREI    *)
1055                          BEGIN
1056                           NEW(O.LLINK);
1057                           O.LLINK^.TYP:=T;
1058                           O.LLINK^.MODEIDF:=MODE;
1059                           O.LLINK^.STATNIV:=SCHACHT-1;
1060                           IF MRP<>NIL THEN
1061                            BEGIN
1062                             O.LLINK^.MODEP:=MRP^.MODEP;
1063                             O.LLINK^.RESULTMOD:=MRP^.RESULTMOD;
1064                            END;
1065                           IF T=2 THEN O.LLINK^.RELADD:=ADR
1066                                  ELSE O.LLINK^.STARTADR:=LABELNR;
1067                           O.LLINK^.NAMENR:=NAMENR;
1068                           O.LLINK^.LLINK:=NIL;
1069                           O.LLINK^.RLINK:=NIL;
1070                          END
1071                        ELSE                        (* WEITERSUCHEN LINKER AST   *)
1072                         NEWLINK(NAMENR,O.LLINK^)
1073                   ELSE FEHLER(16);               (* DOPPELTDEKLARATION        *)
1074
1075    END; (*OF PROCEDURE NEWLINK *)
1076
1077   BEGIN                                          (* OF PROCEDURE NEWID        *)
1078
1079    PACK(ID,NR,NAMENR);                           (* NAMENR:  ID-NAME*NUMMER   *)
1080    IF WURZELP=NIL THEN                           (* KEIN EINTRAG IM IDFBAUM   *)
1081                     BEGIN
1082                      NEW(WURZELP);
1083                      WURZELP^.TYP:=T;
1084                      WURZELP^.MODEIDF:=MODE;
1085                      WURZELP^.STATNIV:=SCHACHT-1;
1086                      IF MRP<>NIL THEN
1087                       BEGIN
1088                        WURZELP^.MODEP:=MRP^.MODEP;
1089                        WURZELP^.RESULTMOD:=MRP^.RESULTMOD;
1090                       END;
1091                      IF T=2 THEN WURZELP^.RELADD:=ADR
1092                             ELSE WURZELP^.STARTADR:=LABELNR;
1093                      WURZELP^.NAMENR:=NAMENR;
1094                      WURZELP^.LLINK:=NIL;
1095                      WURZELP^.RLINK:=NIL
1096                     END
1097                     ELSE NEWLINK(NAMENR,WURZELP^);(* SUCHE VON FREIEM PLATZ    *)
1098    IF T=1 THEN LABELNR:=LABELNR+1;                (*NAECHSTE FREIE STARTADRESSE*)
1099
1100   END;  (* OF PROCEDURE NEWID  *)
1101
1102
1103
1104
1105   PROCEDURE ENTER(VAR SCHACHT : longint;VAR NUMBER,HNUMBER : NUMB);
1106   (* DIE PROZEDUR AENDERT DIE NUMMER FUER DIE EINDEUTIGE IDENTIFIKATORUNTER-  *)
1107   (* SCHEIDUNG.SIEHE KAPITEL ???? ????? .AUFRUF ERFOLGT JEWEILS BEI ERREICHEN *)
1108   (* DES BEGINS EINES ERWEITERTEN PROZEDURRUMMPFES.AN DIESER STELLE ERHOEHT   *)
1109   (* SICH DIE SCHACHTELUNGSTIEFE UM EINS .                                    *)
1110
1111   VAR I : longint;                               (* LAUFVARIABLE              *)
1112
1113   BEGIN
1114
1115    IF SCHACHT<MAXNEST THEN SCHACHT:=SCHACHT+1     (* ERHOEHUNG SCHACHTEL.TIEFE*)
1116                       ELSE FEHLER(22);            (* ZU TIEFE SCAHCHTELUNG    *)
1117    FOR I := ANZSTLLN*(SCHACHT-1)+1 TO ANZSTLLN*SCHACHT
1118     DO NUMBER[I] := HNUMBER[I]
1119   (*OD*)
1120
1121
1122   END;          (*  OF PROCEDURE ENTER  *)
```

```
1123
1124
1125
1126    PROCEDURE LEAVE(VAR SCHACHT : longint;VAR NUMBER,HNUMBER : NUMB);
1127    (* DIE PROZEDUR AENDERT DIE NUMMER UND DIE HILFSNUMMER.SIE WIRD JEW. AM     *)
1128    (* PROZEDURRUMPFENDE AUFGERUFEN. FERNER ERNIEDRIGT SICH DIE SCHACHTELUNGS - *)
1129    (* TIEFE AN DIESER STELL UM EINS.                                          *)
1130
1131    VAR I : longint;                                          (* LAUFVARIABLE  *)
1132
1133    BEGIN
1134
1135     CHANGEIND(HNUMBER,SCHACHT,+1,RELATIV);    (* ERHOEHE HNUMBER[SCHACHT] UM 1 *)
1136     IF SCHACHT<(MAXNEST-1) THEN CHANGEIND(HNUMBER,SCHACHT+1,1,ABSOLUT) (*FI*);
1137     FOR I := ANZSTLLN*(SCHACHT-1) + 1  TO  ANZSTLLN * SCHACHT
1138      DO NUMBER[I] := ' '
1139    (*OD*);
1140     SCHACHT:=SCHACHT-1;
1141     IF SCHACHT<0 THEN FEHLER(23)
1142
1143
1144    END;  (* OF PROCEDURE LEAVE  *)
1145
1146
1147    PROCEDURE SEARCHM(MODEIDF:strin;VAR MIDFRECP : POMIDFREC);
1148    (* DIE PROZEDUR SUCHT IM MODEBAUM DEN MODEIDENTIFIKATOR(MODEIDF) UND LIEFERT*)
1149    (* IN MIDFRECP EINEM ZEIGER AUF DAS ZUGEHOERIGE RECORD AB,FALLS MODEIDF EIN-*)
1150    (* GETRAGEN IST.                                                           *)
1151
1152
1153    VAR
1154         FOUND    :  BOOLEAN;                     (* =TRUE : MODEIDF GEFUNDEN  *)
1155         H        :  POMIDFREC;                   (* JEW. STELLE IM MODEBAUM   *)
1156
1157    BEGIN
1158
1159     FOUND:=FALSE;
1160     H:=MWURZELP;                                  (* SUCHE BEGINNT A. D. WURZEL*)
1161     WHILE (H<>NIL) AND NOT(FOUND) DO
1162          BEGIN
1163           IF H^.NAME=MODEIDF THEN
1164                             BEGIN
1165                              FOUND:=TRUE;
1166                              MIDFRECP:=H
1167                             END
1168                            ELSE
1169                             IF H^.NAME>MODEIDF THEN H:=H^.LLINK
1170                                                ELSE H:=H^.RLINK;
1171          END;
1172     IF NOT(FOUND) THEN FEHLER(18);
1173
1174    END;  (* OF PROCEDURE SEARCHM  *)
1175
1176
1177    PROCEDURE STANDARDFUNCS;
1178    (* DIE PROZEDUR DEKLARIERT DIE LISP/N-STANDARDFUNKTIONEN ATOM,CAR,CDR,CONS  *)
1179    (* UND EQ.AUFRUF ERFOLGT AM ANFANG DES FUNKTIONSDEKALRATIONSTEIL.           *)
1180
1181
1182    BEGIN
1183                                             (*1-STELLIGE STANDARDFUNKTIONEN*)
1184    SEARCHM('SMODE1  ',MIDFRECP);            (* ALLE MIT MODE SMODE1        *)
1185    NEWID('ATOM    ',1,'SMODE1  ',NUMBER,MIDFRECP,0);
1186    NEWID('CAR     ',1,'SMODE1  ',NUMBER,MIDFRECP,0);
1187    NEWID('CDR     ',1,'SMODE1  ',NUMBER,MIDFRECP,0);
1188
1189                                             (*2-STELLIGE STANDARDFUNKTIONEN*)
1190    SEARCHM('SMODE2  ',MIDFRECP);              (*ALLE VOM MODE SMODE2       *)
1191    NEWID('CONS    ',1,'SMODE2  ',NUMBER,MIDFRECP,0);
1192    NEWID('EQ      ',1,'SMODE2  ',NUMBER,MIDFRECP,0);
1193    newid('PCONS_  ',1,'SMODE2  ',number,midfrecp,0); (* fuer permutierte Param.*)
1194
1195    labelnr:=labelnr+9;                      (* also 9 RUECKKEHRADR. VERBRAUCHT*)
1196
1197   END;   (* OF PROCEDURE STANDARDFUNCS *)
```

```
1198
1199
1200
1201
1202    PROCEDURE FUNCDECL;
1203
1204    VAR
1205           FUNCRESMODE : strin;                    (* RESULTMODE DES FUNKT.MODES*)
1206           MODEN       : strin;                    (* MODENAME DER JEW. FUNKT.  *)
1207
1208
1209
1210     PROCEDURE MIDF;
1211     (* DIE PROZEDUR MIDF STELLT FEST OB EIN MODENAME VORLIEGT,OB ER DEKJLARIERT*)
1212     (* IST.FALLS JA WIRD DER NAME IN MODEN GEHALTEN ;DER ZUGEHOERIGE RESULTMODE*)
1213     (* IN FUNCRESMODE.                                                        *)
1214
1215
1216     BEGIN
1217     (* <MIDF.>::=<IDENTIFIER>/SMODE1/SMODE2                                   *)
1218
1219      IF (ISSYM<>'SMODE1  ')AND(ISSYM<>'SMODE2  ') THEN
1220       IF NOT(IDENTIFIER) THEN FEHLER(2);
1221      SEARCHM(ISSYM,MIDFRECP);                    (* SUCHEN IM MODEBAUM        *)
1222      MODEN:=ISSYM;
1223      FUNCRESMODE:=MIDFRECP^.RESULTMOD;
1224
1225     END;   (* OF PROCEDURE MIDF IN FUNCDECL *)
1226
1227
1228     PROCEDURE FIDF;
1229     (* DIE PROZEDUR STELLT FEST OB IDENTIFIER VORLIEGT UND TRAEGT IHN GGF. IN  *)
1230     (* DENN IDENTIFIKATORBAUM ALS FUNKTIONSIDENTIFIKATOR EIN  .                *)
1231
1232     VAR
1233          RELADD     :  longint;                 (* REDUNDANT - FUER PARAM.POS*)
1234
1235     BEGIN
1236     (* <FIDF>::=<IDENTIFIER>                                                  *)
1237
1238      IF NOT(IDENTIFIER) THEN FEHLER(2)
1239                         ELSE NEWID(ISSYM,1,MODEN,NUMBER,MIDFRECP,RELADD);
1240
1241     END;  (* OF PROCEDURE  FIDF IN FUNCDECL  *)
1242
1243
1244     PROCEDURE FORMPARLIST;
1245
1246     VAR
1247           H          : POMR ;                    (* ZEIGER IN MODELISTE DES   *)
1248                                                  (* MODES D. JEW. FUNKTION    *)
1249           MIDFRECP1 : POMIDFREC;                 (* ZEIGER IN MODERECORD DES  *)
1250                                                  (* ZULETZT ERKANNTEN MODEIDF.*)
1251                                                  (* IN FORM. PARAMETERLISTE   *)
1252           RELADD     : longint;                  (* RELATIVADRESSE F.FORM.PAR.*)
1253
1254
1255      PROCEDURE MODE1(VAR MIDFRECP1 : POMIDFREC);
1256      (* DIE PROZEDUR MODE1 PRUEFT OB EIN MODEIDENTIFIKATOR VORLIEGT.D.H.=S-EXPR*)
1257      (* ODER SUCHE IM MODEBAUM . IN MIDFRECP1  WIRD IN DIESEM FALL DER VERWEIS *)
1258      (* AUF DAS ZUGEHOERIGE MODERECORD IM MODEBAUM GEHALTEN.DIESER VERWEIS WIRD*)
1259      (* SPAETER IN DAS PARAMETERRECORD EINGETRAGEN. AUSSERDEM WIRD UEBERPRUEFT *)
1260      (* OB DIESER MODENAME(ISSYM) IDENTISCH MIT DEM IN DER DEKLARATION DES MODE*)
1261      (* DER JEW. FUNKTION IST (=H^.MODEIDF). VGL. DEF. II.??????????????      *)
1262
1263      BEGIN
1264      (* <MODE>::=<MIDF.>/S-EXPR                                                *)
1265
1266       MIDFRECP1:=NIL;                            (* VORBESETZUNG              *)
1267       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1268       IF NOT(IDENTIFIER OR (ISSYM='S-EXPR  ')) THEN FEHLER(8);
1269       IF IDENTIFIER THEN SEARCHM(ISSYM,MIDFRECP1);(* SUCHEN NUR FALLS<>S-EXPR  *)
1270       MODEN:=ISSYM;                              (* NAME DES MODEIDF. HALTEN  *)
1271       IF H=NIL THEN FEHLER(19)                   (* KEINE WEITEREN MODES IN   *)
1272                                                  (* MODELISTE DER MODEDEKLAR. *)
```

```
1273               ELSE
1274                IF ISSYM<>H^.MODEIDF THEN         (* MODENAME<>NAME IN MODEDEK *)
1275                                      FEHLER(19)
1276                                     ELSE         (* H AUF NAECHSTEN EINTRAG I *)
1277                                      H:=H^.NEXTP;(* MODELISTE D. MODEDEKL.    *)
1278
1279      END;  (*  OF PROCEDURE MODE1 IN FORMPARLIST  *)
1280
1281
1282
1283      PROCEDURE FORMALIDENTIFIER;
1284      (* DIE PROZEDUR PRUEFT, OB EIN IDENTIFIKATOR VORLIEGT UND TRAEGT IHN GGF. *)
1285      (* ALS PARAMETERIDENTIFIKATOR IN DEN IDITIFIKATORBAUM EIN .               *)
1286
1287      BEGIN
1288      (* <FORMALIDF.> ::= <IDENTIFIER>                                         *)
1289
1290       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1291       IF NOT(IDENTIFIER) THEN FEHLER(2)
1292                          ELSE NEWID(ISSYM,2,MODEN,NUMBER,MIDFRECP1,RELADD);
1293
1294      END;    (* OF PROCEDURE FORMALIDENIFIER IN FORMPARLIST *)
1295
1296
1297
1298     BEGIN                                        (* OF PROCEDURE FORMPARLIST  *)
1299     (* <FORM.PAR.LIST>::=<EMPTY>/<MODE>:<FORMALIDF.>[,<MODE>:<FORMALIDF.>]     *)
1300
1301      RELADD:=0;                                  (* VORBESETZEN RELATIVADRESSE*)
1302      H:=MIDFRECP^.MODEP;                         (* ANFANG DER MODE-LISTE DER *)
1303                                                  (* DEKL. DES MODES DER FUNKT.*)
1304      WHILE ISNEXTSYM<>')       ' DO
1305              BEGIN
1306               RELADD:=RELADD+1;                  (* NAECHSTER PARAMETERPLATZ  *)
1307               MODE1(MIDFRECP1);
1308               NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1309               IF ISSYM<>':       ' THEN FEHLER(11);
1310               FORMALIDENTIFIER;
1311               IF ISNEXTSYM<>')       ' THEN       (* PARAMETERLISTE GEHT WEITER*)
1312                   BEGIN
1313                    NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1314                    IF ISSYM<>',        ' THEN FEHLER(12);
1315                    IF ISNEXTSYM=')       ' THEN FEHLER(2);
1316                   END;
1317              END;                                (* OF WHILE                  *)
1318      IF H<>NIL THEN FEHLER(19);                  (* MODELISTE IN DEKL. LAENGER*)
1319
1320     END;  (* OF PROCEDURE FORMPARLIST  *)
1321
1322
1323     PROCEDURE FUNCBODY;
1324
1325      PROCEDURE EXPRESSION;
1326      (* DIE PROZEDUR PRUEFT , OB AUSDRUCK LT. DEKLARATION LEER(VOID) SEIN MUSS.*)
1327      (* FALLS NICHT WIRD BIS ZUM FUNKTIONSENDE (]) GELESEN. DIE WEITERE ANALYSE*)
1328      (* DES AUSDRUCKS ERFOLGT ERST IM 2. LAUF: FUNCDECLARATIONPART2            *)
1329
1330      BEGIN
1331
1332       IF FUNCRESMODE='VOID    ' THEN FEHLER(39);
1333       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1334       WHILE ISSYM<>'}       ' DO                 (* LESEN UND ZWISCHENCODE ER-*)
1335          NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);  (* ZEUGEN BIS ZUM RUMPFENDE. *)
1336
1337      END;   (*  OF PROCEDURE EXPRESSION  *)
1338
1339
1340     BEGIN                                        (* OF PROCEDURE FUNCBODY     *)
1341     (* <FUNCBODY>::={<EMPTY>]/{<FUNCT.DECL.PART><EXPRESSION>]                  *)
1342
1343      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);     (* { IN DEN ZWCODE           *)
1344      IF ISSYM<>'{       ' THEN FEHLER(13);
1345      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1346      IF ISNEXTSYM=':       ' THEN FUNCDECLPART   (* INNERE FUNKTIONSDEKLARAT. *)
1347                              ELSE                (* FALLS <>":" :ERSTES SYMBOL*)
```

```
1348                              BEGIN               (* DES AUSDRUCKS IN ZWCODE   *)
1349                               IF SYMZ=0 THEN WRITE(ZWCODE,ISSYM)
1350                                         ELSE WRITE(ZWCODE,' ',ISSYM);
1351                               SYMZ:=SYMZ+1;       (* nach 8 Zeichen: neue Zeile*)
1352                               if symz>7 then begin writeln(zwcode); symz:=0 end;
1353                              END;
1354      IF ISSYM<>'}       ' THEN EXPRESSION
1355                           ELSE IF FUNCRESMODE<>'VOID    ' THEN FEHLER(49);
1356
1357      LEAVE(SCHACHT,NUMBER,HNUMBER);               (* ENDE DES ERW. FUNKT.RUMPF *)
1358
1359     END; (*  OF PROCEDURE FUNCBODY *)
1360
1361
1362
1363     PROCEDURE RESULTMODE;
1364     (* DIE PROZEDUR SUCHT DEN RESULTMODENAMEN,FALLS ER <> S-EXPR BZW. VOID IST,*)
1365     (* IM MODEBAUM.FERNER PRUEFT SIE, OB DIESER NAME IDENTISCH MIT DEMJENIGEN  *)
1366     (* (FUNCRESMODE) IN DER DEKLARATION DES MODES DER JEW. FUNKTION IST.       *)
1367
1368     VAR
1369           MIDFREC1    : POMIDFREC ;               (* REDUNDANTER PARAMETER     *)
1370
1371     BEGIN
1372     (* <RESULTMODE>::=<MIDF.>/VOID/S-EXPR                                      *)
1373
1374      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1375      IF NOT((ISSYM='VOID    ')OR(ISSYM='S-EXPR  '))THEN
1376       IF NOT(IDENTIFIER) THEN FEHLER(7)
1377                          ELSE
1378                           SEARCHM(ISSYM,MIDFREC1);(* MIDFREC1 REDUNDANT        *)
1379      IF ISSYM<>FUNCRESMODE THEN FEHLER(21);       (* NICHT IDENT. MIT DEKLARAT.*)
1380
1381     END;  (* OF  PROCEDURE RESULTMODE *)
1382
1383
1384
1385    BEGIN                                          (* OF PROCEDURE FUNCDECL     *)
1386    (*<FUNC.DECL>::=<MIDF>:<FIDF>(<FORM.PAR.LIST>)<RESULT MODE>;<FUNC.BODY>     *)
1387
1388     MIDF;
1389     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);      (* ":" IN FUNCDECLPART ERK.  *)
1390     IF SYMZ=0 THEN WRITE(ZWCODE,'**FUNC**')
1391               ELSE WRITE(ZWCODE,' **FUNC**');
1392     SYMZ:=SYMZ+1;
1393     if symz>7 then begin writeln(zwcode); symz:=0 end;   (* nach 8 Zeichen: CR *)
1394     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);       (* UEBERLIESST ":"           *)
1395     FIDF;
1396     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1397     IF ISSYM<>'(       ' THEN FEHLER(5);
1398     ENTER(SCHACHT,NUMBER,HNUMBER);                (* BEGINN ERW. FUNKTIONSRUMPF*)
1399     FORMPARLIST;
1400     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1401     IF ISSYM<>')       ' THEN FEHLER(09);
1402     RESULTMODE;
1403     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1404     IF ISSYM<>';       ' THEN FEHLER(6);
1405     FUNCBODY;
1406
1407    END;  (* OF PROCEDURE FUNCDECL  *)
1408
1409   BEGIN                                           (* OF PROCEDURE FUNCDECLPART *)
1410   (* <FUNC.DECL.PART>::=<EMPTY>/<FUNC.DECL.>;[<FUNC.DECL.>;]                   *)
1411
1412    IF SCHACHT=1 THEN STANDARDFUNCS;               (* BEIM ERSTEN AUFRUF:STAND..*)
1413    IF ISNEXTSYM<>':       ' THEN                  (* KEINE FUNKTIONDEKLARATION *)
1414                          BEGIN                    (* SOMIT **EMPTY** IN ZWCODE *)
1415                           IF SYMZ=0 THEN WRITE(ZWCODE,'**EMPTY*')
1416                                     ELSE WRITE(ZWCODE,' **EMPTY*');
1417                           SYMZ:=SYMZ+1;           (* nach 8 Zeichen: neue Zeile*)
1418                           if symz>7 then begin writeln(zwcode); symz:=0 end;
1419                          END
1420                         ELSE
1421                          WHILE ISNEXTSYM=':       ' DO
1422                           BEGIN
```

