Ulrich Peters

Definition und Implementierung der Programmiersprache PAL

Ausarbeitung einer Vorbereitung von Fortgeschrittenen-Praktika

A 74 / 14

Inhaltsverzeichnis

Einleitung

Die vorliegende Arbeit definiert in ihrem ersten Teil die von mir erdachte Programmiersprache PAL; der zweite Teil ist eine (vielleicht nicht völlig erschöpfende) Ausarbeitung einer Praktikumsvorbereitung, die ich im Auftrag von Herrn Prof. Dr. H. Langmaack im Wintersemester 73/74 am Lehrstuhl für Informatik I der Universität des Saarlandes zur Erstellung eines Compilers für PAL mittels einer Reihe von Fortgeschrittenen-Praktika für Studenten in Form einer vierstündigen Vorlesung durchgeführt habe. Hauptaufgabe dieser Vorbereitung war es, die Studenten an die Masse der tatsächlich durchzuführenden Arbeiten und Überlegungen heranzuführen, ihnen konkrete und fest und eindeutig umrissene Aufgaben zuzuweisen und sie durch vorzugebende erschöpfende Schnittstellen möglichst unabhängig zu machen (dazu wäre der Compiler am besten natürlich schon vorher fix und fertig vorhanden gewesen). Allgemeingültige Compilertechniken wurden als bekannt vorausgesetzt und nur soweit motiviert und wiederholt oder (meist) spezialisiert, wie es zum Verständnis notwendig erschien; sie tauchen deshalb hier auch nur in diesem Umfang auf. Auf diese Weise befreit die vorliegende Arbeit denjenigen, der ähnliche Fortgeschrittenen-Praktika vergeben will, von der ganzen "Schmutzarbeit", die bei der Erstellung eines Compilers unweigerlich anfällt und befähigt ihn, sich vor allem der Illustration und Motivation von Compilertechniken (der angedeuteten oder auch von solchen seiner persönlichen Wahl) zu widmen. Für den Studenten andererseits ist diese Arbeit eine weitgehend erschöpfende Zusammenstellung all derjenigen Details, die (von der Definition der Sprache abgesehen) theoretisch unwichtig sind, in praxi aber doch beachtet werden müssen. In dieser Arbeit ausgesprochene Empfehlungen geben Änderungen (meist Erweiterungen) der augenblicklichen Implementierung an, die aus den jeweils genannten Gründen wünschenswert erscheinen; der Praktikumsleiter sollte festlegen, welche Empfehlungen realisiert werden sollen und welche nicht.

Über die Entscheidung, den Scanner ein Protokoll der PAL-Quelle erstellen zu lassen, dessen Gestaltung von der der Quelle unabhängig ist und nach einem Satz von wünschenswerten Gesichtspunkten erfolgte, kann man vielleicht streiten; diese Entscheidung ist fraglos beeinflußt von meinen vielen ärgerlichen Erfahrungen mit unordentlich geschriebenen Programmen, paßt jedoch zu der Tatsache, daß das ganze Projekt ein Studien- und kein kommerzielles Projekt ist, sodaß logische und logisch wünschenswerte Gesichtspunkte vor Effizienzerwägungen großen Vorrang haben dürfen. Im übrigen halte ich den Einwand, daß man so im Fehlerfall das Auffinden des Fehlerorts in der Quelle unzumutbar erschwert, für eine fadenscheinige Ausrede. Im Sinn des Studienprojekts liegt auch die Maschinenunabhängigkeit des Compilers sowie die (neben der Protokollierung) weitere Extraleistung, dem PAL-Programmierer eine besonders reichhaltige Palette von Darstellungsmöglichkeiten anzubieten, sodaß er den Zeichenvorrat des Eingabegeräts seiner Wahl uneingeschränkt und sehr arbeitssparend nutzen kann; der Student erhält so nebenbei einen Einblick, daß Extraleistungen längst nicht immer so "teuer" sind wie es gern behauptet wird.

Es sei erwähnt, daß die Nichtexistenz oder doch Nichteinsatzfähigkeit von compilergenerierenden Hilfsmitteln zu "handgeschneiderten" Lösungen der Probleme zwang, ein Umstand, der im Rahmen eines Studienprojekts durchaus auch seine Vorteile hat.

Abschließend möchte ich es nicht versäumen, mich bei Herrn Prof. Dr. H. Langmaack für die zahlreichen Anregungen und die Aufforderung zur Abfassung dieser Arbeit und bei den Herren Prof. Dr. G. Seegmüller und Dr. C. Zenger für die tatkräftige Unterstützung bei der Abfassung dieser Arbeit zu bedanken.

München, den 16. März 1975 Ulrich Peters

A) Definition der Sprache PAL

I. Überblick

Die Sprache PAL (Praktikums-ALGOL) ist ein Dialekt von ALGOL 60; mit ihr sollen drei Zielsetzungen erreicht werden, von denen die erste die wichtigste ist:

1) Die Sprache soll einfach und nicht zu umfangreich sein, um es zu ermöglichen, mit Informatik-Studenten (mit Vordiplom) im Rahmen eines einsemestrigen Fortgeschrittenen-Praktikums einen Übersetzer für diese Sprache zu erstellen, andererseits aber doch so kompliziert, daß die Studenten die wesentlichen übersetzertypischen Strukturen und Probleme (Zwischensprachen, Informationstabellen, lexikale, syntaktische und semantische Analyse, Codegenerierung) kennen und lösen lernen.

2) Die Sprache soll es gestatten, in einfacher Weise alle Sprachstrukturen von ALGOL 60 auszudrücken oder doch zu simulieren; lediglich Prozeduren, die rekursiv aufgerufen werden oder nichtspezifizierte formale Parameter haben, bleiben außer Acht (dies geschieht vor allem wegen Zielsetzung 1).

3) Wie ALGOL 60 soll PAL im wesentlichen auf numerische Bedürfnisse ausgerichtet sein. Als Besonderheit sollen PAL-Programme Rechnungen mit (prinzipiell) beliebig langen und damit beliebig genauen Zahlen direkt ausführen können.

Um 1) zu erfüllen, verzichten wir auf jegliche Art von Prozeduren, ferner auf bedingte Anweisungen, Laufanweisungen, zusammengesetzte Anweisungen, Verteiler, reelle Zahlen und Größen und alle Anpassungen außer impliziten Dereferenzierungen in gewissen Fällen. Zur weiteren Vereinfachung besitzen alle Größen eine Lebensdauer, die sich vom Start des Programms bis zu seiner Terminierung erstreckt; der Gültigkeitsbereich einer jeden Größe ist das gesamte Programm; der Gültigkeitsbereich einer Bezeichnung einer Größe dagegen rich-

tet sich nach der Blockstruktur des Programms mit Ausnahme von Standardbezeichnungen, welche (mit einer Ausnahme) im gesamten Programm gelten. Die Blockstruktur hat damit nur noch die allerdings wichtige Funktion, Nomenklatur-Ebenen festlegen zu können.

Um 2) zu erfüllen, führen wir Variable ein, deren Werte Marken sind, ferner Zeiger, deren Werte Bezüge auf Variable (einfache wie auch indizierte) oder Felder (das sind Variablenmengen) sind, schließlich noch Deklarationen, die im Sinn von ALGOL 68 Identitätsdeklarationen sind, aber ausnahmslos zur Übersetzungszeit ausführbar sind. Außerdem kann man zur Laufzeit des Programms Felder automatisch (prinzipiell beliebig) vergrößern lassen wie auch die jeweils aktuellen Feldgrenzen abfragen.

Um 3) zu erfüllen, ermöglichen wir direkte Rechnungen mit ganzen Zahlen von (prinzipiell) beliebiger Länge und ohne Beschränkung auf eine maximale Stellenzahl (Datentyp long). Natürlich dauern arithmetische Operationen umso länger, je länger die Werte der zu verknüpfenden Operanden sind; nicht zuletzt deshalb wurde die Länge von arithmetischen Werten (in Dezimalstellen) direkt abfragbar gemacht.