```
1423                          FUNCDECL;
1424                          NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1425                          IF ISSYM<>';       ' THEN FEHLER(6);
1426                          NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,FALSE);
1427                         END;
1428    IF SYMZ=0 THEN WRITE(ZWCODE,ISSYM)             (* DAS ERSTE SYMBOL VON EXPR.*)
1429              ELSE WRITE(ZWCODE,' ',ISSYM);        (* BZW. MAINPR. IN ZWCODE    *)
1430    SYMZ:=SYMZ+1;
1431    if symz>7 then begin writeln(zwcode); symz:=0 end     (* nach 8 Zeichen: CR *)
1432
1433  END;  (* OF PROCEDURE FUNCDECLPART *)
1434
1435
1436
1437  (* ************************************************************************ *)
1438  (* ********  2. LAUF : ERZEUGUNG DES PASCAL-ZIELCODES     ***************** *)
1439  (* ********                                               ***************** *)
1440  (* ************************************************************************ *)
1441
1442
1443
1444  (* ************************************************************************ *)
1445  (* ********************      M A I N P R O G R A M     ********************** *)
1446  (* ********************                                ********************** *)
1447  (* ********   VOLLSTAENDIGE ANALYSE UND ERZEUGUNG VON ZIELCODE:  *********** *)
1448  (* ********   KONSTANTEN AUF PASFIL1/REST AUF PASFIL2            *********** *)
1449  (* ************************************************************************ *)
1450
1451
1452
1453
1454  PROCEDURE MAINPROGRAM;
1455
1456
1457  VAR
1458        H          :   POMR;                    (* REDUNDANTER PAR.BEI IFST  *)
1459        IFPART     :   CHAR;                    (* TYP DES THEN/ELSE-TEILS   *)
1460        RESULTM    :   strin;                   (* FUER RESULTMODE BEI FUNCST*)
1461        VOIDBODY   :   BOOLEAN;                 (*TRUE : MAINPROGRAMMODE VOID*)
1462
1463
1464
1465   PROCEDURE PACK(IDFNAME : strin;NR : NUMB;VAR NAMENR : PACKAR);
1466   (* ANALOG ZU PROZEDUR PACK IN FUNCDECLPART.                                *)
1467   (* IM HAUPTPROGRAMM BEKOMMEN ALLE IDENTIFIKATOREN DIE NUMMER 1.            *)
1468
1469   VAR
1470    I, J  :  longint;                           (* LAUFVARIABBLE             *)
1471
1472   BEGIN
1473    FOR I:=1 TO PACKLENG DO NAMENR[I]:=' ';
1474    I:=1;
1475    WHILE IDFNAME[I]<>' ' DO
1476     BEGIN
1477      NAMENR[I]:=IDFNAME[I];
1478      I:=I+1
1479     END;
1480    NAMENR[I]:='*';                             (* TRENNZEICHEN              *)
1481    FOR J := I+1 TO I+ANZSTLLN-1                (* IM HAUPTPROGRAMM IST DER  *)
1482     DO NAMENR[J] :='0'                         (* INDEX 0..01               *)
1483   (*OD*);
1484    NAMENR[I+ANZSTLLN] := '1';
1485
1486   END;   (* OF PROCEDURE PACK *)
1487
1488
1489
1490   PROCEDURE SEARCH(IDFNAME:PACKAR;VAR IDFP : POIDF);
1491   (* DIE PROZEDUR SUCHT EINEN IDENTIFIKATOR(IDFNAME) IM IDENTIFIKATORBAUM UND *)
1492   (* LIEFERT GGF. IN IDFP EINEN VERWEIS AUF DAS ZUG. RECORD IM IDF.BAUM AB.   *)
1493
1494   VAR
1495       FOUND   :  BOOLEAN;                      (* =TRUE : IDENTIF. GEFUNDEN *)
1496       H       :  POIDF;                        (* JEW. STELLE IM IDF.BAUM   *)
1497
```

```
1498    BEGIN
1499
1500     FOUND:=FALSE;
1501     H:=WURZELP;                                    (* WURZEL DES IDENTIF.BAUMES *)
1502     WHILE (H<>NIL) AND (NOT FOUND) DO
1503         BEGIN
1504          IF H^.NAMENR=IDFNAME THEN
1505                           BEGIN
1506                            FOUND:=TRUE;
1507
1508                            IDFP:=H;
1509                           END
1510                          ELSE
1511                            IF H^.NAMENR>IDFNAME THEN H:=H^.LLINK
1512                                                 ELSE H:=H^.RLINK;
1513         END;
1514     IF NOT(FOUND) THEN FEHLER(18);
1515
1516    END;  (* OF PROCEDURE SEARCH  *)
1517
1518
1519    PROCEDURE SEARCHM(NS:strin;VAR MIDFRECP : POMIDFREC);
1520    (* DIE PROZEDUR SEARCHM SUCHT EINEN MODEIDENTIFIKATOR IM MODEIDENT.BAUM .   *)
1521    (* IN MIDFRECP WIRD GGF. DER VERWEIS AUF DAS ZUGEHOERIGE RECORD GEHALTEN .  *)
1522
1523     VAR
1524          FOUND   :  BOOLEAN;
1525          H       :  POMIDFREC;
1526
1527     BEGIN
1528      FOUND:=FALSE;
1529      H:=MWURZELP;
1530      IF (NS='S-EXPR  ') THEN FEHLER(43);
1531      IF (NS='VOID    ') THEN FEHLER(44);
1532      WHILE (H<>NIL) AND NOT(FOUND) DO
1533          BEGIN
1534           IF H^.NAME=NS THEN
1535                           BEGIN
1536                            FOUND:=TRUE;
1537                            MIDFRECP:=H
1538                           END
1539                          ELSE
1540                           IF H^.NAME>NS THEN H:=H^.LLINK
1541                                         ELSE H:=H^.RLINK;
1542          END;
1543      IF NOT(FOUND) THEN FEHLER(42);
1544
1545     END;  (* OF PROCEDURE SEARCHM  *)
1546
1547
1548   (* *********  LOKALE FORWARDDEKLARATIONEN                          ********** *)
1549
1550
1551    PROCEDURE FUNCST(VAR RESULTMOD : strin);FORWARD;
1552    PROCEDURE IFST(H:POMR;TEST:BOOLEAN;VAR IFPART:CHAR);FORWARD;
1553
1554   (* ************************************************************************** *)
1555
1556
1557
1558    PROCEDURE SEXPREXPR;
1559
1560    LABEL 1;                                       (* MARKE AM ENDE DER PROZ.   *)
1561
1562    VAR    RESULTMODE  :   strin;                  (* RESULTMODE BEI FUNCST.    *)
1563
1564
1565
1566     PROCEDURE SEXPR;
1567     (* DIE PROZEDUR LIESST S-AUSDRUECKE UND BEREITET SIE FUER DIE LZS-PROZEDUR *)
1568     (* INHEAP AUF.S-AUSDRUECKE SIND PROGRAMMKONSTANTEN!!                       *)
1569
1570
1571      FUNCTION ATOMSEXPR : BOOLEAN;
1572      (* DIE FUNKTION PRUEFT OB EIN ATOMARER S-AUSDRUCK VORLIEGT .              *)
```

```
1573
1574      VAR
1575          I : longint;                                (* LAUFVARIABLE             *)
1576
1577      BEGIN
1578
1579       I:=2;
1580       ATOMSEXPR:=TRUE;                              (* VORBESETZUNG             *)
1581       IF NOT(ISSYM[1] IN LETTER) THEN
1582                                   ATOMSEXPR:=FALSE
1583                                  ELSE
1584                                   REPEAT
1585                                    IF NOT(ISSYM[I] IN LETDIG)
1586                                          THEN ATOMSEXPR:=FALSE;
1587                                    I:=I+1
1588                                   UNTIL I=(SYMLENG+1);
1589
1590      END; (*  OF FUNCTION ATOMSEXPR  *)
1591
1592
1593
1594      PROCEDURE LIST;
1595
1596      (* DIE PROZEDUR LIESST NICHT ATOMARE S-AUSDRUECKE UND BEREITET SIE FUER   *)
1597      (* DIE LZS PROZEDUR INHEAP AUF.                                           *)
1598
1599      BEGIN
1600      (* <LIST>::=<ATOM> / <DOTTED PAIR>/<LIST> <LIST>/(<LIST>)                 *)
1601
1602       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1603       WRITE(PASFIL1,ISSYM);
1604       IF ISSYM='.       ' THEN FEHLER(51);
1605       WHILE ISSYM<>')       ' DO
1606            BEGIN
1607                WHILE ISSYM='(       ' DO
1608                      BEGIN
1609                       LIST;
1610                       NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1611                       WRITE(PASFIL1,ISSYM);
1612                      END ;
1613                IF ISSYM<>')       ' THEN
1614                      BEGIN
1615                       IF ISSYM<>'.       ' THEN
1616                          BEGIN
1617                           IF NOT(ATOMSEXPR) THEN FEHLER(41);
1618                           NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1619                           WRITE(PASFIL1,ISSYM);
1620                          END
1621                                            ELSE           (* ISSYM='.       '  *)
1622                          BEGIN
1623                           NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1624                           WRITE(PASFIL1,ISSYM);
1625                           IF (ISSYM=')       ')OR(ISSYM='.       ')
1626                             THEN FEHLER(51);
1627                           IF ISSYM='(       ' THEN LIST
1628                                                ELSE IF NOT(ATOMSEXPR)
1629                             THEN FEHLER(41);
1630                           NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1631                           WRITE(PASFIL1,ISSYM);
1632                           IF ISSYM<>')       ' THEN FEHLER(51);
1633                          END;
1634                      END;
1635            END;
1636
1637      END;  (* OF PROCEDURE LIST *)
1638
1639
1640     BEGIN                                          (* OF PROCEDURE SEXPR       *)
1641     (* <S-EXPR>::= (<ATOM> / (<LIST>)                                         *)
1642
1643      KONSTZ:=KONSTZ+1;
1644      IF ISSYM='"       'THEN
1645                         BEGIN
1646                          NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1647                          IF NOT(ATOMSEXPR) THEN FEHLER(41);
```

```
1648                        WRITELN(PASFIL1,'INHEAP(''"',ISSYM,''',',KONSTZ,');');
1649                       END
1650                      ELSE                        (* ISSYM='(       '         *)
1651                       BEGIN
1652                        WRITE(PASFIL1,'INHEAP(''(');
1653                        LIST;
1654                        WRITELN(PASFIL1,''',',KONSTZ,');');
1655                       END;
1656     WRITELN(PASFIL2,'AC:=B[',KONSTZ,'];');
1657
1658    END;  (* OF PROCEDURE SEXPR *)
1659
1660
1661
1662    PROCEDURE CONS;
1663    (*    CONS( <S-EXPR.EXPR.>,<S-EXPR.EXPR.>     / ISSYM='CONS    '           *)
1664
1665    BEGIN
1666
1667     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1668     IF ISSYM<>'(       ' THEN FEHLER(05);
1669     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1670     SEXPREXPR;
1671     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1672     IF ISSYM<>',       ' THEN FEHLER(12);
1673     WRITELN(PASFIL2,'PUSH;');
1674     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1675     SEXPREXPR;
1676     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1677     IF ISSYM<>')       ' THEN FEHLER(09);
1678     WRITELN(PASFIL2,'CONS;');
1679
1680    END;  (* OF PROCEDURE CONS *)
1681
1682
1683    PROCEDURE CARCDR;
1684    (* CAR(<S-EXPR.EXPR.>) BZW. CAR(<S-EXPR.EXPR.>) / ISSYM='CAR ' BZW. ='CDR '*)
1685
1686    VAR
1687          SYM      :  strin;                  (* ="CAR  " BZW. ="CDR  "       *)
1688
1689    BEGIN
1690
1691     SYM:=ISSYM;                                (* MERKEN OB CAR ODER CDR    *)
1692     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1693     IF ISSYM<>'(       ' THEN FEHLER(05);
1694     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1695     SEXPREXPR;
1696     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1697     IF ISSYM<>')       ' THEN FEHLER(09);
1698     if sym='CAR     ' then writeln(pasfil2,'CAR;'); (* IM CODE  CAR  BZW. CDR *)
1699     if sym='CDR     ' then writeln(pasfil2,'CDR;')
1700    END;  (* OF PROCEDURE CARCDR *)
1701
1702
1703
1704
1705   BEGIN
1706   (*<S-EXPR.EXPR.>::=<S-EXPR>/<FUNC.ST.>/CONS(<S-EXPR.EXPR.>,<S-EXPR.EXPR.>)/ *)
1707   (*                CDR(<S-EXPR.EXPR.>)/CAR(<S-EXPR.EXPR.>)/<FUNC.ST.>/      *)
1708   (*                IF<BOOL.EXPR.>THEN<S-EXPR.EXPR.>ELSE<S-EXPR.EXPR.>FI     *)
1709   (*                                                                          *)
1710   (* FUER CAR,CDR,CONS  SEPARATE LOKALE PROZEDUREN.                           *)
1711
1712    IF (ISSYM='T       ')OR(ISSYM='F       ') THEN (*SPEZIELLE S-AUSDRUECKE T/F*)
1713                       BEGIN
1714                        IF ISSYM='T       ' THEN
1715                                             WRITELN(PASFIL2,'AC:=B[1];')
1716                                            ELSE        (* ISSYM="F      "      *)
1717                                             WRITELN(PASFIL2,'AC:=B[2];');
1718                        GOTO 1
1719                       END;
1720    IF (ISSYM='"       ')OR(ISSYM='(       ') THEN
1721                       BEGIN
1722                        SEXPR;
```

```
1723                       GOTO 1;
1724                      END;
1725    IF ISSYM='CONS    ' THEN
1726                         BEGIN
1727                          CONS;
1728                          GOTO 1
1729                         END;
1730    IF (ISSYM='CAR     ') OR (ISSYM='CDR     ')
1731                      THEN
1732                       BEGIN
1733                        CARCDR;
1734                        GOTO 1;
1735                       END;
1736    IF ISSYM='IF      ' THEN
1737                         BEGIN
1738                          IFST(H,FALSE,IFPART);
1739                          IF (IFPART<>'S') THEN FEHLER(17);
1740                          GOTO 1;
1741                         END;
1742    FUNCST(RESULTMODE);
1743    IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(37);
1744  1:
1745
1746    END;  (* OF PROCEDURE SEXPREXPR *)
1747
1748    PROCEDURE BOOLEXPR;
1749    (* BOOLEXPR IST STANDARDMAESSIG VOM MODE S-EXPR.                          *)
1750
1751    LABEL   1       ;                              (* MARKE AM PROZEDURENDE     *)
1752    VAR
1753            RESULTMODE    :  strin;                (* RESULTMODE BEI FUNCST.    *)
1754
1755
1756     PROCEDURE ATOM;
1757     (*  ATOM(<S-EXPR.EXPR.>)            /   ISYM='ATOM  '                    *)
1758
1759     BEGIN
1760
1761      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1762      IF ISSYM<>'(       ' THEN FEHLER(05);
1763      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1764      SEXPREXPR;
1765      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1766      IF ISSYM<>')       ' THEN FEHLER(09);
1767      WRITELN(PASFIL2,'ATOM;');
1768
1769     END;  (* OF PROCEDURE ATOM   *)
1770
1771
1772
1773     PROCEDURE EQ;
1774     (*   EQ( <S-EXPR.EXPR.>,<S-EXPR.EXPR.>)    /   ISSYM='EQ  '               *)
1775
1776     BEGIN
1777
1778      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1779      IF ISSYM<>'(       ' THEN FEHLER(05);
1780      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1781      SEXPREXPR;
1782      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1783      IF ISSYM<>',       ' THEN FEHLER(12);
1784      WRITELN(PASFIL2,'PUSH;');
1785      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1786      SEXPREXPR;
1787      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
1788      IF ISSYM<>')       ' THEN FEHLER(09);
1789      WRITELN(PASFIL2,'EQ;');
1790
1791     END;  (* OF PROCEDURE EQ   *)
1792
1793
1794    BEGIN
1795    (*<BOOL.EXPR.>::=T/F/ATOM(<S-EXPR.EXPR.>)/EQ(<S-EXPR.EXPR.>,<S-EXPR.EXPR.>)/*)
1796    (*               IF<BOOL.EXPR.>THEN<BOOL.EXPR.>ELSE<BOOL.EXPR.>FI/<FUNC.ST> *)
1797    (*                                                                        *)
```

```
1798    (*  FUER ATOM UND EQ SEPARATE LOKALE PROZEDUREN .                          *)
1799
1800     IF (ISSYM='T       ')OR(ISSYM='F       ')    (* ANALOG ZU T/F IN SEXPREXPR *)
1801                          THEN
1802                           BEGIN
1803                            SEXPREXPR;
1804                            GOTO 1
1805                           END;
1806     IF ISSYM='ATOM    ' THEN
1807                           BEGIN
1808                            ATOM;
1809                            GOTO 1
1810                           END;
1811     IF ISSYM='EQ      ' THEN
1812                           BEGIN
1813                            EQ;
1814                            GOTO 1
1815                           END;
1816     IF ISSYM='IF      ' THEN
1817                           BEGIN
1818                            IFST(H,FALSE,IFPART);
1819                            IF IFPART<>'S' THEN FEHLER(17);
1820                            GOTO 1;
1821                           END;
1822     IF ISNEXTSYM='(       ' THEN
1823                              BEGIN
1824                               FUNCST(RESULTMODE);
1825                               IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(40);
1826                                                  (* HIER ZUR UEBERS.ZEIT NICHT*)
1827                                                  (* PRUEFBAR,OB ZUR LAUFZEIT  *)
1828                                                  (* S-AUSDRUCK= "T" BZW. "F"  *)
1829                               GOTO 1;
1830                              END;
1831     FEHLER(28);
1832    1:
1833
1834    END;  (* OF PROCEDURE BOOLEXPR *)
1835
1836
1837    PROCEDURE FUNCST;(*    (VAR RESULTMOD : strin);                            *)
1838    (* DIE PROZEDUR LIEFERT IN DER VARIABLEN RESULTMOD DEN RESULTMODE DES       *)
1839    (* ZUGEHOERIGEN FUNKTIONSIDENTIFIKATOR AB.                                 *)
1840    (* <FUNC. ST.> ::= <IDENTIFIER>(<ACT.PAR.LIST>) [(<ACT.PAR.LIST.>)]        *)
1841
1842    VAR
1843               IDFP   :  POIDF ;                  (*VERWEIS A. FUNKT.IDF.RECORD*)
1844               MODEP  :  POMR;                    (* ZEIGER IN MODEL.D.FUNCIDF.*)
1845               NAMENR :  PACKAR;                  (* MIT NR.VERSEHENER IDF.NAME*)
1846               SADR   :  longint;                 (* STARTADRESSSE DER FUNKT.  *)
1847               STN    :  longint;                 (* STATISCHES NIVEAU D. FUNKT*)
1848
1849
1850     PROCEDURE ACTPARLIST;
1851
1852     VAR
1853          H   :  POMR;                           (* ZEIGER IN JEW. MODELISTE  *)
1854                                                 (* DER DEKL. DES FUNKT.MODES *)
1855      PROCEDURE ACTPAR(VAR H : POMR    )  ;
1856      (* DIE VARIABLE H ZEIGT IN DIE JEW. MODELISTE DER DEKLARATION DES MODES   *)
1857      (* DES FUNKTIONSIDENTIFIKATORS.DIESE MODES MUESSEN MIT DENJENIGEN DER     *)
1858      (* EINZELNEN AKTUELLLEN PARAMETER IDENTISCH SEIN.S. KAP. II ????????????? *)
1859
1860      LABEL 1;
1861
1862      VAR
1863              CARCONSCDR  :    BOOLEAN;     (* =TRUE,FALLS ISSYM=CAR/CONS/CDR   *)
1864              HLABELNR    :    longint;     (* HAELT LABELNR FUER DAS LABEL AN  *)
1865                                            (* DEM PARAMETER BESCHR. FORTSETZT  *)
1866              IDFP        :    POIDF;       (* VERWEIS AUF FUNKTIONSIDF.RECORD  *)
1867              RESULTMODE  :    strin;       (* RESULTMODE BEI FUNC.ST.          *)
1868
1869
1870
1871       PROCEDURE FUNCEXPR(VAR H : POMR);
1872       (* H WIE BEI ACTPAR . LOKAL ZU ACTPAR, DA FUNKTIONSAUSDRUECKE IM HAUPT-  *)
```

```
1873       (* PROGRAMM NUR ALS AKT. PARAMETER ZULAESSIG SIND.                      *)
1874
1875       VAR      IDFP       :  POIDF ;       (* VERWEIS AUF FUNKTIONSIDF.RECORD  *)
1876                NAMENR     :  PACKAR;       (* MIT NR. VERSEHENER IDF.NAME      *)
1877
1878       BEGIN
1879       (* <FUNC. EXPR.>::=ATOM/CAR/CDR/CONS/EQ/<IDENTIFIER>                     *)
1880                                                (* ISSYM STANDARDFUNKT.IDF.? *)
1881        IF (ISSYM<>'ATOM    ')AND(ISSYM<>'CAR     ') THEN
1882           IF (ISSYM<>'CDR     ')AND(ISSYM<>'CONS    ') THEN
1883              IF (ISSYM<>'EQ      ') THEN
1884               IF NOT(IDENTIFIER) THEN FEHLER(2);
1885        PACK(ISSYM,NUMBER,NAMENR);              (* NAMENR = ISSYM*NUMBER     *)
1886        SEARCH(NAMENR,IDFP);                    (* IDENT. SUCHEN IM IDF.BAUM *)
1887        IF H^.MODEIDF<>IDFP^.MODEIDF THEN FEHLER(30);
1888        H:=H^.NEXTP;                            (* NACHFOLGER IN MODELISTE   *)
1889        WRITELN(PASFIL2,'LZSPARB(1,',IDFP^.STATNIV,',',IDFP^.STARTADR,');');
1890
1891       END;  (*  OF PROCEDURE FUNCEXPR *)
1892
1893      BEGIN                                     (* OF PROCEDURE ACTPAR       *)
1894      (* <ACT. PAR.>::=<BOOL. EXPR.>/<S-EXPR.EXPR.>/FUNC.EXPR.>/FUNC.ST.>/IN    *)
1895
1896      IF ISSYM='IN      ' THEN                  (* "IN" NUR IM HAUPTPR.      *)
1897           BEGIN
1898            IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
1899            H:=H^.NEXTP;
1900            KONSTZ:=KONSTZ+1;
1901            WRITELN(PASFIL2,'INP(',KONSTZ,');');
1902            WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',',KONSTZ,');');
1903            GOTO 1;
1904           END;
1905      IF ISSYM='IF      ' THEN                  (* "DICKER PARAMETER"        *)
1906           BEGIN
1907            HLABELNR:=LABELNR;                  (* BEGINN N.PARAMETER IM CODE*)
1908            WRITELN(PASFIL2,'LZSPARB(2,',SCHACHT-1,',',LABELNR+1,');');
1909            WRITELN(PASFIL2,'GOTO ',LABELNR,';');
1910            WRITELN(PASFIL2,LABELNR+1,':');
1911            LABELNR:=LABELNR+2;                 (* HLABELNR + RUECKKEHRADR   *)
1912            IFST(H,TRUE,IFPART);
1913            WRITELN(PASFIL2,'LZSFEND; goto 0;');
1914            WRITELN(PASFIL2,HLABELNR,':');
1915            H:=H^.NEXTP;                        (* NAECHSTER EINTRAG MODELIST*)
1916            GOTO 1;
1917           END;
1918      IF (ISSYM='T       ')OR(ISSYM='F       ')OR
1919         (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
1920           BEGIN
1921            IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
1922                IF ISNEXTSYM<>'(       ' THEN   (* PROC.EXPR. LIEGT VOR      *)
1923                                       BEGIN
1924                                       FUNCEXPR(H);
1925                                       GOTO 1;
1926                                      END;
1927            IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
1928            H:=H^.NEXTP;
1929            IF ISSYM='T       ' THEN            (* SPEZIELLER S-AUSDRUCK     *)
1930              WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',1);')
1931               ELSE
1932                IF ISSYM='F       ' THEN
1933                 WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',2);')
1934                  ELSE                          (* ISSYM=EQ/ATOM             *)
1935                   BEGIN                        (* DICKER PARRAMETER         *)
1936                    HLABELNR:=LABELNR;
1937                    WRITELN(PASFIL2,'LZSPARB(2,',SCHACHT-1,',',LABELNR+1,');');
1938                    WRITELN(PASFIL2,'GOTO ',LABELNR,';');
1939                    WRITELN(PASFIL2,LABELNR+1,':');
1940                    LABELNR:=LABELNR+2;
1941                    BOOLEXPR;
1942                    WRITELN(PASFIL2,'LZSFEND; goto 0;');
1943                    WRITELN(PASFIL2,HLABELNR,':');
1944                   END;
1945            GOTO 1;
1946           END;
1947      IF (ISSYM='CONS    ')OR(ISSYM='CAR     ')OR(ISSYM='CDR     ')
```

```
1948           THEN CARCONSCDR:=TRUE
1949           ELSE CARCONSCDR:=FALSE;
1950      IF (ISSYM='(       ')OR CARCONSCDR OR(ISSYM='"       ') THEN
1951           BEGIN
1952            IF CARCONSCDR THEN
1953                IF ISNEXTSYM<>'(       ' THEN      (* FUNCEXPRESSION            *)
1954                                     BEGIN
1955                                      FUNCEXPR(H);
1956                                      GOTO 1;
1957                                     END;
1958                IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
1959                H:=H^.NEXTP;
1960                IF NOT(CARCONSCDR) THEN            (* ISSYM = " / (             *)
1961                  BEGIN
1962                   SEXPREXPR;
1963                   WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',',KONSTZ,');');
1964                  END
1965                 ELSE                              (* ISSYM = CONS/CDR/CAR      *)
1966                  BEGIN                            (* DICKER PARAMETER          *)
1967                   HLABELNR:=LABELNR;
1968                   WRITELN(PASFIL2,'LZSPARB(2,',SCHACHT-1,',',LABELNR+1,');');
1969                   WRITELN(PASFIL2,'GOTO ',LABELNR,';');
1970                   WRITELN(PASFIL2,LABELNR+1,':');
1971                   LABELNR:=LABELNR+2;
1972                   SEXPREXPR;
1973                   WRITELN(PASFIL2,'LZSFEND; goto 0;');
1974                   WRITELN(PASFIL2,HLABELNR,':');
1975                  END;
1976                GOTO 1;
1977           END;
1978      IF ISNEXTSYM='(       ' THEN                (* DICKER PAR.:PROZEDURANW.  *)
1979           BEGIN
1980            HLABELNR:=LABELNR;
1981            WRITELN(PASFIL2,'LZSPARB(2,',SCHACHT-1,',',LABELNR+1,');');
1982            WRITELN(PASFIL2,'GOTO ',LABELNR,';');
1983            WRITELN(PASFIL2,LABELNR+1,':');
1984            LABELNR:=LABELNR+2;
1985            FUNCST(RESULTMODE);
1986            WRITELN(PASFIL2,'LZSFEND; goto 0;');
1987            WRITELN(PASFIL2,HLABELNR,':');
1988            IF RESULTMODE<>H^.MODEIDF THEN FEHLER(30);
1989            H:=H^.NEXTP;
1990           END
1991          ELSE                                    (* PROZEDURAUSDRUCK          *)
1992           FUNCEXPR(H);                           (* CODEERZEUGUNG IN FUNCEXPR *)
1993   1:
1994
1995      END; (* OF PROCEDURE ACTPAR *)
1996
1997
1998     BEGIN                                       (* OF PROCEDURE ACTPARLIST   *)
1999     (* <ACTPARLIST>::=<EMPTY>/<ACT.PAR.>[,<ACT.PAR.>]                         *)
2000
2001      H:=MODEP;                                  (* ZEIGER IN JEW. MODELISTE  *)
2002                                                 (* AUS DER MODEDEKLARATION   *)
2003      NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);    (* ISSYM=') ' : ACTPARL.EMPTY*)
2004      WHILE ISSYM<>')       ' DO
2005                          BEGIN
2006                           IF H=NIL THEN FEHLER(38);(* KEINE WEIT.PAR. DEKLARIERT*)
2007                           ACTPAR(H);
2008                           NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2009                           IF ISSYM<>')       ' THEN
2010                                IF ISSYM<>',       ' THEN
2011                                     FEHLER(12)
2012                                       ELSE NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE)
2013                          END;
2014      IF H<>NIL THEN FEHLER(38);                 (* WEITERE PARAM. DEKLARIERT *)
2015
2016     END;  (* OF PROCEDURE ACTPARLIST *)
2017
2018    BEGIN                                        (* OF PROCEDURE FUNCST       *)
2019    (* <FUNCST.>::=<IDENTIFIER>( <ACT.PAR.LIST> ) [ <ACT.PAR. LIST> ]          *)
2020
2021     IF NOT(IDENTIFIER) THEN FEHLER(2);
2022     PACK(ISSYM,NUMBER,NAMENR);                  (* NAMENR = ISSYM*NUMBER     *)
```

```
2023    SEARCH(NAMENR,IDFP);                          (* SUCHT FUNKTIONSIDF.       *)
2024    RESULTMOD:=IDFP^.RESULTMOD;                   (*RESULTMODE D. FUNKTIONSIDF.*)
2025    MODEP:=IDFP^.MODEP;                           (* ZEIGER AUF MODELISTE  " " *)
2026    NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2027    IF ISSYM <>'(       ' THEN FEHLER(05);
2028    ACTPARLIST;
2029    IF (MODEP = NIL)                                (*  LEERE PARAMETERLISTE   *)
2030          AND
2031       (RESULTMOD <> 'S-EXPR  ')
2032          AND
2033       (RESULTMOD <> 'VOID    ')                    (*  FUNKTIONALES ERGEBNIS  *)
2034          AND
2035       (ISNEXTSYM <> '(       ')                    (* KEINE PENDING PARAMETER *)
2036     THEN             (* evtl. anfallende PP's werden nicht zu akt. Parametern *)
2037       WRITELN(PASFIL2,'LZSTRENN;')
2038  (*FI*);
2039    WHILE ISNEXTSYM='(       ' DO                   (* PENDING PARAMETERS      *)
2040                          BEGIN
2041                           WRITELN(PASFIL2,'LZSTRENN;');
2042                           SEARCHM(RESULTMOD,MIDFRECP);
2043                           RESULTMOD:=MIDFRECP^.RESULTMOD;
2044                           NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2045                           MODEP:=MIDFRECP^.MODEP;
2046                           ACTPARLIST
2047                          END;
2048                                                  (* LABELNR: RUECKKEHRADR.    *)
2049    SADR:=IDFP^.STARTADR;                         (* ABKUERUNG FUER CODEERZ.   *)
2050    STN:=IDFP^.STATNIV;                           (*    "       "     "        *)
2051    WRITELN(PASFIL2,'LZSARK(0,',SADR,',',LABELNR,',',STN,','R'',0); goto 0;');
2052    WRITELN(PASFIL2,LABELNR,':');
2053    LABELNR:=LABELNR+1;
2054
2055   END;  (* OF PROCEDURE FUNCST *)
2056
2057
2058
2059
2060   PROCEDURE IFST; (* H : POMR,TEST :  BOOLEAN;VAR IFPART : CHAR               *)
2061   (* WIRD VON DEN PROZEDUREN  SEXPREXPR/BOOLEXPR/FUNCST/IFST/ACTPAR           *)
2062   (* AUFGERUFEN,FALLS DORT AUF "IF" GETROFFEN WIRD.                           *)
2063   (* WIRD IFPART AUS ACTPAR AUFGERUFEN,SO IST TEST=TRUE UND H ZEIGT IN DIE    *)
2064   (* MODELISTE ANALOG ZU H IN ACTPAR. IN IFPART WIRD DER TYP DES IF-TEILS     *)
2065   (* GEHALTEN.(ER MUSS GLEICH DEM TYP DES ELSE -TEILS SEIN UND DER AUFRUFENDEN*)
2066   (* PROZEDUR ENTSPRECHEN.                                                    *)
2067
2068
2069
2070   LABEL 1,                                       (* MARKE AM ENDE DES IFPARTS *)
2071         2;                                       (* MARKE AM ENDE DER PROZEDUR*)
2072
2073   VAR
2074        CARCONSCDR       :   BOOLEAN ;            (* =TRUE,ISSYM=CAR/CONS/CDR  *)
2075        HIFPART          :   CHAR;                (* HAELT BEI REK. IFPART     *)
2076        MODE             :   strin;               (* MODE EINES PROZEDURAUSDR. *)
2077        RESULTMODE       :   strin;               (* RESULTMODE BEI FUNCST.    *)
2078
2079
2080
2081    PROCEDURE FUNCEXPR(VAR MODE:strin);
2082    (* LOKAL ZU IFST -  AUFRUF NUR MOEGLICH WENN IFST AUS ACTPAR AUFGERUFEN    *)
2083    (* WURDE, DA IM HAUPTPROGRAMM PROZEDURAUSDRUECKE NUR ALS AKT. PARAMETER    *)
2084    (* ZULAESSIG SIND.IN MODE WIRD DER MODENAME DES IDENTIFIKATORS(ISSYM) GE-  *)
2085    (* HALTEN.ER MUSS MIT MODE VON ACT.PARAMTER UEBEREINSTIMMEN.               *)
2086
2087    VAR
2088         IDFP  :   POIDF;                         (* VERW. FUNKTIONSIDF.RECORD *)
2089         NAMENR:   PACKAR;                        (* MIT NR. VERSEHENER IDFNAME*)
2090         SADR  :   longint;                       (* ABKUERZUNG F.CODEERZEUGUNG*)
2091
2092    BEGIN
2093
2094     IF (ISSYM<>'ATOM    ')AND(ISSYM<>'CAR     ')AND(ISSYM<>'CDR     ') THEN
2095        IF (ISSYM<>'CONS    ')AND (ISSYM<>'EQ      ') THEN
2096            IF NOT(IDENTIFIER) THEN FEHLER(2);
2097     PACK(ISSYM,NUMBER,NAMENR);
```

```
2098      SEARCH(NAMENR,IDFP);
2099      MODE:=IDFP^.MODEIDF;
2100      SADR:=IDFP^.STARTADR;
2101                              (* Rechter Ast; Im Hauptprogramm max.stat.Niv. = 0 *)
2102      WRITELN(PASFIL2,
2103             'LZSARK(0,',SADR,',',LABELNR,',',IDFP^.STATNIV,','R'',0); goto 0;');
2104      WRITELN(PASFIL2,LABELNR,':');
2105      LABELNR:=LABELNR+1;
2106
2107     END;  (*  OF PROCEDURE FUNCEXPR IN IFST  *)
2108
2109
2110
2111    BEGIN                                          (* OF PROCEDURE IFST         *)
2112
2113     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2114     BOOLEXPR;
2115     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2116     IF ISSYM<>'THEN    ' THEN FEHLER(31);
2117     WRITELN(PASFIL2,'IF BOOLTEST THEN BEGIN ');
2118     NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2119     IF ISSYM='IF      ' THEN
2120           BEGIN
2121            IFST(H,TEST,IFPART);
2122            GOTO 1
2123           END;
2124     IF (ISSYM='CONS    ')OR(ISSYM='CAR     ')OR(ISSYM='CDR     ')
2125           THEN CARCONSCDR:=TRUE
2126           ELSE CARCONSCDR:=FALSE;
2127     IF CARCONSCDR OR(ISSYM='(       ')OR(ISSYM='"       ') THEN
2128           BEGIN
2129            IF CARCONSCDR THEN
2130             IF ISNEXTSYM<>'(       ' THEN         (* FUNKTIONSAUSDRUCK         *)
2131                 BEGIN
2132                  IF NOT(TEST) THEN FEHLER(52);    (* FUNCEXPR. NUR AUS ACTPAR! *)
2133                  FUNCEXPR(MODE);
2134                  IF H^.MODEIDF<>MODE THEN FEHLER(30);
2135                  IFPART:='E';
2136                  GOTO 1;
2137                 END;
2138            IF TEST THEN IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
2139            SEXPREXPR;
2140            IFPART:='S';
2141            GOTO 1;
2142           END;
2143     IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ')
2144      OR(ISSYM='T       ')OR(ISSYM='F       ')THEN
2145           BEGIN
2146            IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
2147              IF ISNEXTSYM<>'(       ' THEN        (* FUNKTIONSAUSDRUCK         *)
2148                 BEGIN
2149                  IF NOT(TEST) THEN FEHLER(52);
2150                  FUNCEXPR(MODE);
2151                  IF H^.MODEIDF<>MODE THEN FEHLER(30);
2152                  IFPART:='E';
2153                  GOTO 1;
2154                 END;
2155            IF TEST THEN IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
2156            BOOLEXPR;
2157            IFPART:='S';
2158            GOTO 1;
2159           END;
2160     IF ISNEXTSYM='(       ' THEN                   (* FUNKTIONSANWEISUNG        *)
2161           BEGIN
2162            FUNCST(RESULTMODE);
2163            IF TEST THEN
2164              IF H^.MODEIDF<>RESULTMODE THEN FEHLER(30);
2165            IF RESULTMODE='S-EXPR  '
2166              THEN IFPART:='S'
2167                ELSE IF RESULTMODE='VOID    '
2168                       THEN IFPART:='V'
2169                         ELSE FEHLER(54);
2170            GOTO 1;
2171           END
2172          ELSE                                     (* FUNKTIONSAUSDRUCK         *)
```