II. Definition von PAL im einzelnen

1. Zeichenvorrat, Identifikatoren, Konstante, Grundkonzepte

1.1 PAL-Zeichen

<sign> ::= <escape sign> | <basic sign> | <functional sign> |
<special sign>

Der Vorrat an PAL-Zeichen gliedert sich in vier Gruppen: Fluchtzeichen (escape signs), Grundzeichen (basic signs), Funktionszeichen (functional signs) und Spezialzeichen (special signs). Jedem PAL-Zeichen ist ein Wert (sog. PAL-Wert) zugeordnet, den man, bei 0 beginnend, durch Abzählen der PAL-Zeichen findet. Die PAL-Werte der PAL-Zeichen sind (aus der Sicht des PAL-Programmierers) dazu da, Ersatzdarstellungen für die Funktions- und Spezialzeichen zur Verfügung zu haben, wenn diese selbst nicht darstellbar sind oder in ihrer unmittelbaren Bedeutung gemeint sind; für die Flucht- und Grundzeichen gibt es keine Ersatzdarstellungen. Ersatzdarstellungen dürfen nur sehr eingeschränkt verwendet werden, nämlich nur innerhalb von Kommentaren (siehe 1.10), Strings (siehe 1.9) und Spezialzeichendefinitionen (siehe 4); in eben diesen Fällen ist inre Benützung unter Umständen sogar notwendig. Eine Ersatzdarstellung ist eine Sequenz von vier Zeichen, die allenfalls von Zeilenwechseln unterbrochen sein darf. Das erste Zeichen ist das Fluchtzeichen F und die restlichen Zeichen sind Ziffern, die, als Zahl aufgefaßt, gerade den PAL-Wert des Funktions- oder Spezialzeichens angeben. Zur Darstellung von F selbst ist im Rahmen einer Spezialzeichendefinition ein Spezialzeichen heranzuziehen (näheres siehe 4). Ist z. B. $ das F darstellende Spezialzeichen, so ist $120 gerade die Ersatzdarstellung von $ selbst, d. h. ohne seine Fluchtzeichenfunktion.

1.1.1 Fluchtzeichen

<escape sign> ::= Ø | 1 | 2 | Ü | Ö | F | S | K | K | _

Die Fluchtzeichen besitzen die PAL-Werte 0 bis 9 und keine Ersatzdarstellungen. Sie sind im wesentlichen nur innerhalb des PAL-Compilers von Bedeutung. Der PAL-Programmierer kann nur drei von ihnen, nämlich F, S und K, darstellen und dies auch nur indirekt, nämlich durch Spezialzeichen (näheres siehe 4); im übrigen ist für ihn die Bedeutung der folgenden fünf Fluchtzeichen wichtig:

Ø : Erscheint im Protokoll des PAL-Programms dieses Zeichen, so weist das PAL-Programm an der betreffenden Stelle ein PAL-Zeichen auf, welches auf dem entsprechenden Drucker nicht durch Kombination zweier direkt druckbarer Zeichen einigermaßen leserlich darstellbar ist (jedenfalls nicht nach dem Geschmack des Implementierers des PAL-Compilers).

Ø : Erscheint im Protokoll des PAL-Programms dieses Zeichen, so wurde beim Einlesen des PAL-Programms (zum Zweck seiner Übersetzung) an der betreffenden Stelle ein Zentralcodewert des Computers gefunden, dem gar kein PAL-Zeichen zugeordnet ist.

F : Dies Zeichen ist das Fluchtzeichen für Wortsymbole (siehe 1.4, 1.6 und 1.8), Protokollvorschübe (siehe 1.10) und Ersatzdarstellungen; es ist im PAL-Programm nur indirekt durch ein (frei wählbares) Spezialzeichen darstellbar.

S : Dieses Zeichen markiert den Anfang wie auch das Ende eines Strings, vgl. 1.9; es ist (wie F) nur durch Spezialzeichen darstellbar.

K : Dieses Zeichen markiert den Anfang wie auch das Ende eines Kommentars (einer Kommentarzeile), vgl. 1.10; es ist (wie F) nur durch Spezialzeichen darstellbar.

1.1.2 Grundzeichen

<basic sign> ::= ␣ | . | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
A | B | C | D | E | F | G | H | I | J | K | L |

M | N | O | P | Q | R | S | T | U | V | W | X |
Y | Z | Ä | Ö | Ü | ß | (|)

Die Grundzeichen besitzen die PAL-Werte 10 bis 53 und sind nur durch sich selbst darstellbar. Sie umfassen den Zwischenraum, den Punkt, die Ziffern, die großen Buchstaben (einschließlich der Umlaute und dem Zeichen ß, welches als Darstellung des großen scharfen S gedacht ist) und die runden Klammern.

1.1.3 Funktionszeichen

<functional sign> ::= + | - | * | / | ↑0 | < | > | ≤ | ≥ | = |
≠ | ¬ | ∧ | ∨ | < | > | ≡ | & | : | ; | , |
₡ | ₦ | ₱

Die Funktionszeichen besitzen die PAL-Werte 54 bis 77 und Ersatzdarstellungen der erwähnten Art. Die meisten von ihnen sind binäre Operatoren. Sie sind außerdem überall im PAL-Programm, außer innerhalb von Spezialzeichendefinitionen, durch Spezialzeichen darstellbar; entsprechende Zuordnungen sind dann in Spezialzeichendefinitionen festzulegen (siehe 4).

Die letzten drei Funktionszeichen sind zur Verwendung innerhalb von Strings (siehe 1.9) gedacht. Wird ein derartiger String zur Laufzeit des übersetzten PAL-Programms ausgegeben, so bewirkt die Ausgabe von ₡ bzw. ₦ bzw. ₱ einen (drucktechnischen) Rücksprung auf den Anfang der jüngst begonnenen Zeile (carriage return) bzw. einen Sprung auf den Anfang der nächsten Zeile bzw. nächsten Seite (new line bzw. new page).

1.1.4 Spezialzeichen

<special sign> ::= a | b | c | d | e | f | g | h | i | j | k | l |
m | n | o | p | q | r | s | t | u | v | w | x |
y | z | ä | ö | ü | ß | % | ? | ! | [|] | { |
} | | | \ | ↑ | ↓ | → | $ | ¢ | # | @ | ' | " |
´ | ` | ^ | (|) | • | ~ | _ | ¯ | × | ✠ | § |
¤ | π

Die aufgeführten 62 Spezialzeichen besitzen die PAL-Werte 78 bis 139 und Ersatzdarstellungen der erwähnten Art. Sie sind gerade diejenigen Zeichen, die im Zeichenvorrat von 5-Kanal-Fernschreibern oder in den Zentralcodes des CDC3300 oder des TR440 auftreten und, wenn durch Spezialzeichendefinitionen nicht ausdrücklich etwas anderes festgelegt wird, in PAL-Programmen keine Funktion haben. Der PAL-Compiler ist so konstruiert, daß man ohne Schwierigkeiten noch bis zu 860 weitere Spezialzeichen dazunehmen kann, die dann die PAL-Werte 140 bis 999 und entsprechende Ersatzdarstellungen hätten. Wie Funktionszeichen sind auch Spezialzeichen durch Spezialzeichen darstellbar; entsprechende Zuordnungen sind dann in Spezialzeichendefinitionen festzulegen.