```
2173          BEGIN
2174           IF NOT(TEST) THEN FEHLER(52);
2175           FUNCEXPR(MODE);
2176           IF TEST THEN IF H^.MODEIDF<>MODE THEN FEHLER(30);
2177           IFPART:='E';
2178           GOTO 1;
2179          END;
2180    FEHLER(34);
2181   1:NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2182    IF ISSYM<>'ELSE    ' THEN FEHLER(32);
2183    WRITELN(PASFIL2,'END ELSE BEGIN ');
2184    NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2185    IF ISSYM='IF      ' THEN
2186          BEGIN
2187           HIFPART:=IFPART;
2188           IFST(H,TEST,IFPART);
2189           IF IFPART<>HIFPART THEN FEHLER(33);
2190           GOTO 2;
2191          END;
2192    IF (ISSYM='CONS    ')OR(ISSYM='CAR     ')OR(ISSYM='CDR     ')
2193          THEN CARCONSCDR:=TRUE
2194          ELSE CARCONSCDR:=FALSE;
2195    IF CARCONSCDR OR(ISSYM='(       ')OR(ISSYM='"       ') THEN
2196          BEGIN
2197           IF CARCONSCDR THEN
2198            IF ISNEXTSYM<>'(       ' THEN         (* FUNKTIONSAUSDRUCK         *)
2199                BEGIN
2200                 IF IFPART<>'E' THEN FEHLER(33);
2201                 FUNCEXPR(MODE);
2202                 IF H^.MODEIDF<>MODE THEN FEHLER(30);
2203                 GOTO 2;
2204                END;
2205           IF IFPART<>'S' THEN FEHLER(33)
2206                ELSE
2207                 BEGIN
2208                  IF TEST THEN IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
2209                  SEXPREXPR;
2210                  GOTO 2;
2211                 END;
2212          END;
2213    IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ')
2214     OR(ISSYM='T       ')OR(ISSYM='F       ')THEN
2215          BEGIN
2216           IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
2217            IF ISNEXTSYM<>'(       ' THEN         (* FUNKTIONSAUSDRUCK         *)
2218                 BEGIN
2219                  IF IFPART<>'E' THEN FEHLER(33);
2220                  FUNCEXPR(MODE);
2221                  IF H^.MODEIDF<>MODE THEN FEHLER(30);
2222                  GOTO 2;
2223                 END;
2224           IF IFPART<>'S'
2225             THEN FEHLER(33)
2226                ELSE
2227                 BEGIN
2228                  IF TEST THEN IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
2229                  BOOLEXPR;
2230                  GOTO 2;
2231                 END;
2232          END;
2233    IF ISNEXTSYM='(       ' THEN                  (* FUNKTIONSANWEISUNG        *)
2234          BEGIN
2235           IF (IFPART<>'V')AND(IFPART<>'S') THEN FEHLER(33)
2236                ELSE
2237                 BEGIN
2238                  FUNCST(RESULTMODE);
2239                  IF TEST THEN IF H^.MODEIDF<>RESULTMODE
2240                                 THEN FEHLER(30)
2241                                   ELSE
2242                                    IF RESULTMODE<>'S-EXPR  ' THEN
2243                                      IF RESULTMODE<>'VOID    ' THEN FEHLER(54);
2244                               GOTO 2;
2245                 END
2246          END
2247         ELSE                                     (* FUNKTIONSAUSDRUCK         *)
```

```
2248          BEGIN
2249           IF IFPART<>'E' THEN FEHLER(33)
2250                          ELSE
2251                           BEGIN
2252                            FUNCEXPR(MODE);
2253                            IF TEST THEN IF H^.MODEIDF<>MODE THEN FEHLER(30);
2254                            GOTO 2;
2255                           END;
2256          END;
2257    FEHLER(35);
2258   2:NEXTSYM(ISSYM,ISNEXTSYM,NEXTCHAR,TRUE);
2259    IF ISSYM<>'FI      ' THEN FEHLER(36);
2260    WRITELN(PASFIL2,'END; ');
2261
2262   END;     (*  OF PROCEDURE IFST  *)
2263
2264
2265   BEGIN                                          (* OF PROCEDURE MAINPROGRAM  *)
2266   (* <MAINPROGRAM>::=<EMPTY>/S-EXPR.EXPR.>/<BOOL.EXPR.>/<FUNC.ST.>             *)
2267
2268    VOIDBODY:=FALSE;
2269    if symz>7 then begin writeln(zwcode); writeln(zwcode,'**MAIN**') end
2270              else
2271    if symz=0 then writeln(zwcode,'**MAIN**')
2272              else WRITELN(ZWCODE,' **MAIN**');    (* ENDE VON ZWCODE F. 2. LAUF*)
2273    symz:=0;                                       (* Anfang neuer Zeile        *)
2274   IF ISSYM='END     ' THEN                       (* LEERES PROGRAMM           *)
2275        BEGIN
2276         REWRITE(ZPROG);
2277         WRITELN(ZPROG,'PROGRAM LEER;');
2278         WRITELN(ZPROG,'BEGIN END.');
2279         WRITELN('  *****                                                ******  ');
2280         WRITELN('  *****      LEERES PROGRAMM WIRD ERZEUGT              ******  ');
2281         WRITELN('  *****                                                ******  ');
2282         ende                                                  (* Programmende *)
2283        END
2284       ELSE
2285        BEGIN
2286         IF (ISSYM='IN      ') THEN
2287                BEGIN
2288                 KONSTZ:=KONSTZ+1;
2289                 WRITELN(PASFIL2,'INP(,',KONSTZ,');');
2290                END
2291               ELSE
2292                IF (ISSYM='CONS    ')OR(ISSYM='(       ')
2293                 OR(ISSYM='CDR     ')OR(ISSYM='CAR     ')
2294                 OR(ISSYM='"       ') THEN SEXPREXPR
2295                         ELSE
2296                            IF (ISSYM='F       ')OR(ISSYM='T       ')
2297                             OR(ISSYM='ATOM    ')OR(ISSYM='EQ      ')
2298                                THEN BOOLEXPR
2299                                ELSE
2300                                 IF ISSYM='IF      ' THEN
2301                                     BEGIN
2302                                      IFST(H,FALSE,IFPART);
2303                                      IF IFPART='V'
2304                                        THEN VOIDBODY:=TRUE
2305                                          ELSE
2306                                           IF (IFPART<>'S') THEN FEHLER(53);
2307                                     END
2308                                    ELSE
2309                                     BEGIN
2310                                      FUNCST(RESULTM);
2311                                      IF RESULTM='VOID    '
2312                                        THEN VOIDBODY:=TRUE
2313                                         ELSE
2314                                          IF (RESULTM<>'S-EXPR  ')
2315                                            THEN FEHLER(53);
2316                                     END;
2317         IF NOT(VOIDBODY) THEN WRITELN(PASFIL2,'OUT; ');
2318        END;
2319
2320   END; (* OF MAINPROGRAM *)
2321
2322
```

```
2323
2324  (* ********************************************************************** *)
2325  (* *********   F U N C T I O N   D E C L A R A T I O N   P A R T  2   ******** *)
2326  (* *********                                                          ******** *)
2327  (* *********   KONSTANTEN AUF PASFIL1 / REST DES CODES AUF PASFIL2  ******** *)
2328  (* *********   QUELLCODE IST DER VOM 1.LAUF ERZEUGTE ZWISCHENCODE   ******** *)
2329  (* *********   AUF ZWCODE.DIE ANGEGEBENEN PRODUKTIONEN IN DIESEM    ******** *)
2330  (* *********   TEIL DES UEBERSAETZERS BEZIEHEN SICH AUF DIE SYNTAX  ******** *)
2331  (* *********   DES ERZEUGTEN ZWISCHENCODES UND SIND DAHER I.A. NICHT******** *)
2332  (* *********   IMMER IDENTISCH MIT DEN LISP/N-PRODUKTIONEN.         ******** *)
2333  (* ********************************************************************** *)
2334
2335
2336
2337  procedure gmarkfdecp2; (* Klammernde Prozedur von funcdeclpart2 und GPMARK   *)
2338
2339   var key : char;     (* Hilfsvariable fuer Tastatureingaben                 *)
2340       perm: boolean;  (* Flag: cons/eq gegebenenfalls permutieren ?          *)
2341
2342
2343   procedure funcdeclpart2; forward;
2344
2345
2346   procedure nextgmcodesym(var issym,isnextsym: strin; var gmniv,dpniv: longint);
2347
2348   (* Die Prozedur liesst jeweils ein Symbol auf GMZWCODE vorraus und legt es  *)
2349   (* in der Variablen ISNEXTSYM ab. Wird eine Markierung gelesen, so wird sie *)
2350   (* in GMNIV uebergeben und das naechste Symbol gelesen. Die Prozedur ist    *)
2351   (* ansonsten identisch mit NEXTCODESYM.                                    *)
2352
2353   VAR
2354     I            :  longint;                    (* LAUFINDEX                *)
2355     NEXTCHAR     :  CHAR;                       (* JEW. GELESENES ZEICHEN   *)
2356
2357  BEGIN
2358
2359    ISSYM:=ISNEXTSYM; ISNEXTSYM:=EMPTYSTRING;
2360    repeat
2361      if eoln(gmzwcode) then readln(gmzwcode)         (* Lesen vom Zeilenende  *)
2362                        else READ(gmzwcode,NEXTCHAR); (* LESEN D. TRENNSYMBOLS *)
2363      FOR I:=1 TO SYMLENG DO
2364                           BEGIN
2365                            READ(gmzwcode,NEXTCHAR);
2366                            ISNEXTSYM[I]:=NEXTCHAR
2367                           END;
2368      if (isnextsym[1]='(') and (isnextsym[2]='*')    (* Markierung lesen      *)
2369        then if isnextsym[6]='*'
2370               then dpniv:=ord(isnextsym[5])-48       (* Max Niv. Act. Para.   *)
2371               else gmniv:=ord(isnextsym[3])-48       (* Max Niv. im rlk       *)
2372    until not((isnextsym[1]='(') and (isnextsym[2]='*'));
2373    WRITE(LISTFILE,ISnextSYM);                    (* PROTOKOLLIERUNG DES SYMB. *)
2374    IF OUTCOUNT<9 THEN                            (* 10 SYMBOLE JE LISTINGZEILE*)
2375                   OUTCOUNT:=OUTCOUNT+1
2376                  ELSE
2377                   BEGIN
2378                    OUTCOUNT:=0;
2379                    WRITELN(LISTFILE,' ')
2380                   END
2381
2382   END;   (* of procedure nextgmcodesym *)
2383
2384
2385  procedure gminit2;
2386
2387  (* Die Prozedur sorgt fuer Vorbesetzungen vor der Uebersetzung des          *)
2388  (* GMARK-Zwischencodes                                                     *)
2389
2390  VAR                I    :   longint;            (* LAUFVARIABLE             *)
2391
2392  BEGIN
2393   RESET(gmzwcode);                               (* Enth. GMARK-Zwischencode  *)
2394   REWRITE(LISTFILE);
2395   WRITELN(LISTFILE,'UEBERSETZUNG DES ZWISCHENCODES ');
2396   WRITELN(LISTFILE,'FEHLERPROTOKOLL: '); writeln(listfile);
2397   ISNEXTSYM:=EMPTYSTRING; ISSYM:=EMPTYSTRING;
```

```
2398    FOR I := 1 TO INDLENGTH DO NUMBER[I] := ' ' (*OD*);
2399    FOR I := 1 TO MAXNEST DO CHANGEIND(HNUMBER,I,1,ABSOLUT) (*OD*);
2400    CHANGEIND(NUMBER,1,1,ABSOLUT); gmniv:=0;    (* Init. Markierung mit 0       *)
2401    nextgmcodesym(ISSYM,ISNEXTSYM,gmniv,dpniv); (* 1. gmzwcodesym. in ISNEXTSYM *)
2402    SCHACHT:=1; outcount:=0
2403
2404   END;   (* of procedure gminit2 *)
2405
2406
2407   (* Die Prozeduren ENTER, LEAVE, PACK und SEARCH sind vorgezogen, um fuer     *)
2408   (* die Prozedure GMARK erreichbar zu sein.                                   *)
2409
2410
2411    PROCEDURE ENTER(VAR SCHACHT : longint;VAR NUMBER,HNUMBER : NUMB);
2412    (* ANALOG ZU ENTER IN FUNCDECLPART                                          *)
2413
2414    VAR I :longint;                                        (* LAUFVARIABLE    *)
2415
2416    BEGIN
2417
2418     IF SCHACHT<MAXNEST THEN SCHACHT:=SCHACHT+1
2419                        ELSE FEHLER(22);
2420     FOR I := ANZSTLLN*(SCHACHT-1) + 1 TO ANZSTLLN * SCHACHT
2421      DO NUMBER[I] := HNUMBER[I]
2422    (*OD*)
2423
2424    END;  (* OF PROCEDURE ENTER *)
2425
2426
2427    PROCEDURE LEAVE(VAR SCHACHT : longint;VAR NUMBER,HNUMBER : NUMB);
2428    (* ANALOG ZU LEAVE INB FUNCDECLPART                                         *)
2429
2430    VAR I : longint;                                          (* LAUFVARIABLE *)
2431
2432    BEGIN
2433
2434     CHANGEIND(HNUMBER,SCHACHT,+1,RELATIV);
2435     IF SCHACHT<(MAXNEST-1) THEN CHANGEIND(HNUMBER,SCHACHT+1,1,ABSOLUT) (*FI*);
2436     FOR I := ANZSTLLN*(SCHACHT-1) + 1  TO  ANZSTLLN * SCHACHT
2437      DO NUMBER[I] := ' '
2438    (*OD*);
2439     SCHACHT:=SCHACHT-1;
2440     IF SCHACHT<0 THEN FEHLER(23);
2441
2442    END;  (*  OF PROCEDURE LEAVE  *)
2443
2444
2445
2446    PROCEDURE PACK(IDFNAME : strin;NR : NUMB;VAR NAMENR : PACKAR);
2447     (* ANALOG ZU FUNCDECLPART : IDF. WIRD MIT NUMMER VERSEHEN.                 *)
2448
2449     VAR
2450          I,J   :  longint;                        (* LAUFVARIABLE            *)
2451
2452     BEGIN
2453
2454      FOR I:=1 TO PACKLENG DO NAMENR[I]:=' ';
2455      I:=1;
2456      WHILE IDFNAME[I]<>' ' DO
2457       BEGIN
2458        NAMENR[I]:=IDFNAME[I];
2459        I:=I+1
2460       END;
2461      NAMENR[I]:='*';
2462      I:=I+1;
2463      J:=1;
2464      WHILE NR[J]<>' ' DO
2465       BEGIN
2466        NAMENR[I]:=NR[J];
2467        I:=I+1;
2468        J:=J+1;
2469       END
2470
2471     END;   (* OF PROCEDURE PACK *)
2472
```

```
2473
2474    PROCEDURE SEARCH(NAMENR:PACKAR;VAR STATNIV:SCHACHTINT;VAR TYP:EOZ;
2475     VAR ADR:longint;VAR MODEP:POMR;VAR MODE,RESULTMODE:strin);
2476     (* DIE PROZEDUR SUCHT DEN MIT NUMMER VERSEHENEN IDENTIFIKATOR(NAMENR) MIT  *)
2477     (* HILFE DER PROZEDUR SEARCH1 IM IDF.-BAUM.DABEI WIRD - SOLANGE DER IDF.   *)
2478     (* NOCH NICHT GEFUNDEN IST-  DIE NUMMER SUKZESSIVE UM DIE LETZTE ZIFFER    *)
2479     (* VERKUERZT. SIEHE KAP. ????????????????????????????????                 *)
2480     (* WIRD DER IDENTIFIKATOR MIT (REST-)NUMMER SCHLIESSLICH GEFUNDEN,SO WERDEN*)
2481     (* DAS STATISCHE NIVEAU(STATNIV),DER TYP(TYP),DIE STARTADRESSE BZW.RELATIV-*)
2482     (* ADRESSE (ADR) ,GGF. DER ZEIGER AUF DIE ZUGEH. MODELISTE(MODEP),DER MODE *)
2483     (* (MODE) UND GGF. DER ZUG. RESULTMODE(RESULTMODE) DES IDENTIFIKATORS      *)
2484     (* ABGELIEFERT.                                                           *)
2485
2486     VAR
2487         FOUND : BOOLEAN;                          (*  =TRUE: IDF. GEFUNDEN     *)
2488         I     : longint;                          (* LAUFVARIABLE              *)
2489         IDFP  : POIDF;                            (* ZEIGER A. GES. IDF.RECORD *)
2490         J,K   : longint;                          (* J:LAENGE NAME/K:L. NUMMER *)
2491
2492
2493      PROCEDURE SEARCH1(NS:PACKAR;VAR IDFP : POIDF;VAR FOUND : BOOLEAN);
2494      (* DIE PROZEDUR SUCHT IM IDENTIFIKATORBAUM DEN IDF. MIT NUMMER NAMENR.    *)
2495      (* WIRD ER GEFUNDEN, SO IST FOUND=TRUE.                                   *)
2496
2497      VAR
2498            H   :  POIDF;                         (* JEW. STELLE IM IDF.-BAUM  *)
2499
2500      BEGIN
2501
2502       FOUND:=FALSE;
2503       H:=WURZELP;
2504       WHILE (H<>NIL) AND (NOT FOUND) DO
2505           BEGIN
2506            IF H^.NAMENR=NS THEN
2507                             BEGIN
2508                              FOUND:=TRUE;
2509                              IDFP:=H;
2510                             END
2511                            ELSE
2512                              IF H^.NAMENR>NS THEN H:=H^.LLINK
2513                                              ELSE H:=H^.RLINK;
2514           END;
2515
2516      END;     (* OF PROCEDURE SEARCH1  *)
2517
2518     BEGIN
2519      I:=1;J:=0;K:=0;
2520      WHILE NAMENR[I] <>'*' DO                     (* LESEN DES IDENTIF.NAMEN   *)
2521                              BEGIN
2522                               J:=J+1;
2523                               I:=I+1
2524                              END;                 (* J : LAENGE DES NAMENSTEILS*)
2525      I:=I+1;                                      (*  UEBERLESEN TRENNSYM. '*' *)
2526      WHILE NAMENR[I]<>' ' DO                      (* LESEN DES NUMMERNTEILS    *)
2527                            BEGIN
2528                             I:=I+1;
2529                             K:=K+1
2530                            END;                   (* K : LAENGE DES NUMMERNTEIL*)
2531      FOUND:=FALSE;
2532      WHILE (K>0)AND(NOT FOUND) DO                 (* SUCHEN DES IDF. MIT NUMM. *)
2533                BEGIN
2534                 SEARCH1(NAMENR,IDFP,FOUND);
2535                 IF NOT(FOUND)                     (* ABSCHNEIDEN LETZTE ZIFFER *)
2536                     THEN NAMENR[J+K+1]:=' ';
2537                 K:=K-1
2538                END;
2539      IF NOT(FOUND) THEN FEHLER(18);
2540      STATNIV:=IDFP^.STATNIV;                      (* PARAM. ZUWEISUNGNEN       *)
2541      TYP:=IDFP^.TYP;
2542      IF TYP=2 THEN ADR:=IDFP^.RELADD ELSE ADR:=IDFP^.STARTADR;
2543      MODE:=IDFP^.MODEIDF;
2544      IF (MODE<>'S-EXPR  ')AND(MODE<>'VOID    ') THEN
2545                 BEGIN
2546                  RESULTMODE:=IDFP^.RESULTMOD;
2547                  MODEP:=IDFP^.MODEP
```