1.1.5 PAL-Werte

Abschließend sei nochmal die Zuordnung PAL-Zeichen / PAL-Werte angegeben, so weit sie den PAL-Programmierer wegen der möglichen Bildung von Ersatzdarstellungen interessiert:

	0	1	2	3	4	5	6	7	8	9
50					+	-	*	/	10	<
60	>	≤	≥	=	≠	¬	∧	∨	<	>
70	≡	&	:	;	,	⊄	⊅	⊮	a	b
80	c	d	e	f	g	h	i	j	k	l
90	m	n	o	p	q	r	s	t	u	v
100	w	x	y	z	ä	ö	ü	ß	%	?
110	!	[	]	{	}	|	\	↑	↓	→
120	$	¢	#	@	'	"	´	`	^	(
130	)	•	~	_	¯	×	✠	§	¤	π

1.2 Buchstaben

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

Ä | Ö | Ü | ß

1.3 Ziffern

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1.4 Begrenzer (Delimiter)

<delimiter> ::= <operator> | <separator> | <bracket> |
<declarator>

<operator> ::= <arithmetic operator> | <bound operator> |
<relational operator> | <logical operator> |
<string operator> | <sequential operator> |
<i/o operator>

<arithmetic operator> ::= + | - | * | / | div | mod | ↑ | abs |
sgp | sgn | sign | dig

<bound operator> ::= lwb | upb

<relational operator> ::= < | ≤ | = | ≥ | > | ≠

<logical operator> ::= ¬ | ∧ | ∨ | < | > | ≡

<string operator> ::= &

<sequential operator> ::= goto | if | then | else | fi

<i/o operator> ::= inp | out

<separator> ::= , | ; | .= | :=

<bracket> ::= (|) | begin | end | bop | eop

<declarator> ::= ident | loc | ref | dim | bool | int | long |
label | chain

Die Bedeutung der einzelnen Begrenzer wird in den Abschnitten 2, 3 und 4 erklärt und ihre Darstellungen in Abschnitt 5.

1.5 Identifikatoren

<identifier> ::= <letter> | <identifier> <letter> | <identifier> . <letter> | <identifier> <digit> | <identifier> . <digit>

In Identifikatoren kann man zur besseren Lesbarkeit und aus Dokumentationsgründen Einzelpunkte einstreuen; diese haben identifizierende Bedeutung, d. h. daß z. B. die Identifikatoren AB und A.B wie auch A.BC und AB.C voneinander verschieden sind. Jede frei wählbare Bezeichnung für eine Größe muß ein Identifikator sein und jeder Identifikator ist eine frei wählbare Bezeichnung für eine Größe. Außer frei wählbaren Bezeichnungen gibt es an Bezeichnungen für Größen nur noch Standardbezeichnungen und diese nur für Konstante. Eine Kapazitätsbeschränkung für Identifikatoren gibt es nicht (jedenfalls nicht prinzipiell); alle Zeichen haben identifizierende Bedeutung.

1.6 Logische Werte

<logical value> ::= true | false

Damit sind die Standardbezeichnungen für die beiden Konstanten "wahr" und "falsch" vom Typ bool eingeführt.

1.7 Zahlen

<integer> ::= <digit> | <integer> <digit> | <integer> . <digit>

In ganzen Zahlen kann man zur besseren Lesbarkeit Einzelpunkte einstreuen; diese haben aber keine identifizierende Bedeutung und auch keine Dezimalpunktfunktion. Damit sind die Standardbezeichnungen für die Konstanten vom Typ int eingeführt. Für ganze Zahlen existiert eine implementierungsabhängige Kapazitätsbeschränkung (in unserer Implementierung: $0 \leq |X| \leq 2\uparrow 22 - 1 = 4\ 194\ 303$).

<long number> ::= . <integer>

<number> ::= <integer> | <long number>

In überlangen (ganzen) Zahlen kann man zur besseren Lesbarkeit Einzelpunkte einstreuen; diese haben aber keine identifizierende Bedeutung und auch keine Dezimalpunktfunktion. Damit sind die Standardbezeichnungen für die Konstanten vom Typ long eingeführt. Eine Kapazitätsbeschränkung für long-Zahlen gibt es nicht (jedenfalls nicht prinzipiell).

1.8 Marken

<proper label> ::= . <identifier>

<label> ::= <proper label> | here

In eigentlichen Marken (proper labels) kann man zur besseren Lesbarkeit und aus Dokumentationsgründen Einzelpunkte einstreuen; diese haben identifizierende Bedeutung, d. h. daß z. B. die Marken .AB und .A.B wie auch .A.BC und .AB.C voneinander verschieden sind. Damit sind die Standardbezeichnungen für die Konstanten vom Typ label eingeführt. Von here abgesehen haben sie die besondere Eigenschaft, in einem Programm genau einmal definiert werden zu müssen, vgl. 1.11. Eine Kapazitätsbeschränkung für Marken gibt es nicht (jedenfalls nicht prinzipiell).

1.9 Strings

<string item> ::= <basic sign> | <functional sign> |
<special sign>

<proper string> ::= <string item> | <proper string> <string item>

<string> ::= $ <proper string> $ | $$

Zur Darstellung von $ ist ein Spezialzeichen heranzuziehen, siehe 4. Soll das F oder $ darstellende Spezialzeichen selbst im String auftreten, so ist die entsprechende Ersatzdarstellung anzugeben; das K darstellende Spezialzeichen (siehe 1.10 und 4) braucht nicht

als Ersatzdarstellung, sondern kann mit gleichem Effekt auch direkt angegeben werden. Es bleibt noch zu erwähnen, daß man einen String auf mehrere Quellprogrammzeilen aufteilen kann: Eine im Zustand "String" auftretende zwischenraumlose und zeilenwechsellose Folge $$, auf die nur Zwischenräume und/oder Zeilenwechsel und dann ein $ folgen, wird einschließlich des letztgenannten $ ignoriert. Nach eben dieser Konvention wird ein "überlanger" String auch im Protokoll über mehrere Zeilen gebrochen. Damit sind die Standardbezeichnungen für die Konstanten vom Typ chain eingeführt. Eine Kapazitätsbeschränkung für Strings gibt es nicht (jedenfalls nicht prinzipiell).

1.10 Protokoll, Kommentare und Protokollvorschübe

Dieser Abschnitt behandelt Einfügungen in PAL-Programmen, die in der Syntax von PAL nicht erwähnt sind, da sie für die Semantik von PAL-Programmen keine Bedeutung haben. Sie sind jedoch wichtig für den menschlichen Leser und damit auch immerhin für den Scanner des PAL-Compilers, zu dessen Aufgaben die Erstellung eines ordentlichen und übersichtlich eingerückten Protokolls des Quellprogramms (natürlich gemäß den Wünschen des PAL-Programmierers) gehört; nach welchem Konzept das Quellprogramm selbst zunächst gestaltet wurde, ist dabei (bis auf eine einzige Ausnahme) ohne jeden Einfluß (die in 5 festgelegten oder durch Verweise angedeuteten Darstellungskonventionen sind natürlich in jedem Fall einzuhalten).

Für das folgende ist es zweckmäßig, "elementare Einheiten" zu definieren. Eine "elementare Einheit" ist entweder ein Delimiter (siehe 1.4), ein Identifikator (siehe 1.5), eine Standardbezeichnung (siehe 1.6, 1.7, 1.8 und 1.9), ein Kommentar (siehe 1.10), ein Protokollvorschub (siehe 1.10) oder eine Spezialzeichendefinition (siehe 4). Es gilt, daß jedes korrekte PAL-Programm eine Sequenz von elementaren Einheiten ist. Mit diesem Begriff lassen sich nun die Konventionen für Kommentare und Protokollvorschübe leicht formulieren:

Ein Kommentar ist eine Zeichenkette der Form

K <Zeichenkette aus PAL-Zeichen * Fluchtzeichen> K

und darf überall zwischen zwei elementaren Einheiten eingefügt werden; vor oder hinter dem Programm (siehe 4) ist Kommentar unzulässig. Er wird vom Compiler ersatzlos überlesen, jedoch im Protokoll des Programms reproduziert. Zur Darstellung von K ist ein Spezialzeichen heranzuziehen, siehe 4. Soll das F oder K darstellende Spezialzeichen selbst im Kommentar auftreten, so ist die entsprechende Ersatzdarstellung anzugeben; das S darstellende Spezialzeichen (siehe 1.9 und 4) braucht nicht als Ersatzdarstellung, sondern kann mit gleichem Effekt auch direkt angegeben werden.