```
2548                END;
2549
2550     END;  (* OF PROCEDURE SEARCH  *)
2551
2552
2553
2554    PROCEDURE FUNCDECL2;
2555
2556      VAR
2557        FUNCRESMODE   :  strin;                    (* RESULTMODE DER FUNKTION   *)
2558        MODE          :  strin;                    (* MODE DES FUNKT.IDENTIF.   *)
2559        MODEP         :  POMR;                     (* ZEIGER A.MODELISTE ZU MODE*)
2560        NAMENR        :  PACKAR;                   (* MIT NR VERSEHENER IDF.NAME*)
2561        STARTADR      :  longint;                  (* STARTADRESSE DER FUNKTION *)
2562        STATNIV       :  SCHACHTINT;               (* STATISCHES NIVEAU DER  "  *)
2563        TYP           :  EOZ;                      (* TYP BEI SEARCH -HIER RED. *)
2564
2565
2566
2567     PROCEDURE EXPRESSION;                         (* IN FUNCDECL2              *)
2568
2569     LABEL    1,                                   (* MARKE AM ENDE DER PROZEDUR*)
2570              2;                                   (* MARKE FUER FUNC.EXPR.-BEH.*)
2571
2572     type   polast = ^last;                        (* Die Var. LKlschacht wird  *)
2573            last   = record                        (* kellerartig verwaltet: bei*)
2574                       pred      : polast;        (* Eintritt in Rumpf oder    *)
2575                       lklschacht: longint        (* akt. Parameterliste wird  *)
2576                     end;                         (* alter Wert von LKlschacht *)
2577                                                  (* gekellert u.LKlschacht neu*)
2578                                                  (* mit 0 initialisiert. Beim *)
2579                                                  (* Austritt wird alter Wert  *)
2580                                                  (* wiederhergestellt.        *)
2581     VAR
2582            ADR        :   longint;               (* START- BZW. RELATIVADRESSE*)
2583            CARCONSCDR :   BOOLEAN;               (* =TRUE:ISSYM=CAR/CONS/CDR  *)
2584            H          :   POMR;                  (* IFST-PARAM.-HIER REDUNDANT*)
2585            IFCOUNT    :   longint;               (* >0 =^ IN EINEM IF-PART    *)
2586            IFPART     :   CHAR;                  (* IFST-PARAM.-HIER REDUNDANT*)
2587            LSCHACHT   :   longint;               (*SCHACHTEL.V."LINKEN" AESTEN*)
2588            MODE       :   strin;                 (* MODE EINES IDENTIFIKATORS *)
2589            MODEP      :   POMR;                  (* ZEIGER AUF MODELISTE      *)
2590            NAMENR     :   PACKAR;                (* MIT NR. VERSEHENER IDENTIF*)
2591            RESULTMODE :   strin;                 (* RESULTMODE ZU EINEM MODE  *)
2592            STATNIV    :   SCHACHTINT;            (* STATISCHES NIV. EINES IDF.*)
2593            TYP        :   EOZ;                   (* TYP EINES IDENTIFIKATORS  *)
2594            la,laold   :   polast;                (* Zeiger auf akt. LKlschacht*)
2595
2596
2597      PROCEDURE SEARCHM(NS:strin;VAR MIDFRECP : POMIDFREC);
2598      (* ANALOG ZU SEARCHM IN MAINPROGRAM : SUCHT EINEN MODEIDF. IM MODEBAUM    *)
2599
2600      VAR
2601           FOUND   :  BOOLEAN;                    (* =TRUE,FALLS IDF. GEFUNDEN *)
2602           H       :  POMIDFREC;                  (* JEW. STELLE IM MODEBAUM   *)
2603
2604      BEGIN
2605
2606       FOUND:=FALSE;
2607       H:=MWURZELP;
2608       IF (NS='S-EXPR  ') THEN FEHLER(43);
2609       IF (NS='VOID    ') THEN FEHLER(44);
2610       WHILE (H<>NIL) AND (NOT FOUND) DO
2611           BEGIN
2612            IF H^.NAME=NS THEN
2613                           BEGIN
2614                            FOUND:=TRUE;
2615                            MIDFRECP:=H
2616                           END
2617                          ELSE
2618                           IF H^.NAME>NS THEN H:=H^.LLINK
2619                                         ELSE H:=H^.RLINK;
2620           END;
2621       IF NOT(FOUND) THEN FEHLER(42);
2622
```

```
2623      END;  (* OF PROCEDURE SEARCHM  *)
2624
2625
2626
2627      PROCEDURE GDVSTATUS(LSChaCHT: longint; VAR GDVST: CHAR);
2628      (* DIE PROZEDUR STELLT ANHAND DES PARAMETERS LSCHACHT FEST, OB DIE ANALYSE*)
2629      (* SICH Z. ZT. IN EINEM LINKEN BZW. RECHTEM AST BEFINDET.DER PARAMETER    *)
2630      (* GDVST BEKOMMT ENTSPRECHEND DEN WERT 'L' BZW. 'R' ZUGEWIESEN.           *)
2631      (* Bei der Berabeitung rechter Aeste wird nicht markiert und die Aufrufe  *)
2632      (* von lzsark erfolgen mit dem default-Wert gmniv:=0.                     *)
2633
2634      BEGIN
2635
2636       IF LSCHACHT>0 THEN GDVST:='L'
2637                     ELSE begin GDVST:='R'; gmniv:=0 end;
2638
2639      END;      (* OF PROCEDURE GDVSTATUS *)
2640
2641
2642
2643      PROCEDURE FUNCEXPRCONST(TYP : EOZ;ADR,SNV : longint);
2644      (* BEI AUFTRETEN EINES IDENTIFIKATORS WIRD CODE ERZEUGT:                  *)
2645      (* LIEGT EIN GEW. FUNKTIONSIDENTIFIKATOR VOR(TYP=1): CODE F. FUNKT.AUFRUF *)
2646      (* LIEGT EIN FORM. IDENTIFIKATOR VOR(TYP=2) : CODE DER ZUR LAUFZEIT IN   -*)
2647      (* ABHAENGIKEIT VOM TYP DES AKTUELLEN PARAMETERS (FUNKTIONSSIDF. BZW.     *)
2648      (* KONSTANTE )  EINEN FUNKTIONSAUFRUF BZW. KONSTANTENZUGRIFF AUSFUEHRT.   *)
2649      (* DER PARAMTER ADR ENTHAELT DIE START- BZW. RELATIVADRESSE; DER PARAMETER*)
2650      (* SNV DAS STATISCHE NIVEAU DES FORM. PARAMETERS.                         *)
2651
2652      VAR      GDVST   :      CHAR;                (*='R'/'L':RECHTER/LINKER AST*)
2653
2654      BEGIN
2655
2656       GDVSTATUS(LSCHACHT,GDVST);
2657       IF TYP=1 THEN                               (* GEWOEHNLICHER FUNKT.IDF.  *)
2658         BEGIN
2659          WRITELN(PASFIL2,
2660        'LZSARK(0,',ADR,',',LABELNR,',',SNV,',''',GDVST,''',',gmniv,'); goto 0;');
2661          WRITELN(PASFIL2,LABELNR,':');
2662          IF GDVST='L' THEN WRITELN(PASFIL2,'LZSLEFTEND(',labelnr,');');
2663          LABELNR:=LABELNR+1;
2664         END
2665        ELSE             (* TYP=2 : FORMALER IDF. *)
2666         BEGIN
2667          WRITELN(PASFIL2,'LZSFORMIDF(',SNV,',',ADR,',TYP,ADR);');
2668          WRITELN(PASFIL2,'IF TYP=0 THEN AC:=B[ADR] ');
2669          WRITELN(PASFIL2,'ELSE BEGIN ');
2670          if mode<>'S-EXPR  ' then              (* Call By Need: Typ-1 Ergebnis *)
2671            writeln(pasfil2,'if need then T1update:=true;');
2672          WRITELN(PASFIL2,
2673      'LZSARK(typ,',ADR,',',LABELNR,',',SNV,',''',GDVST,''',',gmniv,'); goto 0;');
2674          WRITELN(PASFIL2,LABELNR,':');
2675          IF GDVST='L' THEN WRITELN(PASFIL2,'LZSLEFTEND(',labelnr,');');
2676          if mode='S-EXPR  ' then               (* Call By Need: Typ-0 Ergebnis *)
2677            writeln(pasfil2,'if need then T0update;');
2678          WRITELN(PASFIL2,'END; ');
2679          LABELNR:=LABELNR+1;
2680         END;
2681
2682      END;  (* OF PROCEDURE FUNCEXPRCONST  *)
2683
2684
2685
2686      (* ***********  LOKALE FORWARD DEKLARATIONEN        ******************** *)
2687
2688      PROCEDURE FUNCST(ADR,SNV,TYP:longint;MODEP:POMR;VAR RESMOD:strin);FORWARD;
2689      PROCEDURE IFST(H:POMR;TEST:BOOLEAN;VAR IFPART:CHAR);FORWARD;
2690
2691      (* ********************************************************************** *)
2692
2693
2694
2695
2696      PROCEDURE SEXPREXPR;
2697
```

```
2698      LABEL 1;                                    (* MARKE AM ENDE DER PROZEDUR*)
2699
2700      VAR
2701          RESULTMODE    :   strin;                (* RESULTMODE BEI FUNST.     *)
2702
2703
2704       FUNCTION ATOMSEXPR : BOOLEAN;
2705       (* DIE FUNKTION PRUEFT OB EIN ATOMARER S-AUSDRUCK VORLIEGT.             *)
2706
2707
2708       VAR          I   : longint;
2709
2710       BEGIN
2711
2712        I:=2;
2713        ATOMSEXPR:=TRUE;
2714        IF NOT(ISSYM[1] IN LETTER) THEN
2715                                    ATOMSEXPR:=FALSE
2716                                   ELSE
2717                                    REPEAT
2718                                     IF NOT(ISSYM[I] IN LETDIG)
2719                                           THEN ATOMSEXPR:=FALSE;
2720                                     I:=I+1
2721                                    UNTIL I=(SYMLENG+1);
2722
2723       END; (*  OF FUNCTION ATOMSEXPR  *)
2724
2725
2726       PROCEDURE SEXPR;
2727
2728       (* DIE PROZEDUR LIESST S-AUSDRUECKE,PRUEFT SIE UND BEREITET SIE FUER DIE *)
2729       (* LZS PROZEDUR INHEAP AUF.(S-AUSDRUECKE SIND PROGRAMMKOMNSTANTEN.)      *)
2730
2731       BEGIN
2732
2733        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2734        WRITE(PASFIL1,ISSYM);
2735        IF ISSYM='.       ' THEN FEHLER(51);
2736        WHILE ISSYM<>')       ' DO
2737            BEGIN
2738             WHILE ISSYM='(       ' DO
2739                    BEGIN
2740                     SEXPR;
2741                     nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2742                     WRITE(PASFIL1,ISSYM);
2743                    END;
2744             IF ISSYM<>')       ' THEN
2745                    BEGIN
2746                     IF ISSYM<>'.       ' THEN
2747                         BEGIN
2748                          IF NOT(ATOMSEXPR) THEN FEHLER(41);
2749                          nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2750                          WRITE(PASFIL1,ISSYM);
2751                         END
2752                        ELSE                      (* ISSYM='.  '              *)
2753                         BEGIN
2754                          nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2755                          WRITE(PASFIL1,ISSYM);
2756                          IF (ISSYM=')       ')
2757                           OR(ISSYM='.       ') THEN FEHLER(51);
2758                          IF  ISSYM='(       '  THEN SEXPR
2759                                                ELSE IF NOT(ATOMSEXPR)
2760                                                     THEN FEHLER(41);
2761                          nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2762                          WRITE(PASFIL1,ISSYM);
2763                          IF ISSYM<>')       ' THEN FEHLER(51);
2764                         END;
2765                    END;
2766                END;
2767
2768       END;  (* OF PROCEDURE SEXPR *)
2769
2770
2771
2772       PROCEDURE CONS;
```

```
2773       (*  CONS(<S-EXPR.EXPR.>,<S-EXPR.EXPR.>)                              *)
2774
2775       BEGIN
2776        LSCHACHT:=LSCHACHT+1;                      (* BEGINN EINES LINKEN ASTES *)
2777        la^.lklschacht:=la^.lklschacht+1;
2778        if la^.lklschacht=1                        (* Anfang eines RLA          *)
2779          then writeln(pasfil2,'rlacall:=false;');
2780        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2781        IF ISSYM<>'('       ' THEN FEHLER(05);
2782        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2783        SEXPREXPR;
2784        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2785        IF ISSYM<>',        ' THEN FEHLER(12);
2786        if la^.lklschacht=1                        (* Ende eines RLA            *)
2787          then writeln(pasfil2,'rlacall:=true;');
2788        la^.lklschacht:=la^.lklschacht-1;
2789        LSCHACHT:=LSCHACHT-1;                      (* ENDE EINES LINKEN ASTES   *)
2790        WRITELN(PASFIL2,'PUSH;');
2791        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2792        SEXPREXPR;
2793        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2794        IF ISSYM<>')        ' THEN FEHLER(09);
2795        WRITELN(PASFIL2,'CONS;');
2796
2797       END;  (* OF PROCEDURE CONS *)
2798
2799
2800       procedure pcons;   (* erzeugt Aufruf von pcons, ansonsten wie proc. cons *)
2801
2802       BEGIN
2803        LSCHACHT:=LSCHACHT+1;                      (* BEGINN EINES LINKEN ASTES *)
2804        la^.lklschacht:=la^.lklschacht+1;
2805        if la^.lklschacht=1                        (* Anfang eines RLA          *)
2806          then writeln(pasfil2,'rlacall:=false;');
2807        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2808        IF ISSYM<>'('       ' THEN FEHLER(05);
2809        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2810        SEXPREXPR;
2811        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2812        IF ISSYM<>',        ' THEN FEHLER(12);
2813        if la^.lklschacht=1                        (* Ende eines RLA            *)
2814          then writeln(pasfil2,'rlacall:=true;');
2815        la^.lklschacht:=la^.lklschacht-1;
2816        LSCHACHT:=LSCHACHT-1;                      (* ENDE EINES LINKEN ASTES   *)
2817        WRITELN(PASFIL2,'PUSH;');
2818        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2819        SEXPREXPR;
2820        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2821        IF ISSYM<>')        ' THEN FEHLER(09);
2822        WRITELN(PASFIL2,'pcons;')                     (* fuer permutiertes cons *)
2823
2824       END;  (* of procedure pcons *)
2825
2826
2827       PROCEDURE CARCDR;
2828       (*   CAR( <S-EXPR.EXPR> )    BZW.      CDR( <S-EXPR.EXPR.> )            *)
2829
2830       VAR       SYM   :  strin;                  (* ='CAR ' BZW. ='CDR  '     *)
2831
2832       BEGIN
2833
2834        SYM:=ISSYM;
2835        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2836        IF ISSYM<>'('       ' THEN FEHLER(05);
2837        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2838        SEXPREXPR;
2839        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2840        IF ISSYM<>')        ' THEN FEHLER(09);
2841        if sym='CAR     ' then writeln(pasfil2,'CAR;');
2842        if sym='CDR     ' then writeln(pasfil2,'CDR;')
2843       END;  (* OF PROCEDURE CARCDR *)
2844
2845
2846
2847       BEGIN
```

```
2848       (*<S-EXPR.EXPR.>::=<S-EXPR>/<FUNC.ST.>/CONS(<S-EXPR.EXPR.>,<S-EXPREXPR.>)*)
2849       (*                 /CDR(<S-EXPR.EXPR.>)/CAR(<S-EXPR.EXPR.>)/<FUNC.ST.>/  *)
2850       (*                 IF<BOOL.EXPR.>THEN<S-EXPR.EXPR.>ELSE<S-EXPR.EXPR.>FI  *)
2851
2852        IF (ISSYM='T       ')OR(ISSYM='F       ') THEN
2853                   BEGIN
2854                    IF ISSYM='T       ' THEN WRITELN(PASFIL2,'AC:=B[1];')
2855                                       ELSE WRITELN(PASFIL2,'AC:=B[2];');
2856                    GOTO 1;
2857                   END;
2858        IF ISSYM='"       ' THEN
2859                   BEGIN
2860                    nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2861                    KONSTZ:=KONSTZ+1;
2862                    IF NOT(ATOMSEXPR) THEN FEHLER(41);
2863                    WRITELN(PASFIL1,'INHEAP(''"',ISSYM,''',',KONSTZ,');');
2864                    WRITELN(PASFIL2,'AC:=B[',KONSTZ,'];');
2865                    GOTO 1;
2866                   END;
2867        IF ISSYM='(       ' THEN
2868                   BEGIN
2869                    KONSTZ:=KONSTZ+1;
2870                    WRITE(PASFIL1,'INHEAP(''(');
2871                    SEXPR;
2872                    WRITELN(PASFIL1,''',',KONSTZ,');');
2873                    WRITELN(PASFIL2,'AC:=B[',KONSTZ,'];');
2874                    GOTO 1
2875                   END;
2876        IF ISSYM='CONS    ' THEN
2877                   BEGIN
2878                    CONS;
2879                    GOTO 1
2880                   END;
2881        if issym='PCONS_  ' then                          (* neue Funktion pcons *)
2882                   begin
2883                    pcons;
2884                    goto 1
2885                   end;
2886        IF (ISSYM='CAR     ') OR (ISSYM='CDR     ')  THEN
2887                   BEGIN
2888                    CARCDR;
2889                    GOTO 1
2890                   END;
2891        IF ISSYM='IF      ' THEN
2892                   BEGIN
2893                    IFST(H,FALSE,IFPART);
2894                    IF (IFPART<>'S') THEN FEHLER(17);
2895                    GOTO 1
2896                   END;
2897        IF NOT(IDENTIFIER) THEN FEHLER(27);
2898        PACK(ISSYM,NUMBER,NAMENR);
2899        SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
2900        IF ISNEXTSYM='(       ' THEN               (* FUNKTIONSANWEISUNG        *)
2901                   BEGIN
2902                    FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
2903                    IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(37);
2904                   END
2905                  ELSE                               (* KONSTANTE BZW. FUNKT.AUSD.*)
2906                   BEGIN
2907                    IF MODE<>'S-EXPR  ' THEN       (* HIER NUR KONSTANTE ERLAUBT*)
2908                                        FEHLER(48);
2909                    FUNCEXPRCONST(TYP,ADR,STATNIV);
2910                   END;
2911     1:
2912
2913      END;  (* OF PROCEDURE SEXPREXPR *)
2914
2915
2916
2917        PROCEDURE BOOLEXPR;
2918        (* BOOLEXPR. IST STANDARDMAESSIG VOM MODE S-EXPR.                        *)
2919        LABEL 1;                                   (* MARKE AM PROZEDURENDE     *)
2920        VAR
2921              RESULTMODE : strin;                  (* RESULTMODE BEI FUNCST.    *)
2922
```

```
2923        PROCEDURE ATOM;
2924        (*  ATOM( <S-EXPR.EXPR.> )                                          *)
2925
2926        BEGIN
2927         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2928         IF ISSYM<>'(       ' THEN FEHLER(05);
2929         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2930         SEXPREXPR;
2931         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2932         IF ISSYM<>')       ' THEN FEHLER(09);
2933         WRITELN(PASFIL2,'ATOM;');
2934
2935        END;  (* OF PROCEDURE ATOM   *)
2936
2937
2938
2939        PROCEDURE EQ;
2940        (*  EQ( <S-EXPR.EXPR.> , <S-EXPR.EXPR.> )                          *)
2941
2942        BEGIN
2943         LSCHACHT:=LSCHACHT+1;                    (* BEGINN EINES LINKEN ASTES *)
2944         la^.lklschacht:=la^.lklschacht+1;
2945         if la^.lklschacht=1                      (* Anfang eines RLA          *)
2946           then writeln(pasfil2,'rlacall:=false;');
2947         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2948         IF ISSYM<>'(       ' THEN FEHLER(05);
2949         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2950         SEXPREXPR;
2951         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2952         IF ISSYM<>',       ' THEN FEHLER(12);
2953         if la^.lklschacht=1                      (* Ende eines RLA            *)
2954           then writeln(pasfil2,'rlacall:=true;');
2955         la^.lklschacht:=la^.lklschacht-1;
2956         LSCHACHT:=LSCHACHT-1;                    (* ENDE EINES LINKEN ASTES   *)
2957         WRITELN(PASFIL2,'PUSH;');
2958         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2959         SEXPREXPR;
2960         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
2961         IF ISSYM<>')       ' THEN FEHLER(09);
2962         WRITELN(PASFIL2,'EQ;');
2963
2964        END;  (* OF PROCEDURE EQ   *)
2965
2966
2967
2968       BEGIN                                     (* OF PROCEDURE BOOLEXPR     *)
2969       (*<BOOL.EXPR.>::=T/F/ATOM(<S-EXPR.EXPR.>)/EQ(<S-EXPR.EXPR>,<S-EXPR.EXPR>)*)
2970       (*               IF<BOOL.EXPR>THEN<BOOL.EXPR>ELSE<BOOL.EXPR.>FI/<FUNC.ST>*)
2971
2972        IF (ISSYM='T       ') OR (ISSYM='F       ') THEN
2973                   BEGIN
2974                    SEXPREXPR;
2975                    GOTO 1
2976                   END;
2977        IF ISSYM='ATOM    ' THEN
2978                   BEGIN
2979                    ATOM;
2980                    GOTO 1
2981                   END;
2982        IF ISSYM='EQ      ' THEN
2983                   BEGIN
2984                    EQ;
2985                    GOTO 1
2986                   END;
2987        IF ISSYM='IF      ' THEN
2988                   BEGIN
2989                    IFST(H,FALSE,IFPART);
2990                    IF IFPART<>'S' THEN FEHLER(17);
2991                    GOTO 1;
2992                   END;
2993        IF NOT (IDENTIFIER) THEN FEHLER(2);
2994        PACK(ISSYM,NUMBER,NAMENR);
2995        SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
2996        IF ISNEXTSYM='(       ' THEN              (* FUNKTIONSANWEISUNG       *)
2997                   BEGIN
```

```
2998                  FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
2999                  IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(40);
3000                 END
3001                ELSE                              (* FUNKT.AUSDR. / KONSTANTE  *)
3002                 BEGIN
3003                  IF MODE<>'S-EXPR  ' THEN FEHLER(48);        (* HIER NUR     *)
3004                  FUNCEXPRCONST(TYP,ADR,STATNIV);             (* KONST.ZUGEL. *)
3005                 END;
3006     1:
3007
3008       END;  (* OF PROCEDURE BOOLEXPR *)
3009
3010
3011
3012       PROCEDURE FUNCST; (*(ADR,SNV,TYP:longint;MODEP:POMR;VAR RESMOD:strin)    *)
3013
3014       VAR       GDVST    :    CHAR;               (*='L'/'R':LINKER/RECHTER AST*)
3015                 MIDFRECP :    POMIDFREC;          (* ZEIGER A.MODEREC.IM MODEB.*)
3016                 STN      :    longint;            (* ABKUERZUNG FUER STATNIV   *)
3017                 gmnivh1  :    longint;            (* Hilfsvariable             *)
3018
3019
3020        PROCEDURE ACTPARLIST;
3021
3022        VAR
3023               H   :  POMR;                        (* ZEIGER IN JEW. MODELISTE  *)
3024
3025
3026
3027         PROCEDURE ACTPAR( VAR H : POMR );
3028         (* ANALOG ZU PROCEDURE ACTPAR IN MAINPROGRAMM .                          *)
3029
3030         LABEL 1,                                  (* MARKE AM PROZEDURENDE     *)
3031               2;                                  (* MARKE FUER PROC.EXPR.BEH. *)
3032
3033         VAR
3034              ADR         :    longint;            (* START- BZW. RELATIVADR.   *)
3035              CARCONSCDR  :    BOOLEAN;            (* =TRUE,FALLS ISSYM=CAR...  *)
3036              HLABELNR    :    longint;            (* HAELT LABELNR F.DAS LABEL *)
3037                                                   (* F. PARAM.BESCHR.-FORTS.   *)
3038              IDFP        :    POIDF;              (*VERWEIS A.FUNKTIONSIDF.REC.*)
3039              MODE        :    strin;              (* MODENMAE EINES IDF.       *)
3040              MODEP       :    POMR;               (* ZEIGER AUF MODELISTE      *)
3041              NAMENR      :    PACKAR;             (* MIT NUMMER VERSEHENER NAME*)
3042              RESULTMODE  :    strin;              (* RESULTMODE BEI FUNCST.    *)
3043              STATNIV     :    SCHACHTINT ;        (* STATISCHES NIVEAU         *)
3044              TYP         :    EOZ ;               (* TYP EINES IDF.            *)
3045
3046         BEGIN
3047
3048          IF ISSYM='IF      ' THEN
3049                  BEGIN
3050                   HLABELNR:=LABELNR;
3051                   WRITELN(PASFIL2,'LZSPARB(2,',dpniv,',',LABELNR+1,');');
3052                   WRITELN(PASFIL2,'GOTO ',LABELNR,';');
3053                   WRITELN(PASFIL2,LABELNR+1,':');
3054                   LABELNR:=LABELNR+2;
3055                   IFST(H,TRUE,IFPART);
3056                   WRITELN(PASFIL2,'LZSFEND; goto 0;');
3057                   WRITELN(PASFIL2,HLABELNR,':');
3058                   H:=H^.NEXTP;
3059                   GOTO 1
3060                  END;
3061          IF (ISSYM='T       ')OR(ISSYM='F       ')
3062           OR(ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3063               BEGIN
3064                IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3065                IF ISNEXTSYM<>'(       ' THEN GOTO 2;     (* FUNKTIONS AUSDRUCK *)
3066                IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
3067                H:=H^.NEXTP;
3068                IF ISSYM='T       ' THEN
3069                 WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',1);')
3070                  ELSE
3071                   IF ISSYM='F       ' THEN
3072                    WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',2);')
```

```
3073                    ELSE                          (* ISSYM='EQ '/'ATOM '       *)
3074                     BEGIN
3075                      HLABELNR:=LABELNR;
3076                      WRITELN(PASFIL2,'LZSPARB(2,',dpniv,',',LABELNR+1,');');
3077                      WRITELN(PASFIL2,'GOTO ',LABELNR,';');
3078                      WRITELN(PASFIL2,LABELNR+1,':');
3079                      LABELNR:=LABELNR+2;
3080                      BOOLEXPR;
3081                      WRITELN(PASFIL2,'LZSFEND; goto 0;');
3082                      WRITELN(PASFIL2,HLABELNR,':')
3083                     END;
3084                GOTO 1
3085               END;
3086          IF (ISSYM='CAR     ')OR(ISSYM='CONS    ')OR(ISSYM='CDR     ')or
3087             (issym='PCONS_  ')
3088               THEN  CARCONSCDR:=TRUE
3089               ELSE  CARCONSCDR:=FALSE;
3090          IF (ISSYM='(       ')OR CARCONSCDR OR (ISSYM='"       ') THEN
3091               BEGIN
3092                IF CARCONSCDR THEN
3093                 IF ISNEXTSYM<>'(       ' THEN GOTO 2;  (* PROZEDURAUSDRUCK     *)
3094                IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
3095                H:=H^.NEXTP;
3096                IF NOT(CARCONSCDR) THEN            (* ISSYM= '"  ' /'(  '       *)
3097                    BEGIN
3098                     SEXPREXPR;
3099                     WRITELN(PASFIL2,'LZSPARB(0,',SCHACHT,',',KONSTZ,');');
3100                    END
3101                   ELSE                            (* ISSYM ='CONS'/'CDR'/'CAR' *)
3102                    BEGIN
3103                     HLABELNR:=LABELNR;
3104                     WRITELN(PASFIL2,'LZSPARB(2,',dpniv,',',LABELNR+1,');');
3105                     WRITELN(PASFIL2,'GOTO ',LABELNR,';');
3106                     WRITELN(PASFIL2,LABELNR+1,':');
3107                     LABELNR:=LABELNR+2;
3108                     SEXPREXPR;
3109                     WRITELN(PASFIL2,'LZSFEND; goto 0;');
3110                     WRITELN(PASFIL2,HLABELNR,':');
3111                    END;
3112                GOTO 1;
3113               END;
3114          IF NOT(IDENTIFIER) THEN FEHLER(2);
3115     2:   PACK(ISSYM,NUMBER,NAMENR);
3116          SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
3117          IF ISNEXTSYM='(       ' THEN              (* FUNKTIONSANWEISUNG        *)
3118              BEGIN
3119               HLABELNR:=LABELNR;
3120               WRITELN(PASFIL2,'LZSPARB(2,',dpniv,',',LABELNR+1,');');
3121               WRITELN(PASFIL2,'GOTO ',LABELNR,';');
3122               WRITELN(PASFIL2,LABELNR+1,':');
3123               LABELNR:=LABELNR+2;
3124               FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
3125               WRITELN(PASFIL2,'LZSFEND; goto 0;');
3126               WRITELN(PASFIL2,HLABELNR,':');
3127               IF RESULTMODE<>H^.MODEIDF THEN FEHLER(30);
3128               H:=H^.NEXTP;
3129              END
3130             ELSE                                  (* FUNKTIOMNSAUSDRUCK        *)
3131              BEGIN
3132               IF MODE<>H^.MODEIDF THEN FEHLER(30);
3133               H:=H^.NEXTP;
3134               IF TYP=1 THEN                       (* GEWOEHNLICHER FUNC.IDF.   *)
3135                         WRITELN(PASFIL2,'LZSPARB(1,',STATNIV,',',ADR,');')
3136                        ELSE                       (* FORMALER IDENTIFIKATOR    *)
3137                         WRITELN(PASFIL2,'LZSPARB(3,',STATNIV,',',ADR,');')
3138              END;
3139     1:
3140           END; (* OF PROCEDURE ACTPAR *)
3141
3142
3143
3144          BEGIN                                    (* OF PROCEDURE ACTPARLIST   *)
3145
3146           H:=MODEP;
3147           laold:=la; new(la);                     (* Neuer LA auf akt. P-Pos.  *)
```

```
3148         la^.pred:=laold; la^.lklschacht:=0;       (* moeglich. Push LKlschacht.*)
3149         nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3150         WHILE ISSYM<>')       ' DO
3151                   BEGIN
3152                    IF H=NIL THEN FEHLER(38);
3153                    ACTPAR(H);
3154                    nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3155                    IF ISSYM<>')       ' THEN
3156                           IF ISSYM<>',       '
3157                             THEN FEHLER(12)
3158                             ELSE nextgmcodesym(ISSYM,ISNEXTSYM,gmniv,dpniv)
3159                   END;
3160         IF H<>NIL THEN FEHLER(38);
3161
3162         laold:=la; la:=la^.pred; dispose(laold)   (* LKlschacht popen          *)
3163
3164        END;  (* OF PROCEDURE ACTPARLIST *)
3165
3166
3167
3168       BEGIN                                      (* OF PROCEDURE FUNCST       *)
3169
3170
3171        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3172        IF ISSYM <>'(       ' THEN FEHLER(05);
3173        gdvstatus(LSCHACHT,gdvst);
3174        gmnivh1:=gmniv;                                    (* Retten von GMNIV *)
3175        ACTPARLIST;
3176        IF (MODEP = NIL)                            (*  LEERE PARAMETERLISTE   *)
3177              AND
3178           (RESULTMODE <> 'S-EXPR  ')
3179              AND
3180           (RESULTMODE <> 'VOID    ')               (*  FUNKTIONALES ERGEBNIS  *)
3181              AND
3182           (ISNEXTSYM <> '(       ')               (* KEINE PENDING PARAMETER *)
3183         THEN
3184                       (* evtl. anfallende PP's werden nicht zu akt. Parametern *)
3185           WRITELN(PASFIL2,'LZSTRENN;')
3186       (*FI*);
3187         WHILE ISNEXTSYM='(       ' DO               (* PENDING PARAMETER LESEN *)
3188                   BEGIN
3189                    WRITELN(PASFIL2,'LZSTRENN;');
3190                    SEARCHM(RESMOD,MIDFRECP);
3191                    RESMOD:=MIDFRECP^.RESULTMOD;
3192                    nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3193                    MODEP:=MIDFRECP^.MODEP;
3194                    ACTPARLIST
3195                   END;
3196         IF TYP=1 THEN                              (* GEW.FUNKTIONSIDENTIFIKATOR*)
3197         WRITELN(PASFIL2,
3198        'LZSARK(0,',ADR,',',LABELNR,',',SNV,',''',GDVST,''',',gmnivh1,'); goto 0;')
3199         ELSE                                       (* FORMALER IDENTIFIKATOR    *)
3200           BEGIN
3201            WRITELN(PASFIL2,'LZSFORMIDF(',SNV,',',ADR,',TYP,ADR);');
3202            WRITELN(PASFIL2,'IF TYP=0 THEN FEHLER(13);');
3203                                                 (* Call By Need: Typ-1 Ergebnis *)
3204            writeln(pasfil2,'if need then T1update:=true;');
3205            WRITELN(PASFIL2,
3206       'LZSARK(typ,',ADR,',',LABELNR,',',SNV,',''',GDVST,''',',gmnivh1,'); goto 0;')
3207           END;
3208         WRITELN(PASFIL2,LABELNR,':');
3209         IF GDVST='L' THEN WRITELN(PASFIL2,'LZSLEFTEND(',labelnr,');');
3210         LABELNR:=LABELNR+1;
3211
3212       END;  (* OF PROCEDURE FUNCST *)
3213
3214
3215
3216       PROCEDURE IFST ; (* H : POMR,TEST : BOOLEAN;VAR IFPART : CHAR            *)
3217       (* ANAOLAG ZU IFST IN MAINPROGRAM                                       *)
3218
3219       LABEL 1,                                   (* MARKE AM ENDE DES IF-TEILS*)
3220             2,                                   (* MARKE AM ENDE DER PROZEDUR*)
3221             3,                                   (* M: PROC.EXPR. IM IF-TEIL  *)
3222             4;                                   (* M: PROC.EXPR. IM ELSE-TEIL*)
```

```
3223
3224       VAR
3225            CARCONSCDR      :    BOOLEAN;          (* =TRUE,FALLS ISSYM=CAR/....*)
3226            HIFPART         :    CHAR;             (* IFPART BEI REKURSION      *)
3227            RESULTMODE      :    strin;            (* RESULTMODE BEI FUNCST.    *)
3228
3229       BEGIN
3230        LSCHACHT:=LSCHACHT+1;                      (* BEGINN EINES LINKEN ASTES *)
3231        la^.lklschacht:=la^.lklschacht+1;
3232        if la^.lklschacht=1
3233          then writeln(pasfil2,'rlacall:=false;'); (* Anfang eines RLA          *)
3234        IFCOUNT:=IFCOUNT+1;                        (* BEGINN EINES IF-PARTS     *)
3235        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3236        BOOLEXPR;
3237        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3238        IF ISSYM<>'THEN    ' THEN FEHLER(31);
3239        if la^.lklschacht=1                        (* Ende eines RLA            *)
3240          then writeln(pasfil2,'rlacall:=true;');
3241        WRITELN(PASFIL2,'IF BOOLTEST THEN BEGIN ');
3242        la^.lklschacht:=la^.lklschacht-1;
3243        LSCHACHT:=LSCHACHT-1;                      (* ENDE EINES LINKEN ASTES   *)
3244        IFCOUNT:=IFCOUNT-1;                        (* ENDE EINES IF-PARTS       *)
3245        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3246        IF ISSYM='IF      ' THEN
3247              BEGIN
3248               IFST(H,TEST,IFPART);
3249               GOTO 1 ;
3250              END;
3251        IF (ISSYM='CAR     ')OR(ISSYM='CONS    ')OR(ISSYM='CDR     ')or
3252           (issym='PCONS_  ')
3253              THEN CARCONSCDR:=TRUE
3254              ELSE CARCONSCDR:=FALSE;
3255        IF CARCONSCDR OR(ISSYM='(       ')OR(ISSYM='"       ') THEN
3256              BEGIN
3257               IF CARCONSCDR THEN
3258                IF ISNEXTSYM<>'(       ' THEN GOTO 3;  (* PROZEDUR AUSDRUCK    *)
3259               IF IFCOUNT=0 THEN
3260                    BEGIN
3261                     IF TEST THEN                  (* AUFRUF ERFOLGTE AUS ACTPAR*)
3262                         BEGIN
3263                          IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
3264                         END
3265                        ELSE                       (* THEN/ELSE-PART OF EXPR.   *)
3266                         IF FUNCRESMODE<>'S-EXPR  ' THEN FEHLER(45)
3267                    END;
3268               SEXPREXPR;
3269               IFPART:='S';
3270               GOTO 1;
3271              END;
3272        IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ')
3273         OR(ISSYM='T       ')OR(ISSYM='F       ')THEN
3274              BEGIN
3275               IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3276                IF ISNEXTSYM<>'(       ' THEN GOTO 3;  (* FUNKTIONSAUSDRUCK    *)
3277               IF IFCOUNT=0 THEN
3278                       IF TEST THEN
3279                            BEGIN
3280                             IF H^.MODEIDF<>'S-EXPR  ' THEN FEHLER(30);
3281                            END
3282                           ELSE
3283                            IF FUNCRESMODE<>'S-EXPR  ' THEN FEHLER(45);
3284               BOOLEXPR;
3285               IFPART:='S';
3286               GOTO 1;
3287              END;
3288        IF NOT(IDENTIFIER) THEN FEHLER(2);
3289   3:   PACK(ISSYM,NUMBER,NAMENR);
3290        SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
3291        IF ISNEXTSYM='(       ' THEN                   (* FUNKTIONS ANWEISUNG   *)
3292            BEGIN
3293             FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
3294             IF IFCOUNT=0 THEN
3295                 BEGIN
3296                  IF TEST THEN
3297                       BEGIN
```

```
3298                      IF H^.MODEIDF<>RESULTMODE THEN FEHLER(30);
3299                     END
3300                    ELSE
3301                     IF RESULTMODE<>FUNCRESMODE THEN FEHLER(47);
3302                END
3303               ELSE                                (* IN EINEM IF-PART          *)
3304                IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(40);
3305             IF RESULTMODE='S-EXPR  ' THEN IFPART:='S'
3306                                      ELSE IFPART:='P';
3307             GOTO 1
3308            END
3309           ELSE                                    (* FUNKTIONS AUSDRUCK        *)
3310            BEGIN
3311             IF IFCOUNT>0 THEN                     (* IM IF-PART : S-EXPR ERW.  *)
3312               BEGIN
3313                IF MODE<>'S-EXPR  ' THEN FEHLER(48);
3314                IFPART:='S';
3315               END
3316              ELSE
3317               BEGIN
3318                IF TEST THEN
3319                    BEGIN
3320                     IF H^.MODEIDF<>MODE THEN FEHLER(30);
3321                    END
3322                   ELSE
3323                    BEGIN
3324                     IF MODE<>FUNCRESMODE THEN FEHLER(47);
3325                     IF MODE='S-EXPR  ' THEN
3326                                       IFPART:='S'
3327                                      ELSE
3328                                       IFPART:='E'
3329                    END
3330               END;
3331             FUNCEXPRCONST(TYP,ADR,STATNIV);
3332            END;
3333
3334    1:  nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3335        IF ISSYM<>'ELSE    ' THEN FEHLER(32);
3336        WRITELN(PASFIL2,'END ELSE BEGIN ');
3337        nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3338        IF ISSYM='IF      ' THEN
3339             BEGIN
3340              HIFPART:=IFPART;
3341              IFST(H,TEST,IFPART);
3342              IF IFPART<>HIFPART THEN FEHLER(33);
3343              GOTO 2
3344             END;
3345        IF (ISSYM='CAR     ')OR(ISSYM='CONS    ')OR(ISSYM='CDR     ')or
3346           (issym='PCONS_  ')
3347             THEN CARCONSCDR:=TRUE
3348             ELSE CARCONSCDR:=FALSE;
3349        IF CARCONSCDR OR(ISSYM='(       ')OR(ISSYM='"       ') THEN
3350             BEGIN
3351              IF CARCONSCDR THEN
3352               IF ISNEXTSYM<>'(       ' THEN GOTO 4;    (* FUNKTIONSAUSDRUCK    *)
3353              IF IFPART<>'S' THEN FEHLER(33);
3354                             (* HIER KEINE ABFRAGEN "TEST=TRUE" BZW. "IFCOUNT=0"*)
3355                             (* NOETIG,DA DIESE BEREITS IM THEN-PART DURCHLAUFEN*)
3356              SEXPREXPR;
3357              GOTO 2
3358             END;
3359        IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ')
3360         OR(ISSYM='T       ')OR(ISSYM='F       ')THEN
3361             BEGIN
3362              IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3363               IF ISNEXTSYM<>'(       ' THEN GOTO 4;    (* FUNKTIONSAUSDRUCK    *)
3364              IF IFPART<>'S' THEN FEHLER(33);
3365              BOOLEXPR;
3366              GOTO 2
3367             END;
3368        IF NOT(IDENTIFIER) THEN FEHLER(2);
3369    4:  PACK(ISSYM,NUMBER,NAMENR);
3370        SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
3371        IF ISNEXTSYM='(       ' THEN
3372             BEGIN
```