Beispiel: Werde F durch $, S durch ' und K durch " dargestellt. Dann wird der Kommentar

"Dies ist ein Kommentar: $120'$125 "

im Protokoll reproduziert als

KDies ist ein Kommentar: $'" K

Es gibt zwei Sorten von Kommentaren, die sich jedoch nur in der Art und Weise ihrer Reproduktion im Protokoll unterscheiden: "Handlungskommentar" und "zeilenweise strukturierter Kommentar", kurz: "Zeilenkommentar". Handlungskommentar ist Bestandteil oder Anhängsel einer Handlung (siehe 3) oder Unit (siehe 4); um als solcher erkannt werden zu können, muß ihm in derselben Quellprogrammzeile mindestens ein sichtbares PAL-Zeichen irgendeiner elementaren Einheit (eben dieser Handlung oder Unit) vorangehen und die letzte vorangehende elementare Einheit, die kein Protokollvorschub und keine Spezialzeichendefinition ist, darf kein Kommentar sein; umgekehrt wird genau dann auf "Handlungskommentar" erkannt (hier genau liegt die eine Ausnahme, bei der die Gestaltung des Quellprogramms von Bedeutung ist). Ein Handlungskommentar wird direkt in die laufende Protokollzeile eingefügt, wobei man jedoch den Kommentar durch zusätzliche Zwischenräume vor und nach ihm absetzen lassen kann. In allen sonstigen Fällen liegt die erste oder i. a. nächste Zeile eines Zeilenkommentars vor; der Kommentarzeile ging dann in derselben Quellprogrammzeile kein sichtbares Zeichen voran oder die letzte vorangehende elementare Ein-

heit, die kein Protokollvorschub und keine Spezialzeichendefinition ist, ist ein Kommentar. Ein solcher Kommentar (Kommentarzeile) wird im Protokoll stets in eine neue Zeile gesetzt, wobei man ihn relativ zur momentanen Einrückung von Handlungen + Markendefinitionen noch zusätzlich einrücken lassen kann; paßt die Kommentarzeile nicht in die Protokollzeile, so wird die Kommentarzeile in der nächsten Protokollzeile fortgesetzt, wobei die überstehenden Kommentarteile noch zusätzlich eingerückt werden lassen können.

Es bleibt noch zu erwähnen, daß man einen Handlungskommentar wie auch jede Zeile eines Zeilenkommentars auf mehrere Quellprogrammzeilen aufteilen kann: Eine im Zustand "Kommentar" auftretende zwischenraumlose und zeilenwechsellose Folge KK, auf die nur Zwischenräume und/oder Zeilenwechsel und dann ein K folgen, wird einschließlich des letztgenannten K ignoriert. Nach eben dieser Konvention wird ein "überlanger" Kommentar (Kommentarzeile) auch im Protokoll über mehrere Zeilen gebrochen.

Protokollvorschübe schließlich sind dazu da, die Protokollgestaltung lokal zu beeinflussen. Ein Protokollvorschub ist eine Zeichenfolge, die allenfalls von Zeilenwechseln unterbrochen sein darf und die mit F eingeleitet und auch abgeschlossen wird. Es gibt genau die folgenden drei Arten von Protokollvorschüben: FPAGExF, FLINExF und FBLANKxF. x bezeichnet dabei eine Folge von Ziffern, die, wenn sie leer ist, durch x = 1 ersetzt wird. Protokollvorschübe werden im Protokoll nicht wie Kommentare reproduziert, sondern sogleich ausgeführt: FPAGExF bzw. FLINExF bzw. FBLANKxF bewirkt die Ausgabe von x Seitenvorschüben bzw. Zeilenvorschüben bzw. Zwischenräumen an eben der Stelle, an der der Protokollvorschub steht. Nur für FBLANKxF gilt, daß keinesfalls mehr Zwischenräume ausgegeben werden als in die laufende Protokollzeile noch hineinpassen.

1.11 Größen, Arten, Typen, Gültigkeitsbereiche und Lebensdauer

Folgende Arten von Größen werden unterschieden: Konstante, Variable, Felder (d. h. Mengen von Variablen) und Zeiger. Konstante besitzen die Referenzstufe 0, Variable und Felder die Referenzstufe 1 und Zeiger die Referenzstufe 2.

Bei Konstanten werden gemäß ihrem Typ die folgenden fünf Sorten unterschieden: Konstante vom Typ bool, int, long, label und chain. Für jede Konstante gibt es (oft mehrere) Standardbezeichnungen, vgl. 1.6 bis 1.9.

Jede Standardbezeichnung * here für Marken hat ihrer Natur nach die besondere Eigenschaft, im Programm genau einmal definiert werden zu müssen; dabei ist der von ihr bezeichnete Wert gerade derjenige Programmort, an dem sie in definiert werdender Weise auftritt (vgl. 3, label definition). Für jedes Exemplar von here gilt die Sonderregelung, daß dieses stets denjenigen Programmort bezeichnet, der unmittelbar hinter derjenigen Handlung (vgl. 3) liegt, in der das Exemplar von here auftritt; here faßt man also zweckmäßigerweise als nur sehr lokal gültige Phantasie-Standardbezeichnung für Programmorte auf. Andererseits ist der Gültigkeitsbereich aller sonstigen Standardbezeichnungen das gesamte PAL-Programm. Dies gilt insbesondere für die (frei wählbaren) Standardbezeichnungen * here von Marken. Daraus folgt, daß man nicht zwei verschiedene Programmorte durch ein- und dieselbe Standardbezeichnung * here bezeichnen kann.

Konstante können (zusätzlich), einfache Variable, Felder und Zeiger müssen mit frei wählbaren Bezeichnungen (das sind genau die Identifikatoren) bezeichnet werden. Identifikatoren werden durch eine Deklaration eingeführt und sind in dem kleinsten, die Deklaration umfassenden Block (vgl. 4) gültig (dies ist genau die Nomenklatur-Funktion von Blöcken). Bezüglich gleichnamig deklarierter Identifikatoren in untergeordneten Blöcken gilt die Konvention von ALGOL 60.

Von dem Gültigkeitsbereich einer Bezeichnung einer Größe ist der Gültigkeitsbereich der Größe selbst wohl zu unterscheiden: Der Gültigkeitsbereich aller Größen ist das gesamte Programm; ferner erstreckt sich die Lebensdauer aller Größen vom Start des Programms bis zu seiner Terminierung.

Zu jeder der fünf Sorten von Konstanten gibt es Variable (einfache und indizierte), die genau diese Sorte von Konstanten als Wertebereich haben; andersartige Variable gibt es nicht. Unter dem Typ einer Variablen verstehen wir den Typ der Elemente des Wertebereichs dieser Variablen. Da alle Feldelemente (indizierte Variable) eines Feldes vom gleichen Typ sind, vereinbaren wir als Typ eines Feldes den Typ seiner Elemente.

Zu jeder der damit vorhandenen fünf Sorten von Variablen und fünf Sorten von Feldern gibt es Zeiger, die Bezüge auf genau diese Sorte von Variablen bzw. Feldern als Wertebereich haben; andersartige Zeiger gibt es nicht. Unter dem Typ eines Zeigers verstehen wir den Typ derjenigen Variablen bzw. Felder, auf die der Zeiger Bezüge enthält. Damit gibt es zehn Sorten von Zeigern.

2. Ausdrücke

<subscript expression> ::= <expression>

<subscript list> ::= <subscript expression> | <subscript list> ,
<subscript expression>

<subscripted identifier> ::= <identifier> (<subscript list>)

Ein Indexausdruck ist ein Ausdruck vom Typ int (die Typen von Ausdrücken werden im folgenden rekursiv erklärt). Ein indizierter Identifikator bezeichnet stets ein Feldelement (eine indizierte Variable).

<designator> ::= <identifier> | <subscripted identifier> |
<constant>

<constant> ::= <logical value> | <number> | <label> | <string>

<adding operator> ::= + | -

<unary operator> ::= ¬ | abs | sgp | sgn | sign | dig

<multiplying operator> ::= * | / | div | mod

<primary> ::= <designator> | <adding operator> <primary> |
<unary operator> <primary> | (<expression>) |
if <expression> then <expression> else
<expression> fi

<factor> ::= <primary> | <factor> ↑ <primary> | <factor>
<bound operator> <primary>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<sum> ::= <term> | <sum> <adding operator> <term>

<relation> ::= <sum> | <relation> <relational operator> <sum>

<boolean primary> ::= <relation>

<boolean factor> ::= <boolean primary> | <boolean factor> ∧
<boolean primary>

<boolean term> ::= <boolean factor> | <boolean term> ∨
<boolean factor>

<implication> ::= <boolean term> | <implication> <
<boolean term> | <implication> >
<boolean term>

<simple boolean> ::= <implication> | <simple boolean> ≡
<implication>

<expression> ::= <simple boolean> | <expression> &
<simple boolean>

Dieser Syntax wird man zunächst entnehmen, daß in ihr zwischen den Ausdrücken vom Typ bool, int, long, label und chain nicht unterschieden wurde (was ohne typabhängige Syntaxanalyse auch gar nicht möglich ist), ferner, daß folgende Prioritäten gelten:

1. + | - | ¬ | abs | sgp | sgn | sign | dig (unäres + und -)
2. ↑ | lwb | upb
3. * | / | div | mod
4. + | - (binäres + und -)
5. < | ≤ | = | ≥ | > | ≠
6. ∧
7. ∨
8. < | >
9. ≡
10. &
11. if ... then ... else ... fi | (...)

Insbesondere sei angemerkt, daß die unären Operatoren höchste Priorität besitzen und daß es z. B. auch folgende logischen Ausdrücke gibt:

1 ≤ I < J + 1 ≤ N + 3 | A = B ≠ C usw.,

die die Bedeutung der Ausdrücke

1 ≤ I ∧ I < J+1 ∧ J+1 ≤ N+3 | A = B ∧ B ≠ C usw.

haben sollen.

Der Typ eines designators, als Ausdruck aufgefaßt, ist gleich dem in 1 eingeführten Typ des designators. Das Gleiche gilt für die Referenzstufe eines designators.

Die unären Operatoren + und - haben die übliche arithmetische Bedeutung. In dem Ausdruck b: + a bzw. b: - a muß der Ausdruck a vom Typ int oder long sein; b besitzt denselben Typ wie a und die Referenzstufe 0.

Der unäre Operator ¬ bezeichnet die logische Negation. In dem Ausdruck b: ¬ a muß der Ausdruck a vom Typ bool sein; b ist vom Typ bool und hat die Referenzstufe 0.

Der unäre Operator abs bezeichnet die Bildung des Absolutbetrags. In dem Ausdruck b: abs a muß der Ausdruck a vom Typ int oder long sein; b besitzt denselben Typ wie a und die Referenzstufe 0.

Die unären Operatoren sgp, sgn und sign bezeichnen Vorzeichenfunktionen. In dem Ausdruck b: sgp a bzw. b: sgn a bzw. b: sign a muß der Ausdruck a vom Typ int oder long sein; b besitzt denselben Typ wie a und die Referenzstufe 0. Im einzelnen gilt:

```
 sgp a := if a < 0 then -1 else +1 fi
 sgn a := if a > 0 then +1 else -1 fi
sign a := if a > 0 then +1 else if a < 0 then -1 else 0 fi fi
```

Der unäre Operator dig bezeichnet die Ermittlung der Anzahl der Dezimalstellen des Absolutbetrags seines Arguments (dig 0 = 1). In dem Ausdruck b: dig a muß der Ausdruck a vom Typ int oder long sein; b ist vom Typ int und besitzt die Referenzstufe 0.

Der binäre Operator $_{10}$ bezeichnet die Multiplikation mit einer Potenz von 10. In dem Ausdruck c: a $_{10}$ b muß der Ausdruck a vom Typ int oder long und der Ausdruck b vom Typ int sein; c besitzt den Typ von a und die Referenzstufe 0. Der Wert von c ist gleich dem Wert von a, multipliziert mit 10↑b. Ist der Wert von b kleiner als 0, so werden die anfallenden Stellen hinter dem Komma weggeworfen und zwar unabhängig vom Vorzeichen von c.

Die binären Operatoren lwb und upb (lower bound und upper bound) liefern die Werte der unteren bzw. oberen Grenzen eines Feldes. In dem Ausdruck c: a lwb b bzw. c: a upb b muß a ein Ausdruck vom Typ int sein und b ein Ausdruck, der ein Feld definiert. c besitzt den Typ int, die Referenzstufe 0 und hat als Wert die a-te untere bzw. obere Grenze des durch b definierten Feldes. Ergibt der Ausdruck a bei einer Auswertung zur Laufzeit einen Wert kleiner als 1 oder größer als die Dimension des durch b definierten Feldes oder sind die Grenzen des Feldes noch undefiniert (vgl. 3, Deklarationen, Fall "loc array"), so terminiert das Programm irregulär. lwb und upb sind (außer den runden Klammern und if ... then ... else ... fi) die einzigen in Ausdrücken auftretenden Operatoren, die unter ihren Operanden ein Feld gestatten (ja sogar verlangen).

Der binäre Operator * bezeichnet die arithmetische Multiplikation. In dem Ausdruck c: a * b müssen die Ausdrücke a und b vom gleichen Typ sein, wobei nur int und long zulässig sind; c hat denselben Typ wie a und b und die Referenzstufe 0.

Die binären Operatoren / und div bezeichnen ganzzahlige arithmetische Divisionen, die sich nur durch die Rundung des Ergebnisses unterscheiden. In dem Ausdruck c: a / b bzw. c: a div b müssen die Ausdrücke a und b vom gleichen Typ sein, wobei nur int und long zulässig sind; c hat denselben Typ wie a und b und die Referenzstufe 0. Eine Division durch 0 terminiert das Programm irregulär.

Bei der Division / wird das Ergebnis auf Intervallkante gerundet: Liegt das exakte Ergebnis genau zwischen zwei ganzen Zahlen, so ist die nähestliegende gerade Zahl das Ergebnis; andernfalls wird die nähestliegende ganze Zahl als Ergebnis angeliefert.

Bei der Division div gilt: a div b := Entier(a:b), wobei mit : die rundungslose arithmetische Division bezeichnet wurde.

Die binäre Operation mod bewirkt die zur Division div passende Restbildung. In dem Ausdruck c: a mod b müssen die Ausdrücke a und b vom gleichen Typ sein, wobei nur int und long zulässig sind; c hat denselben Typ wie a und b und die Referenzstufe 0. Für mod gilt: a mod b := a - (a div b) * b. Der Wert von a mod b liegt zwischen 0 (einschließlich) und dem Wert von b (ausschließlich); insbesondere besitzt der Wert von a mod b also dasselbe Vorzeichen wie der Wert von b. Konvention: a mod 0 = a.

Die binären Operatoren + und - bezeichnen die arithmetische Addition bzw. Subtraktion. In dem Ausdruck c: a + b bzw. c: a - b müssen die Ausdrücke a und b vom gleichen Typ sein, wobei nur int und long zulässig sind; c hat denselben Typ wie a und b und die Referenzstufe 0.

Die binären Operatoren <, ≤, =, ≥, > und ≠ bezeichnen die entsprechenden arithmetischen Vergleiche. In dem Ausdruck c: a < b bzw. c: a ≤ b bzw. c: a = b bzw. c: a ≥ b bzw. c: a > b bzw. c: a ≠ b müssen die Ausdrücke a und b vom gleichen Typ sein, wobei nur int und long zulässig sind; c hat den Typ bool und die Referenzstufe 0. Wie schon erwähnt, bedeutet z. B. der Ausdruck A < B ≤ C = D ≥ E > F ≠ G dasselbe wie der Ausdruck A<B ∧ B≤C ∧ C=D ∧ D≥E ∧ E>F ∧ F≠G, woraus folgt, daß die Größen A, B, C, D, E, F und G denselben Typ besitzen müssen, nämlich alle int oder long.

Die binären Operatoren ∧, ∨, <, > und ≡ bezeichnen die logische Und- bzw. Oder-Verknüpfung bzw. die logische Implikation nach links bzw. nach rechts bzw. die Äquivalenz. In dem Ausdruck c: a ∧ b bzw. c: a ∨ b bzw. c: a < b bzw. c: a > b bzw. c: a ≡ b müssen die Ausdrücke a und b vom Typ bool sein; c ist auch vom Typ bool und hat die Referenzstufe 0.