```
3373             IF (IFPART<>'P')AND(IFPART<>'S') THEN FEHLER(33);
3374             FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
3375             IF IFCOUNT=0 THEN
3376                 BEGIN
3377                  IF TEST THEN
3378                    BEGIN
3379                     IF H^.MODEIDF<>RESULTMODE THEN FEHLER(30);
3380                    END
3381                  ELSE
3382                    IF RESULTMODE<>FUNCRESMODE THEN FEHLER(47);
3383                 END
3384                ELSE
3385                 IF RESULTMODE<>'S-EXPR  ' THEN FEHLER(40);
3386                 GOTO 2;
3387            END
3388           ELSE                                    (* FUNKTIONSAUSDRUCK         *)
3389            BEGIN
3390             IF IFCOUNT>0 THEN
3391                 BEGIN
3392                  IF MODE<>'S-EXPR  ' THEN FEHLER(48);
3393                  IF IFPART<>'S' THEN FEHLER(33)
3394                 END
3395                ELSE
3396                 BEGIN
3397                  IF TEST THEN
3398                    BEGIN
3399                     IF H^.MODEIDF<>MODE THEN FEHLER(30);
3400                    END
3401                   ELSE
3402                    IF MODE<>FUNCRESMODE  THEN FEHLER(47)
3403                 END;
3404             FUNCEXPRCONST(TYP,ADR,STATNIV);
3405            END;
3406   2:  nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3407       IF ISSYM<>'FI      ' THEN FEHLER(36);
3408       WRITELN(PASFIL2,'END;');
3409
3410      END;     (*  OF PROCEDURE IFST  *)
3411
3412
3413
3414     BEGIN                                       (* OF PROCEDURE EXPRESSION   *)
3415
3416     (*<EXPRESSION>::=<EMPTY>/<BOOL.EXPR>/<S-EXPR.EXPR>/<FUNC.EXPR>/<FUNC.ST.> *)
3417
3418      new(la); la^.pred:=nil; la^.lklschacht:=0;  (* LKlschacht einrichten     *)
3419      LSCHACHT:=0;
3420      nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3421      IF ISSYM<>'}       ' THEN                   (* EXPRESSION NOT EMPTY      *)
3422       BEGIN
3423        IFCOUNT:=0;
3424        IF (ISSYM='CAR     ')OR(ISSYM='CONS    ')OR(ISSYM='CDR     ')or
3425           (issym='PCONS_  ')
3426             THEN CARCONSCDR:=TRUE
3427             ELSE CARCONSCDR:=FALSE;
3428        IF (ISSYM='(       ')OR CARCONSCDR OR(ISSYM='"       ') THEN
3429             BEGIN
3430              IF CARCONSCDR THEN
3431               IF ISNEXTSYM<>'(       ' THEN GOTO 2;   (* FUNKTIONSAUSDRUCK    *)
3432              IF FUNCRESMODE='S-EXPR  ' THEN SEXPREXPR ELSE FEHLER(45);
3433              GOTO 1
3434             END;
3435        IF (ISSYM='T       ')OR(ISSYM='F       ')
3436         OR(ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3437             BEGIN
3438              IF (ISSYM='EQ      ')OR(ISSYM='ATOM    ') THEN
3439               IF ISNEXTSYM<>'(       ' THEN GOTO 2;   (* FUNKTIONSAUSDRUCK    *)
3440              IF FUNCRESMODE='S-EXPR  ' THEN BOOLEXPR ELSE FEHLER(45);
3441              GOTO 1
3442             END;
3443        IF ISSYM='IF      ' THEN
3444             BEGIN
3445              IFST(H,FALSE,IFPART);
3446              GOTO 1
3447             END;
```

```
3448         IF NOT(IDENTIFIER) THEN FEHLER(2);
3449   2:    PACK(ISSYM,NUMBER,NAMENR);
3450         SEARCH(NAMENR,STATNIV,TYP,ADR,MODEP,MODE,RESULTMODE);
3451         IF ISNEXTSYM='(       ' THEN                  (* FUNKTIONSANWEISUNG    *)
3452              BEGIN
3453               FUNCST(ADR,STATNIV,TYP,MODEP,RESULTMODE);
3454               IF RESULTMODE<>FUNCRESMODE THEN FEHLER(47)
3455                                          ELSE GOTO 1
3456              END
3457             ELSE                                  (* FUNKT.AUSDRUCK/KONSTANTE  *)
3458              BEGIN
3459               IF MODE<>FUNCRESMODE THEN FEHLER(47);
3460               FUNCEXPRCONST(TYP,ADR,STATNIV) ;
3461              END;
3462   1:     nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3463        END;
3464
3465       dispose(la)                                 (* Var. LKlschacht freigeben *)
3466
3467      END; (* OF EXPRESSION *)
3468
3469
3470     BEGIN                                        (* OF PROCEDURE FUNCDECL2    *)
3471     (* <FUNCDECL2>::=**FUNC** <FIDF> <FUNCRESMODE>;{<EMPTY>/<FUNCDECLPART2>    *)
3472     (*                <EXPRESSION>]                                            *)
3473
3474      nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3475      PACK(ISSYM,NUMBER,NAMENR);
3476      SEARCH(NAMENR,STATNIV,TYP,STARTADR,MODEP,MODE,FUNCRESMODE);
3477      ENTER(SCHACHT,NUMBER,HNUMBER);              (*BEGINN DES FUNKTIONSRUMPFES*)
3478      nextgmcodesym(issym,isnextsym,gmniv,dpniv);  (* UEBERLIESST ";"           *)
3479      nextgmcodesym(issym,isnextsym,gmniv,dpniv);  (* UEBERLIESST "{"           *)
3480      FUNCDECLPART2;
3481      WRITELN(PASFIL2,STARTADR,':');
3482      EXPRESSION;
3483      LEAVE(SCHACHT,NUMBER,HNUMBER);              (* ENDE DES FUNKTIONSRUMPFES *)
3484      IF ISSYM<>'}       ' THEN FEHLER(14);
3485      WRITELN(PASFIL2,'LZSFEND; goto 0;')
3486     END;    (* OF PROCEDURE FUNCDECL2  *)
3487
3488
3489   procedure funcdeclpart2;
3490   (*  <funcdeclpart2> ::= **EMPTY* / <funcdecl2> [<funcdecl2>]                 *)
3491     begin
3492       IF ISNEXTSYM<>'**EMPTY*' THEN
3493                WHILE ISNEXTSYM='**FUNC**' DO
3494                      BEGIN                       (*   **FUNC**  WIRD GELESEN   *)
3495                       nextgmcodesym(issym,isnextsym,gmniv,dpniv);
3496                       FUNCDECL2;
3497                      END;
3498     end;  { of procedure funcdclpart2 }
3499
3500
3501   procedure gpmark;
3502
3503   (* Markierungs- und Permutationsalgorithmus GPMARK zur Optimierung des       *)
3504   (* Verweises 'Generalisierter Dynamischer Vorgaenger' (kurz: GDV) sowie      *)
3505   (* der Handhabung Dicker Parameter.                                         *)
3506   (*                                                                          *)
3507   (* Permutation: Ist der erste Parameter von CONS, bzw. EQ kein konstanter    *)
3508   (*              S-Ausdruck (PK3), und ist das max statische Niveau im RLK vom*)
3509   (*              zweiten Parameter echt groesser als das max. statische Niveau*)
3510   (*              im rlk vom ersten Parameter, falls dieser mit dem zweiten    *)
3511   (*              Parameter getauscht waere (PK1), und verschlechtert sich der *)
3512   (*              GDV-Verweis vom zweiten Parameter durch den Parametertausch  *)
3513   (*              nicht (PK2), so werden die Parameter von CONS, bzw. EQ       *)
3514   (*              getauscht. Werden die Parameter von CONS getauscht, so wird  *)
3515   (*              CONS in PCONS umbenannt.                                     *)
3516   (*                                                                          *)
3517   (* GDV-Opt.:    Jeder Nichtstandardidentifikator I im linken Ast vom         *)
3518   (*              Anweisungsteil einer NSF wird mit einer Markierung versehen, *)
3519   (*              die das maximale statische Niveau der NS-Identifikatoren im  *)
3520   (*              'relevanten lokalen Kontext' von I (kurz: rlk(I)) angibt.    *)
3521   (*              Falls der rlk(I) leer ist, wird I mit '0' markiert.          *)
3522   (*                                                                          *)
```

```
3523  (* DP-Opt.:     Jeder dicke Parameter d eines NSF-Aufrufes wird mit          *)
3524  (*              max (stat. Verweise in d , 0) + 100  markiert.               *)
3525  (*                                                                           *)
3526  (* Alle Markierungen werden in Kommentarklammern eingeschlossen.             *)
3527
3528
3529  type exptr   = ^exlist;                      (* zeigt auf aktuelles Symbol  *)
3530       varptr  = ^varlist;                     (* akt. Liste der Idf's im rlk *)
3531       exlist  = record
3532                   sym : strin;               (* Symbol aus ZWCODE           *)
3533                   niv : schachtint;          (* Staisches Niveau vom Symbol *)
3534                   next: exptr                (* Nachfolger                  *)
3535                 end;
3536       varlist = record
3537                   prev: varptr;              (* Vorgaenger                  *)
3538                   sym : strin;               (* Symbol im rlk               *)
3539                   niv : schachtint;          (* Staisches Niveau vom Symbol *)
3540                 end;
3541
3542  var exroot : exptr;   (* Anfang der Liste mit Zeichen und Niveaus aus ZWCODE *)
3543      exinp  : exptr;   (* Hilfsvariable zum Anlegen der Liste                *)
3544      vars   : varptr;  (* Zeiger auf rlk                                     *)
3545
3546
3547  procedure einlesen;
3548
3549  (* Einlesen der Symbole aus ZWCODE in eine lineare Liste. Dabei wird unter   *)
3550  (* Benutzung der Proceduren PACK, SEARCH, LEAVE und ENTER (s.o.) das Niveau  *)
3551  (* jedes Identifikators mitabgespeichert.                                    *)
3552
3553    VAR                                        (* Variablen Fuer Proc.SEARCH*)
3554        FUNCRESMODE   :  strin;                (* RESULTMODE DER FUNKTION   *)
3555        MODE          :  strin;                (* MODE DES FUNKT.IDENTIF.   *)
3556        MODEP         :  POMR;                 (* ZEIGER A.MODELISTE ZU MODE*)
3557        NAMENR        :  PACKAR;               (* MIT NR VERSEHENER IDF.NAME*)
3558        STARTADR      :  longint;              (* STARTADRESSE DER FUNKTION *)
3559        STATNIV       :  SCHACHTINT;           (* STATISCHES NIVEAU DER  "  *)
3560        TYP           :  EOZ;                  (* TYP BEI SEARCH -HIER RED. *)
3561
3562    procedure anfuegen(symbol: strin; statniv: schachtint);
3563      (* Haengt ein Symbol und sein statisches Niveau an die Symbolliste an.   *)
3564      (* Ist das Symbol kein Nstd.Idf., so wird als Niveau '0' eingetragen.    *)
3565      var ex2: exptr;                                      (* Hilfsvariable *)
3566      begin
3567        new(ex2); ex2^.next:=nil; exinp^.next:=ex2;
3568        exinp^.sym:=symbol; exinp^.niv:=statniv; exinp:=ex2
3569      end;
3570
3571    procedure liesfunc; forward;
3572
3573    procedure rumpf;                         (* Einlesen eines Funktionsrumpfes *)
3574      var kl: longint;
3575      begin
3576        nextcodesym(issym,isnextsym);                (* Lesen Funktions-Name  *)
3577        pack(issym,number,namenr);                   (* Ermitteln von statniv *)
3578        search(namenr,statniv,typ,startadr,modep,mode,funcresmode);
3579        anfuegen(issym,statniv);
3580        enter(schacht,number,hnumber);
3581        nextcodesym(issym,isnextsym); anfuegen(issym,0);          (* Lesen ';' *)
3582        nextcodesym(issym,isnextsym); anfuegen(issym,0);          (* Lesen '{' *)
3583        liesfunc;                                    (* evtl. geschachtelte Ruempfe *)
3584        while isnextsym<>'}       ' do
3585          begin
3586            nextcodesym(issym,isnextsym);
3587            if issym='"       ' then
3588              begin
3589                anfuegen(issym,0);                        (* Lesen eines Atoms *)
3590                nextcodesym(issym,isnextsym); anfuegen(issym,0);
3591                nextcodesym(issym,isnextsym);
3592              end;
3593            if (issym<>'IF      ') and (issym<>'THEN    ') and
3594               (issym<>'ELSE    ') and (issym<>'FI      ') and
3595               (issym<>',       ') and (issym<>'(       ') and
3596               (issym<>')       ') and (issym<>'F       ') and
3597               (issym<>'T       ')
```

```
3598              then begin
3599                     pack(issym,number,namenr);
3600                     search(namenr,statniv,typ,startadr,modep,mode,funcresmode);
3601                     anfuegen(issym,statniv)               (* Lesen eines Idf. *)
3602                   end
3603              else begin
3604                     anfuegen(issym,0);           (* Lesen eines 'Sonder-Idf.' *)
3605                     if isnextsym='(       ' then
3606                       begin                              (* Lesen einer Liste *)
3607                         kl:=1;
3608                         nextcodesym(issym,isnextsym); anfuegen(issym,0);
3609                         while kl>0 do
3610                           begin
3611                              nextcodesym(issym,isnextsym); anfuegen(issym,0);
3612                              if issym='(       ' then kl:=kl+1;
3613                              if issym=')       ' then kl:=kl-1
3614                           end
3615                       end
3616                   end
3617          end;
3618        nextcodesym(issym,isnextsym); anfuegen(issym,0);          (* Lesen ';' *)
3619        leave(schacht,number,hnumber)
3620      end; (* of rumpf *)
3621
3622    procedure liesfunc;           (* Lesen von '**FUNC**' und Aufruf von RUMPF *)
3623      begin
3624        while isnextsym='**FUNC**'do
3625          begin
3626            nextcodesym(issym,isnextsym); anfuegen(issym,0);
3627            rumpf
3628          end
3629      end;
3630
3631
3632    begin (* of einlesen *)
3633      liesfunc;
3634      nextcodesym(issym,isnextsym);
3635      if isnextsym='**MAIN**' then
3636        begin
3637          (* Kosmetik: Tauschen '**MAIN**' <--> 1. Symbol von Main            *)
3638          anfuegen(isnextsym,0); anfuegen(issym,0);
3639          nextcodesym(issym,isnextsym);
3640          repeat                              (* Lesen des Hauptprogrammes Main *)
3641            nextcodesym(issym,isnextsym); anfuegen(issym,0)
3642          until eof(zwcode)
3643        end
3644    end;  (* of einlesen *)
3645
3646
3647  procedure ausgabe;              (* Ausgabe vom markierten ZWCODE in GMZWCODE *)
3648    var ex  : exptr;
3649        symz: longint;
3650    begin
3651      rewrite(gmzwcode); writeln(gmzwcode); ex:=exroot; symz:=0;
3652      while ex^.next<>nil do
3653        begin
3654          if symz<8 then symz:=symz+1
3655                    else begin symz:=1; writeln(gmzwcode) end;
3656          if symz<8 then write(gmzwcode,ex^.sym,' ')
3657                    else write(gmzwcode,ex^.sym);
3658          ex:=ex^.next
3659        end;
3660      close(gmzwcode)
3661    end; (* of ausgabe *)
3662
3663
3664  function noend(l: exptr): boolean;
3665  (* Da GPMARK vor dem zweiten Lauf des Compilers arbeitet, ist die Syntax in  *)
3666  (* den Funktionsruempfen noch nicht ueberprueft worden. Falls schliessende   *)
3667  (* Klammern oder ein THEN, bzw. ELSE fehlen, erfolgt Fehlermeldung.          *)
3668    begin
3669      if l^.next=nil
3670        then begin                            (* Ende der Programm-Liste       *)
3671               isnextsym:=' <EOF>  ';
3672               fehler(55)                          (* Fehleraustritt           *)
```

```
3673              end
3674          else noend:=true
3675      end;
3676
3677
3678    function condgo(l: exptr; obj: strin): exptr;
3679    (* Sprung zum zugehoerigen 'THEN' bzw. 'ELSE' einer 'IF'-Anweisung           *)
3680      begin
3681        while (l^.sym<>obj) and noend(l) do
3682          if l^.sym='IF      ' then l:=condgo(l^.next,obj)
3683                               else l:=l^.next;
3684        condgo:=l^.next
3685      end;
3686
3687
3688    function listendgo(l: exptr): exptr;       (* Sprung an das Ende einer Liste *)
3689      begin
3690        while (l^.sym<>')       ') and noend(l) do
3691          if l^.sym='(       ' then l:=listendgo(l^.next)^.next
3692                               else l:=l^.next;
3693        listendgo:=l
3694      end;
3695
3696
3697    function nextparago(l: exptr): exptr;
3698    (* Sprung zum naechsten Parameter einer aktuellen Parameter-Liste, bzw.      *)
3699    (* Sprung an das Ende der Liste, falls keine Parameter mehr folgen.          *)
3700
3701      begin (* of nextparago *)
3702        while (l^.sym<>',       ') and (l^.sym<>')       ') and noend(l) do
3703          if l^.sym='(       ' then l:=listendgo(l^.next)^.next
3704                               else l:=l^.next;
3705        if l^.sym=',       ' then nextparago:=l^.next
3706                             else nextparago:=l
3707      end; (* of nextparago *)
3708
3709
3710    function funcgo(l: exptr): exptr;  (* Sprung zum Ende eines Funktionsrumpfes *)
3711      begin
3712        while (l^.sym<>'}       ') and noend(l) do
3713          if l^.sym='**FUNC**' then l:=funcgo(l^.next)
3714                               else l:=l^.next;
3715        funcgo:=l^.next
3716      end;
3717
3718
3719    function max(l: varptr): longint;             (* Maximales stat. Niv. im rlk *)
3720      var m: longint;
3721      begin
3722        if (l^.sym='ATOM    ') or (l^.sym='CAR     ') or (l^.sym='CDR     ') or
3723           (l^.sym='CONS    ') or (l^.sym='EQ      ') or (l^.sym='PCONS_  ')
3724          then m:=0
3725          else m:=l^.niv;
3726        while l^.prev<>nil do
3727          begin
3728            l:=l^.prev;
3729            if (l^.sym<>'ATOM    ') and (l^.sym<>'CAR     ') and
3730               (l^.sym<>'CDR     ') and (l^.sym<>'CONS    ') and
3731               (l^.sym<>'EQ      ') and (l^.sym<>'PCONS_  ')
3732              then if l^.niv>m then m:=l^.niv
3733          end; max:=m
3734      end;
3735
3736    function union(l1,l2: varptr): varptr;
3737    (* Vereinigung von zwei rlk-Listen; Ergebnis ist eine eigene Liste.          *)
3738
3739      var l,h,r: varptr;
3740      begin
3741        new(l); l^.prev:=nil; l^.sym:=l2^.sym; l^.niv:=l2^.niv; union:=l;
3742        while l2^.prev<>nil do
3743          begin
3744            l2:=l2^.prev; h:=l; new(l); l^.prev:=nil;
3745            h^.prev:=l; l^.sym:=l2^.sym; l^.niv:=l2^.niv
3746          end; r:=l;
3747        new(l); l^.prev:=nil; l^.sym:=l1^.sym; l^.niv:=l1^.niv; r^.prev:=l;
```

```
3748      while l1^.prev<>nil do
3749        begin
3750          l1:=l1^.prev; h:=l; new(l); l^.prev:=nil;
3751          h^.prev:=l; l^.sym:=l1^.sym; l^.niv:=l1^.niv
3752        end
3753    end; (* of union *)
3754
3755
3756  procedure permutation(l: exptr);
3757  (* Tauscht Parameter von CONS, bzw. EQ, wobei CONS in PCONS umbenannt wird.  *)
3758
3759    var h1,h2,h3,h4,h5: exptr;
3760
3761    function paraendgo(l: exptr): exptr;
3762    (* Sprung zum Ende eines Parameters einer aktuellen Parameter-Liste.       *)
3763      begin
3764        while (l^.next^.sym<>',       ') and
3765              (l^.next^.sym<>')       ') and noend(l^.next) do
3766          if l^.next^.sym='(       ' then l:=listendgo(l^.next^.next)
3767                                     else l:=l^.next;
3768        paraendgo:=l
3769      end;
3770
3771    begin  (* of permutation *)
3772      if l^.sym='CONS    '                    (* CONS wird in PCONS umbenannt *)
3773        then begin
3774               l^.sym:='PCONS_  ';
3775               writeln('               CONS permutiert...')
3776             end
3777        else   writeln('               EQ   permutiert...');
3778                      (* Permutieren der Parameter durch "Umhangeln" der Liste *)
3779      h1:=l^.next^.next;  h2:=paraendgo(h1); h3:=h2^.next;
3780      h4:=h3^.next; h5:=paraendgo(h4); l^.next^.next:=h4;
3781      h3^.next:=h1; h2^.next:=h5^.next; h5^.next:=h3
3782    end;  (* of permutation *)
3783
3784
3785  function constcond(l: exptr): boolean; forward;
3786
3787
3788  function constant(l: exptr): boolean;
3789  (* Liefert nur dann true, wenn sicher ein konstanter Ausdruck vorliegt.      *)
3790    begin
3791      if l^.sym='IF      '
3792        then constant:=constcond(l^.next)          (* if-Struktur ueberpruefen *)
3793        else
3794      if l^.next^.sym='(       '                   (* es liegt Aufruf vor      *)
3795        then if (l^.sym='CAR     ') or (l^.sym='CDR     ') or (l^.sym='ATOM    ')
3796                               (* Argument(e) der Standardfunktion ueberpruefen *)
3797               then constant:=constant(l^.next^.next)
3798               else if (l^.sym='CONS    ') or (l^.sym='EQ      ')
3799                      then constant:=constant(l^.next^.next) and
3800                                     constant(nextparago(l^.next^.next))
3801                      else constant:=false       (* Nstd.-Fkt.-Aufruf erreicht *)
3802        else if (l^.sym<>'"       ') and (l^.sym<>'(       ') and
3803                (l^.sym<>'T       ') and (l^.sym<>'F       ')
3804               then constant:=false              (* NS-Idf: NSF-Aufruf moeglich*)
3805               else constant:=true               (* Konstanter S-Ausdruck      *)
3806    end;
3807
3808
3809  function constcond(l: exptr): boolean;
3810
3811  (* Liefert nur dann true, wenn IF-, ELSE- und THEN-Zweige konstant sind. Die *)
3812  (* Funktion erkennt nicht den Fall, dass der IF-Zweig konstant true bzw.     *)
3813  (* konstant false liefert, um dann nur noch den THEN- bzw. ELSE-Zweig zu     *)
3814  (* untersuchen. Ersteres ist nur bei der letzten Klausel einer Bedingten Form*)
3815  (* sinnvoll...                                                               *)
3816
3817   begin
3818      constcond:=constant(l) and
3819                 constant(condgo(l,'THEN    ')) and
3820                 constant(condgo(l,'ELSE    '))
3821    end;
3822
```

```
3823
3824  function listmark
3825    (l: exptr; vars: varptr; ins,nsf: boolean; lac: longint): varptr; forward;
3826
3827  function condmark
3828    (l: exptr; vars: varptr; ins: boolean; lac: longint): varptr; forward;
3829
3830
3831  function mark(ex: exptr; vars: varptr; ins: boolean; lac: longint): varptr;
3832    var h1    : exptr;                (* Hilfsvariable                          *)
3833        h2    : varptr;               (*      "                                 *)
3834        i     : longint;              (*      "                                 *)
3835        maxstr: string;               (* Hilfsvariable fuer die procedure STR *)
3836    begin
3837      if ex^.sym='IF      '
3838        then mark:=condmark(ex^.next,vars,ins,lac)  (* Markierung  Konditional *)
3839        else
3840      if ex^.sym='**FUNC**'
3841        then begin                          (* Markierung von Funktionsruempfen *)
3842               mark:=mark(ex^.next^.next^.next^.next,vars,ins,lac);
3843               ex:=funcgo(ex^.next^.next^.next^.next); (* Sprung ans Rumpfende *)
3844               if ex^.sym<>'**MAIN**'         (* Hauptprogramm nicht markieren *)
3845                 then begin
3846                        new(vars); vars^.prev:=nil; vars^.niv:=0; (*rlk:='leer'*)
3847                        mark:=mark(ex,vars,ins,lac)   (* Mark. naechster Rumpf *)
3848                      end
3849             end
3850        else
3851      if ex^.next^.sym='(       '                                  (* Aufruf *)
3852        then
3853          if (ex^.sym='CAR     ') or (ex^.sym='CDR     ') or
3854             (ex^.sym='ATOM    ')
3855            then mark:=listmark(ex^.next^.next,vars,ins,false,lac)
3856            else
3857          if (ex^.sym='CONS    ') or (ex^.sym='EQ      ') then
3858            begin
3859              if perm and ins and        (* evtl. permutieren von cons oder eq *)
3860                 not(constant(ex^.next^.next)) then      (* PK3 ueberpruefen   *)
3861                begin
3862                  i:=max(mark(ex^.next^.next,vars,false,lac));
3863                  if (i<max(mark(nextparago(ex^.next^.next),vars,false,lac)))
3864                     and (i=max(vars))                    (* PK1 und PK2 pruefen*)
3865                    then permutation(ex)                  (* Parameter tauschen *)
3866                end;
3867              mark:=mark(ex^.next^.next,
3868                         listmark(nextparago(ex^.next^.next^.next),
3869                                  vars,ins,false,lac),
3870                         ins,lac+1)
3871            end else
3872
3873          begin         (* NSF-Aufruf: Bearbeiten der aktuellen Parameterliste *)
3874            new(h2); h2^.prev:=nil; h2^.sym:=ex^.sym; h2^.niv:=ex^.niv;
3875            mark:=union(h2,listmark(ex^.next^.next,vars,ins,true,lac));
3876            if ins and (lac>0) then           (* keine rechten Aeste markieren *)
3877              begin    (* Einf. der GMNIV-Markierung vor der akt. Param.-Liste *)
3878                h1:=ex; new(ex); str(max(vars),maxstr); ex^.sym:='(      ';
3879                for i:=1 to length(maxstr) do ex^.sym[i+2]:=maxstr[i];
3880                ex^.sym[i+3]:='*'; ex^.sym[i+4]:=')'; ex^.niv:=0;
3881                ex^.next:=h1^.next; h1^.next:=ex
3882              end
3883          end
3884        else
3885      if (ex^.sym<>'"       ') and (ex^.sym<>'(       ') and
3886         (ex^.sym<>'T       ') and (ex^.sym<>'F       ') and
3887         (ex^.sym<>'CAR     ') and (ex^.sym<>'CDR     ') and
3888         (ex^.sym<>'ATOM    ') and (ex^.sym<>'CONS    ') and
3889         (ex^.sym<>'EQ      ') and (not((ex^.sym[1]='(') and (ex^.sym[2]='*')))
3890        then begin
3891               (* Einfuegen der Markierung nach einem NS-Idf, der nicht        *)
3892               (* der Funktionsidf. eines form. o. gew. Funktionsaufrufes ist. *)
3893
3894               new(h2); h2^.prev:=nil; h2^.sym:=ex^.sym; h2^.niv:=ex^.niv;
3895               mark:=union(h2,vars);                  (* rlk um Idf. erweitern *)
3896
3897               if ins and (lac>0) then        (* keine rechten Aeste markieren *)
```

```
3898                begin                                  (* Einfuegen der Marke   *)
3899                  h1:=ex; new(ex); str(max(vars),maxstr); ex^.sym:='(*      ';
3900                  for i:=1 to length(maxstr) do ex^.sym[i+2]:=maxstr[i];
3901                  ex^.sym[i+3]:='*'; ex^.sym[i+4]:=')'; ex^.niv:=0;
3902                  ex^.next:=h1^.next; h1^.next:=ex
3903                end
3904            end
3905        else mark:=vars        (* Atome, Listen, T und F werden nicht markiert *)
3906    end; (* of mark *)
3907
3908
3909  function listmark
3910    (l: exptr; vars: varptr; ins,nsf: boolean; lac: longint): varptr;
3911
3912  (* Markiert die aktuelle und ggf. pending Parameterliste(n) und fuegt vor    *)
3913  (* jedem dicken Parameter die Markierung DPNIV ein.                          *)
3914
3915    var h1    : exptr;                (* Hilfsvariable                        *)
3916        h2,h3 : varptr;               (*      "                               *)
3917        i     : longint;              (*      "                               *)
3918        maxstr: string;               (* Hilfsvariable fuer die procedure STR *)
3919        dckpar: boolean;              (* liegt ein dicker Parameter vor?      *)
3920    begin
3921      if l^.sym=')       '                          (* Listenende erreicht    *)
3922        then listmark:=vars
3923        else
3924      begin
3925        if listendgo(l)^.next^.sym='(       '        (* PP-Liste(n) markieren  *)
3926          then vars:=union(listmark(listendgo(l)^.next^.next,vars,ins,nsf,lac),
3927                           vars);
3928        dckpar:=(l^.sym='IF      ') or (l^.next^.sym='(       ');
3929        if ins and nsf and dckpar
3930          then begin                          (* Dicker Parameter: max.stat.Niv.*)
3931                 new(h2); h2^.prev:=nil; h2^.niv:=0; h2:=mark(l,h2,false,lac);
3932               end;
3933        if nsf                                (* Aktueller Parameter NSF-Aufruf *)
3934          then begin                          (* LK ist nur aktueller Parameter *)
3935                 new(h3); h3^.prev:=nil; h3^.niv:=0;
3936                 listmark:=union(listmark(nextparago(l^.next),vars,ins,nsf,lac),
3937                                 mark(l,h3,ins,lac))
3938               end
3939          else listmark:=mark(l,listmark(nextparago(l^.next),vars,ins,nsf,lac),
3940                              ins,lac);
3941        if ins and nsf and dckpar
3942          then begin               (* Einfuegen der Marke vor Dickem Parameter *)
3943                 i:=100+max(h2);
3944                 new(h1); h1^.sym:=l^.sym; h1^.niv:=l^.niv; h1^.next:=l^.next;
3945                 str(i,maxstr); l^.sym:='(*      ';
3946                 for i:=1 to length(maxstr) do l^.sym[i+2]:=maxstr[i];
3947                 l^.sym[i+3]:='*'; l^.sym[i+4]:=')'; l^.niv:=0; l^.next:=h1
3948               end
3949      end
3950    end; (* of listmark *)
3951
3952
3953  function condmark(l: exptr; vars: varptr; ins: boolean; lac: longint): varptr;
3954  (* Markierung eines Konditionals, d.h. einer IF-THEN-ELSE-FI Struktur        *)
3955
3956    var h1,h2: varptr;                                (* Hilfsvariable           *)
3957    begin
3958      h1:=mark(condgo(l,'ELSE    '),vars,ins,lac);  (* Else-Teil markieren     *)
3959      h2:=mark(condgo(l,'THEN    '),vars,ins,lac);  (* Then-Teil markieren     *)
3960      condmark:=mark(l,union(h1,h2),ins,lac+1)      (* If-Teil   markieren     *)
3961    end;
3962
3963
3964  begin (* of procedure gpmark *)
3965    if isnextsym<>'**EMPTY*'
3966      then begin       (* Einlesen und Markieren des Programmes mit Funktionen *)
3967             new(exinp); exinp^.next:=nil; exroot:=exinp;
3968             einlesen;
3969             new(vars); vars^.prev:=nil; vars^.niv:=0;
3970             vars:=mark(exroot,vars,true,0);
3971             ausgabe
3972           end
```

```
3973       else begin       (* Hauptprogramm ist leer. ZWCODE auf GMZWCODE kopieren *)
3974              reset(zwcode); rewrite(gmzwcode);
3975              while not(eof(zwcode)) do
3976                begin readln(zwcode,zeile); writeln(gmzwcode,zeile) end;
3977              close(gmzwcode)
3978            end
3979   end;  (* of procedure gpmark *)
3980
3981
3982   begin (* of procedure gmarkfdecp2 *)
3983     writeln('  *****                                                ******');
3984     repeat
3985       write('             CONS/EQ ggf. permutieren (J/N)? ');
3986       readln(key); if key='j' then key:='J'; if key='n' then key:='N'
3987     until key in ['J','N'];
3988     if key='J' then perm:=true
3989                else perm:=false;
3990     writeln('  *****                                                ******');
3991     if perm
3992      then writeln('  *****      GPMARK STARTED                          ******')
3993      else writeln('  *****      GMARK STARTED                           ******');
3994     assign(gmzwcode,'gmzwcode.dat');
3995     init2; gpmark;       (* Vorbesetzungen und Start der Markierung vom ZWCODE *)
3996     WRITELN('  *****      SECOND PART STARTED                     ******   ');
3997     gminit2; funcdeclpart2    (* Vorbesetzungen und Uebersetzung des GMZWCODES *)
3998
3999   end;  (* of procedure gmarkfdecp2 *)
4000
4001
4002
4003   (* ************************************************************************ *)
4004   (* ************     H A U P T P R O G R A M M        ********************** *)
4005   (* ************                                      ********************** *)
4006   (* ******** ANSTOSS DER VORBESTZUNGEN/ DIE <PROGRAM> - PRODUKTION   ******** *)
4007   (* ******** START DES ERSTEN UND DES 2. LAUFES                     ******** *)
4008   (* ******** ANLEGEN DES ZIELCODE (ZUSAMMENKOPIEREN VON PASFIL1,PAS- ******** *)
4009   (* ******** FIL2 UND STANDC )                                     ******** *)
4010   (* ************************************************************************ *)
4011
4012
4013
4014
4015   BEGIN   (* DES HAUPTPROGRAMMS VON ANACOMP                                  *)
4016
4017    WRITELN('  ');                                  (* MELDUNGEN AN DEN BENUTZER *)
4018    WRITELN('  *****                                        ******   ');
4019    WRITELN('  ***** PROGRAM ANACOMP    (VER. ',VERSION,') ******   ');
4020    WRITELN('  *****                                        ******   ');
4021    writeln('  *****      TP-Version ',tpversion,'           ******   ');
4022    writeln('  *****                                        ******   ');
4023    WRITELN('  *****      PROGRAM LISPANACOMP STARTET       ******   ');
4024    INIT;                                       (* VORBESETZUNGEN DES 1. LAUF*)
4025
4026    (*  <PROGRAM>::=BEGIN <MODE DECL. PART><FUNC.DECL.PART><MAIN PROGRAM> END   *)
4027
4028    IF ISSYM<>'BEGIN   ' THEN FEHLER(1);
4029    MODEDECLPART;
4030    FUNCDECLPART;
4031    MAINPROGRAM;
4032    IF ISNEXTSYM<>'END     ' THEN FEHLER(26);
4033    WRITELN(PASFIL2,'GOTO 2;');                 (* MARKE FUER ??????????????*)
4034                                                (* MELDUNGEN FUER BENUTZER   *)
4035    WRITELN('  *****      NO ERRORS IN FIRST PART           ******   ');
4036
4037    gmarkfdecp2;                    (* Aufruf von funcdeclpart2 und ggf. GPMARK *)
4038
4039                                                (*  ABSCHLUSSHANDLUNGEN      *)
4040    (* WHILE NOT(EOF(ZWCODE)) DO NEXTCODESYM(ISSYM,ISNEXTSYM); *)
4041                                                (* MELDUNGEN AN BENUTZER     *)
4042    WRITELN('  *****      NO ERRORS IN PROGRAM              ******   ');
4043    WRITELN('  *****                                        ******   ');
4044    WRITELN('  ');
4045    WRITELN(LISTFILE,' ');
4046    IF SCHACHT>1 THEN FEHLER(25);               (* ] FEHLEN                  *)
4047
```

```
4048
4049
4050  (*************************************************************************)
4051  (***************** ANLEGEN DES ZIELPROGRAMMES   ****************************)
4052  (*************************************************************************)
4053
4054
4055
4056  REWRITE(ZPROG);
4057  WRITELN(ZPROG,'PROGRAM STANDLZSR(INPUT,OUTPUT,kell);');
4058  writeln(zprog,'uses dos;');             (* Fuer Laufzeitbestimmung (TP)      *)
4059  WRITELN(ZPROG,'LABEL ');
4060  FOR I:=0 TO LABELNR-2 DO WRITELN(ZPROG,I:4,',');
4061  WRITELN(ZPROG,LABELNR-1:4,';');
4062  RESET(STANDC);
4063  WHILE NOT(EOF(STANDC)) DO               (* LAUFZEITSYST./STANDARTTEXT        *)
4064    begin                                 (* Kopieren des LZS zum Zielprogramm *)
4065      readln(standc,zeile); writeln(zprog,zeile)
4066    end;
4067  WRITELN(ZPROG,' ');
4068  FOR I:=3 TO LABELNR-2 DO WRITELN(ZPROG,'    ',I:3,': GOTO ',I:3,';');
4069  WRITELN(ZPROG,'    ',LABELNR-1:3,': GOTO ',LABELNR-1:3);
4070  WRITELN(ZPROG,'   END;    (* OF CASE  *) ');
4071  writeln(zprog);
4072  WRITELN(ZPROG,'1:');
4073  WRITELN(ZPROG,' ');
4074  RESET(PASFIL1);
4075  WHILE NOT(EOF(PASFIL1)) DO                    (* KONSTANTEN                *)
4076                          BEGIN
4077                           readln(pasfil1,zeile); writeln(zprog,zeile)
4078                          END;
4079  RESET(PASFIL2);
4080  WHILE NOT(EOF(PASFIL2)) DO                    (* ERZEUGTER CODE OHNE KONST *)
4081                           BEGIN
4082                            readln(pasfil2,zeile); writeln(zprog,zeile)
4083                           END;
4084
4085  writeln(zprog); WRITELN(ZPROG,'2: statistik');
4086  writeln(zprog); writeln(zprog,'END.');
4087
4088  8888: ende
4089
4090  END.
```

Diese Arbeit habe ich selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel benutzt.

Münster, den 7. August 1992