Bezüglich der beiden Implikationen gilt: a < b bedeutet: "a, falls b"; a > b bedeutet: "Falls a, dann b" oder auch "Aus a folgt b".

Der binäre Operator & bezeichnet die Konkatenation von Strings. In dem Ausdruck c: a & b müssen die Ausdrücke a und b vom Typ chain sein; c ist auch vom Typ chain und hat die Referenzstufe 0.

Die Folgeoperatoren if, then, else und fi beschreiben zusammen eine Alternative, die bei jeder Auswertung des Ausdrucks d: if a then b else c fi erneut getroffen wird. Der Ausdruck a muß vom Typ bool sein; die Ausdrücke b und c müssen typgleich sein, wobei alle fünf Typen zulässig sind; d ist vom gleichen Typ wie b und c und hat die kleinere der beiden Referenzstufen von b und c als Referenzstufe. Ist b oder c ein Ausdruck, der ein Feld definiert, so müssen b und c derartige Ausdrücke sein und typgleiche Felder definieren.

Werde d zur Laufzeit ausgewertet. Dann gilt: Zunächst wird a ausgewertet. Ergibt sich a zu true, so wird der Wert von d durch die Auswertung von b erhalten; ergibt sich a zu false, so wird der

Wert von d durch die Auswertung von c erhalten.

Die runden Klammern schließlich sind auch als Folgeoperatoren auffaßbar und haben die übliche Bedeutung. In dem Ausdruck b: (a) kann der Ausdruck a jeden der fünf Typen besitzen; b besitzt denselben Typ und dieselbe Referenzstufe wie a.

3. Handlungen

<action> ::= <label definition> | <declaration> | <statement>

<label definition> ::= <proper label>

Jede Standardbezeichnung * here für Marken wird also dadurch definiert, daß sie in einer aus ihr allein bestehenden Handlung auftritt. Wie schon gesagt muß es zu jeder solchen Standardbezeichnung genau eine solche Handlung geben.

<declaration> ::= ident <identifier list> = <information>

<identifier list> ::= <identifier> | <identifier list> ,
<identifier>

<information> ::= <identifier> | <constant> | <loc variable> |
<loc array> | <loc pointer>

<loc variable> ::= loc <type>

<type> ::= bool | int | long | label | chain

<loc array> ::= loc <integer> <type>

<loc pointer> ::= loc ref <type> | loc ref dim <type>

Deklarationen sind dazu da, für alle Größen frei wählbare Bezeichnungen einführen zu können; durch Verwendung der Bezeichnungen arbeitet man dann gerade mit diesen Größen.

Folgt auf ident eine Liste aus mehr als einem Identifikator, so ist die Deklaration als abkürzende Schreibweise für entsprechend viele Deklarationen aufzufassen, deren identifier list nur einen

Identifikator enthält. Damit bleibt nur noch die Bedeutung einer derartigen Deklaration zu klären.

Der auf ident folgende Identifikator stellt die neu einzuführende Bezeichnung dar; sie gilt innerhalb des kleinsten, die Deklarationsstelle umfassenden Blocks (vgl. 4) sowie in all denjenigen Unterblöcken dieses Blocks, in denen keine gleichbezeichnete Größe deklariert wird. In ein- und demselben Block (ohne Unterblöcke) kann eine Bezeichnung nicht mehrfach eingeführt werden.

Für welche Größe eine Bezeichnung eingeführt werden soll, geht aus der information hervor:

Handelt es sich um einen Identifikator I, so wird für diejenige Größe, die unmittelbar außerhalb desjenigen kleinsten Blocks unter I zugänglich ist, der die Stelle der momentan betrachteten Deklaration umfaßt, eine weitere Bezeichnung eingeführt. Dies ist ein wesentliches Hilfsmittel für die einfache Simulation von ALGOL 60-Prozeduren.

Handelt es sich um eine Konstante, so wird für eben diese Konstante eine Bezeichnung eingeführt. Besondere Erwähnung verdient die "Konstante" here (vgl. 1.11): Der ALGOL 60-Block

begin integer X, Y, Z; ... ; M: ... end

schreibt sich in PAL

begin ident X, Y, Z = loc int; ... ; ident M = here; ... end.

Auf diese Weise wird erreicht, daß die Bezeichnung M für den angedeuteten Programmort wie in ALGOL 60 nur in dem angedeuteten Block gilt.

Der Fall loc variable dient der Einführung einer Bezeichnung für eine einfache Variable; der Typ der Variablen wird hinter loc angegeben. Sie wird außerdem zur Laufzeit mit "undefiniert" initialisiert.

Der Fall loc array dient der Einführung einer Bezeichnung für ein Feld; hinter loc wird dabei die Dimension des Feldes angegeben

(sie muß größer als 0 sein); auf die Dimensionsangabe folgt der Typ des Feldes. Als Besonderheit im Vergleich zu ALGOL 60 gilt, daß keine unteren und oberen Grenzen anzugeben sind; diese werden zunächst auf "undefiniert" gesetzt. Der dynamisch erste Zugriff auf das Feld zur Laufzeit des Programms, etwa mit den Indizes I1, ... , In, bewirkt die Definition der Grenzenpaare zu I1:I1, ... , In:In. Für alle weiteren Zugriffe auf das Feld gilt, daß es jedesmal dann knappestmöglich ("quaderförmig") vergrößert wird, wenn mindestens ein Index außerhalb des zugehörigen Grenzenpaares liegt. Da diese Vergrößerungen beachtlich viel Zeit kosten können, wird man sie vermeiden oder wenigstens nur in größeren "Portionen" durchführen lassen. Beispiel: Das ALGOL 60-Programmstück

```
... begin integer array A[UG1:OG1, UG2:OG2, UG3:OG3]; ...
```

wird man z. B. wie folgt in PAL übersetzen:

```
... begin ident U = loc int;
          ident A = loc 3 int;
          A(UG1, UG2, UG3).=U; A(OG1, OG2, OG3).=U; ...
```

So besitzt A bereits nach zwei Zugriffen seine volle Größe, wobei alle seine Komponenten mit "undefiniert" vorbesetzt sind, da alle neu hinzukommenden Komponenten so vorbesetzt werden und auch U keinen definierten Wert besitzt. Im übrigen sei nochmal erwähnt, daß man mittels lwb und upb (wenn auch nur lesend) auf die jeweils aktuellen Grenzen eines Feldes zugreifen kann.

Der Fall loc pointer schließlich dient der Einführung einer Bezeichnung für einen Zeiger. Dieser zeigt entweder auf eine Variable (im Fall loc ref <type>) oder auf ein Feld (im Fall loc ref dim <type>) beliebiger, auch wechselnder Dimension (Dimensionskontrollen finden bei indizierten Vorkommen von Feldzeigern erst zur Laufzeit statt). Mit <type> hat man den Typ des Zeigers anzugeben. Zeiger werden zur Laufzeit ebenfalls mit "undefiniert" initialisiert.

Abschließend sei zu den Deklarationen noch angemerkt, daß sie nicht wie in ALGOL 60 am Anfang eines Blocks stehen müssen, sondern nur irgendwo innerhalb des Blocks, in dem sie gelten sollen (aber natürlich nicht in einem Unterblock dieses Blocks). Sie sind

auch markierbar: Eine Markendefinition wie auch eine Markendeklaration sind in PAL selbständige Handlungen und nicht nur Anhängsel einer Anweisung wie in ALGOL 60.

<statement> ::= <jump> | <assignation 1> | <assignation 2> |
<input> | <output>

<jump> ::= goto <expression>

Der Ausdruck hinter goto muß vom Typ label sein. Die Sprunganweisung hat dieselbe Bedeutung wie in ALGOL 60.

<assignation 1> ::= <left part> .= <right part>

<assignation 2> ::= <left part> := <right part>

<left part> ::= <identifier> | <subscripted identifier>

<right part> ::= <expression>

Der Fall assignation 1 stellt genau die von ALGOL 60 her bekannte Wertzuweisung dar; die linke Seite muß eine Variable sein (einfach oder indiziert) und damit eine Größe der Referenzstufe 1. Besitzt sie zunächst die Referenzstufe 2, so wird sie (implizit) dereferenziert. Die rechte Seite muß ein Ausdruck der Referenzstufe 0 sein, was ggfs. durch (implizite) Dereferenzierungen erzwungen wird und weiter ein Ausdruck, der genau eine einzelne Konstante als Wert liefert, welche sich der linksseitigen Variablen zuweisen läßt. Deshalb müssen linke und rechte Seite typgleich sein.

Der Fall assignation 2 stellt die Wertzuweisung an Zeiger dar: Die linke Seite muß ein Zeiger sein und damit eine Größe der Referenzstufe 2 und die rechte Seite muß ein Ausdruck der Referenzstufe 1 sein, was ggfs. durch eine (implizite) Dereferenzierung erzwungen wird. Ist die linke Seite ein Zeiger auf eine Variable, so muß die rechte Seite als Wert einen Bezug auf eine Variable (einfach oder indiziert) haben; ist die linke Seite ein Zeiger auf ein Feld, so muß die rechte Seite als Wert einen Bezug auf ein Feld haben. In jedem Fall müssen linke und rechte Seite wieder typgleich sein.

Ausdrücklich sei erwähnt, daß zwar ggfs. (implizite) Dereferenzierungen stattfinden; die dazu "inverse" Anpassung jedoch wird niemals durchgeführt. Ist also z. B. eine linke Seite von der Referenzstufe 0, so liegt in jedem Fall eine Fehlersituation vor.

<inlist> ::= <left part> | <inlist> , <left part>

<input> ::= inp <inlist>

Die Handlung inp <inlist> ist eine Abkürzung für eine entsprechend lange Folge
inp <left part> ; inp <left part> ; ... inp <left part> .
Für left part gilt dasselbe wie bei assignation 1; der entsprechenden Variablen wird durch Einlesen einer Konstanten vom Normaleingabemedium ein Wert zugeordnet; die Variable und die einzulesende Konstante müssen typgleich sein, wobei der Typ label unzulässig ist. Die Konstante ist durch ihre Standardbezeichnung zu bezeichnen, wobei man Zahlen auch maximal ein Vorzeichen voranstellen darf und long-Zahlen keinen einleitenden Punkt aufweisen müssen. Die Konstanten sind durch jeweils ein Komma voneinander zu trennen und hinter die letzte einzulesende Konstante ist ein Semikolon zu setzen.

<outlist> ::= <right part> | <outlist> , <right part>

<output> ::= out <outlist>

Die Handlung out <outlist> ist eine Abkürzung für eine entsprechend lange Folge
out <right part> ; out <right part> ; ... out <right part> .
Für right part gilt dasselbe wie bei assignation 1; der entsprechende Wert, der nicht vom Typ label sein darf, wird auf dem Normalausgabemedium in Standardbezeichnung ausgegeben, Strings jedoch ohne die sie begrenzenden Stringquotes und Zahlen mit einem Minimum an Dezimalstellen, Vorzeichen nur für negative Zahlen und bei long-Zahlen kein einleitender Punkt.

4. Blöcke, Programm, Spezialzeichendefinitionen, globale Parameter

<unit> ::= <action> | <block>

<block head> ::= begin <unit> | <blockhead> ; <unit>

<block> ::= <blockhead> end

<program> ::= bop <block> eop

Ein PAL-Programm beginnt also stets mit bop, dann folgt ggfs. ein (einleitender) Kommentar; dann folgt derjenige Block, der das eigentliche Programm ausmacht, dann folgt ggfs. ein (abschließender) Kommentar und dann eop. Besitzt das PAL-Programm einzulesende Daten, so beginnen diese frühestens in der auf eop unmittelbar folgenden Zeile.

Zur Darstellung von bop wird, wie schon in 1.1.1 erwähnt, eine Darstellung von 𝔽 durch ein Spezialzeichen benötigt, die in einer Spezialzeichendefinition zu vereinbaren ist. Daraus folgt, daß dem bop des PAL-Programms in jedem Fall noch eine Spezialzeichendefinition vorangehen muß; letztere soll deshalb als zusätzlicher integraler Bestandteil eines PAL-Programms aufgefaßt werden. Außerdem wird man im allgemeinen auch Darstellungen von 𝕊 und K durch Spezialzeichen benötigen und von der Möglichkeit Gebrauch machen, Funktionszeichen (z. B. ℂ, ℕ und ℙ, siehe 1.1.3) und vielleicht auch Spezialzeichen (siehe 1.1.4) durch Spezialzeichen darzustellen. Für all diese Zwecke ist die Spezialzeichendefinition gedacht.

Eine Spezialzeichendefinition ist eine Zeichenkette, die mit der Zeichenfolge () einzuleiten und auch abzuschließen ist. Die Folge () selbst darf dabei allenfalls von Zeilenwechseln unterbrochen sein; im übrigen können Zwischenräume und Zeilenwechsel beliebig auftreten, wobei ggfs. jedoch die Konventionen für Ersatzdarstellungen (siehe 1.1) einzuhalten sind. Von der bop vorangehenden Spezialzeichendefinition abgesehen kann man weitere auch überall zwischen zwei elementare Einheiten (vgl. 1.10) einfügen.

Um eine Darstellung von F bzw. S bzw. K zu vereinbaren, schreibe man Fy bzw. Sy bzw. Ky, wobei y das F bzw. S bzw. K darstellende Spezialzeichen andeutet und F bzw. S bzw. K die Funktion von Steuerbuchstaben haben. Um ein Funktions- oder Spezialzeichen z durch ein Spezialzeichen y darzustellen, schreibe man Dzy, wobei D ein weiterer Steuerbuchstabe ist; z wird man dabei meist in seiner Ersatzdarstellung angeben. Schließlich gibt es noch den Steuerbuchstaben X, der bei seinem Auftreten alle momentan gültigen Darstellungen durch Spezialzeichen annulliert (ohne daß der PAL-Programmierer wissen müßte, welche Darstellungen bis dahin galten).

Beispiel: Durch () F$ S' K" () wird vereinbart, daß (bis auf weiteres) F durch $, S durch ' und K durch " dargestellt wird (vgl. Beispiel in 1.10).

Da beliebig viele Spezialzeichendefinitionen in einem PAL-Programm zulässig sind, kann man immer wieder andere Darstellungen vereinbaren. Damit sind bis zu drei Ersetzungsprozesse verbunden, die in folgendem Beispiel alle gleichzeitig auftreten:

Beispiel: Der PAL-Programmierer vereinbart zunächst () F$ S' K" () und als nächste Vereinbarung () F" () . Dann gilt: Bis auf weiteres

a) wird F durch " dargestellt (ab sofort; dies ist wichtig für Ersatzdarstellungen, die noch innerhalb der Spezialzeichendefinition folgen könnten)

b) bezeichnet $ wieder sich selbst

c) gibt es für K keine Darstellung

Ein Spezialzeichen stellt also immer genau ein Zeichen dar (ggfs. sich selbst) und umgekehrt kann ein Zeichen immer nur durch höchstens ein Spezialzeichen dargestellt werden.

Nun fehlt zur Übersetzbarkeit eines PAL-Programms nur noch die Folge der 18 nichtnegativen ganzzahligen Werte der globalen Übersetzungsparameter, die stets als allererste Zeichenkette aufzu-

listen ist. Die Zahlen dieser Folge müssen vorzeichen- und zwischenraumlos sein; jede Zahl ist durch mindestens einen Zwischenraum und/oder Zeilenwechsel abzuschließen. Im einzelnen ist anzugeben:

1) Obere Grenze für BZLI, ≥ 21. Die compilerinterne Bezeichnungsliste BZLI wurde mit Hilfe eines ALGOL 60-Vektors realisiert, dessen obere Grenze hier anzugeben ist.

2) Obere Grenze für IDLI, ≥ 6. Die compilerinterne Identifikatorenliste IDLI wurde mit Hilfe von ALGOL 60-Vektoren realisiert, deren (gemeinsame) obere Grenze hier anzugeben ist.

3) Anfang der Zeilennumerierung des Protokolls, das der Compiler erstellt.

4) Schrittweite der Zeilennumerierung, ≥ 1.

5) Zeilenanzahl je Protokollseite, ≥ 1.

6) Anzahl der Leerzeilen, die nach jedem vom Protokolldruck ausgegebenen Seitenvorschub zusätzlich eingefügt werden sollen.

7) Anzahl der Zwischenräume, die der Zeilennumerierung voranzustellen sind

8) Anzahl der Spalten für die Zeilennumerierung, ≥ 2.

9) Anzahl der Spalten zwischen Zeilennumerierung und Markendefinitionen, ≥ 1. Dazu ist zu sagen, daß Markendefinitionen vom Protokolldruck in eigenen Zeilen ganz nach links (relativ zum sonstigen Programm) abgesetzt werden.

10) Anzahl der Spalten, um die die Markendefinitionen des PAL-Programms nach links aus dem ganzen sonstigen Programm herausragen sollen.

11) Anzahl der Spalten, die für Einrückungen des Programms (ohne seine Markendefinitionen) zur Verfügung stehen.

12) Anzahl der Spalten, die zusätzlich zu 7), 8), 9), 10) und 11) mindestens für das Protokoll zur Verfügung stehen, ≥ 20.

13) Anzahl der Spalten, um die jedes <u>begin</u> zusätzlich zum momentanen Einrückungsniveau von Handlungen * Markendefinitionen einzurücken ist.

14) Anzahl der Spalten, um die jede Handlung * Markendefinitionen zusätzlich zum Einrückungsniveau des begin's des aktuellen Blocks einzurücken ist.

15) Anzahl der Spalten, um die jede weitere Protokollzeile einer "überlangen" Handlung zusätzlich zum momentanen Einrückungsniveau von Handlungen * Markendefinitionen einzurücken ist.

16) Anzahl der Zwischenräume, durch die die Handlungskommentare des Programms innerhalb ihrer Handlungen oder Units von diesen abgehoben werden sollen (zusätzlich zu dem einen standardmäßig vorhandenen Zwischenraum).

17) Anzahl der Spalten, um die jede Zeile eines Zeilenkommentars zusätzlich zum momentanen Einrückungsniveau von Handlungen * Markendefinitionen einzurücken ist.

18) Anzahl der Spalten, um die jede weitere Protokollzeile einer "überlangen" Zeile eines Zeilenkommentars zusätzlich zum momentanen Einrückungsniveau von Zeilenkommentar einzurücken ist.

Die ersten beiden Werte sind notwendig, da der PAL-Compiler in ALGOL 60 geschrieben ist und man ALGOL-Felder nachträglich nicht vergrößern kann (im Gegensatz zu PAL-Feldern). Die restlichen Werte dienen alle der globalen Steuerung des Protokolldrucks. Bezüglich seiner lokalen Steuerung sei an die Protokollvorschübe in 1.10 erinnert.

5. Darstellungen

Die Darstellung von Ersatzdarstellungen wurde in 1.1 festgelegt, die von Protokollvorschüben in 1.10 und die von Spezialzeichendefinitionen in 4. Innerhalb von Strings und Kommentaren werden Zeilenwechsel des Quellprogramms ignoriert, Zwischenräume aber reproduziert. Strings und Kommentare können über mehrere Quellprogrammzeilen gebrochen werden, siehe 1.9 bzw. 1.10. Bezüglich der Darstellung von Identifikatoren, Zahlen und eigentlichen Marken gilt, daß sie keine Zwischenräume enthalten dürfen (als Ersatz

dafür wurden die eingestreuten Einzelpunkte zugelassen); ein Zeilenwechsel mit unmittelbar folgenden Zwischenräumen dagegen ist erlaubt und wird ignoriert. Bedeutungsvoll ist diese Konvention vor allem für long-Zahlen, denn die sollen ja ausdrücklich sehr lang sein können.

Für Wortsymbole und Delimiter schließlich gilt wie für Ersatzdarstellungen und Protokollvorschübe, daß ihre Darstellungen allenfalls Zeilenwechsel und sonst nichts enthalten dürfen; im einzelnen werden folgende Darstellungen akzeptiert:

Referenz-sprache	Darstellungen
+	+
-	-
*	*
/	/
↑o	↑o | FSCALE
<	< | FLT
≤	≤ | <= | ¬> | FLE | FNG
>	> | FGT
≥	≥ | >= | ¬< | FGE | FNL
=	= | FET
≠	≠ | ¬= | <> | >< | FNE
¬	¬ | FNOT
∧	∧ | FAND
∨	∨ | FOR
↓	↓ | FIVIM
↑	↑ | FIMPL
≡	≡ | FEQUIV
&	& | : | .. | FCONC
.=	.=
:=	:= | ..=
,	,
;	; | .,
(	(
)	)
abs	FABS
begin	FBEGIN
bool	FBOOL
bop	FBOP

Referenz-sprache	Darstellungen
chain	FCHAIN
dig	FDIG
dim	FDIM
div	// | FDIV
else	FELSE
end	FEND
eop	FEOP
false	FFALSE
fi	FFI
goto	FGOTO
here	FHERE
ident	FIDENT
if	FIF
inp	FINP
int	FINT
label	FLABEL
loc	FLOC
long	FLONG
lwb	FLWB
mod	/: | /.. | FMOD
out	FOUT
ref	FREF
sgn	FSGN
sgp	FSGP
sign	FSIGN
then	FTHEN
true	FTRUE
upb	FUPB

Als Abschluß der Definition von PAL bringen wir nun noch ein vollständiges Beispiel-Programm:

```
253 90 10 10 50 3 0 4 5 10 50 50 5 3 2 2 0 5
() F§ S' K" ()
§BOP
"DAS FOLGENDE ALGOL 60-PROGRAMM ERZEUGT ALLE KOMBINATIONEN ZU JE"
"M ELEMENTEN AUS DEN NATÜRLICHEN ZAHLEN 1, 2, ..., N :          "
"                                                               "
"'BEGIN'                                                        "
"       'INTEGER' M, N; INPUT(5, '('')', M, N);                 "
"       'IF' 1 <= M 'AND' M <= N 'THEN'                         "
"'BEGIN'                                                        "
"       'INTEGER''ARRAY' A[0:M];                                "
"                                                               "
"       'PROCEDURE' KOMB (MM, UG, OG);                          "
"       'VALUE' MM, UG, OG; 'INTEGER' MM, UG, OG;               "
"'BEGIN'                                                        "
"       'INTEGER' I;                                            "
"       A[UG] := A[UG - 1];                                     "
"WEITER:                                                        "
"       A[UG] := A[UG] + 1;                                     "
"       'IF' A[UG] > OG 'THEN' 'GOTO' ENDE;                     "
"       'IF' MM > 1 'THEN' KOMB (MM - 1, UG + 1, OG + 1) 'ELSE' "
"DRUCK:                                                         "
"'BEGIN'                                                        "
"       OUTPUT (6, '('/')');                                    "
"       'FOR' I := 1 'STEP' 1 'UNTIL' UG 'DO'                   "
"          OUTPUT (6, '('B3Z')', A[I])                          "
"'END';                                                         "
"       'GOTO' WEITER;                                          "
"ENDE:                                                          "
"'END' PROZEDUR KOMB;                                           "
"                                                               "
"       A[0] := 0; KOMB (M, 1, N - M + 1)                       "
"'END'                                                          "
"'END' ALGOL-BEISPIEL                                           "
```

```
"                                                                    "
"DIESES ALGOL-PROGRAMM WIRD NUN IN PAL ÜBERSETZT (WAS I. A. BEI "
"ALGOL-PROGRAMMEN MIT REKURSIVEN PROZEDUREN NICHT MÖGLICH IST). "
"DIE REKURSIVE PROZEDUR KOMB UND DAS HAUPTPROGRAMM WERDEN ZU PA-"
"RALLELEN BLÖCKEN GEMACHT; DIESE BEIDEN BLÖCKE KOMMUNIZIEREN    "
"ÜBER GRÖSEN MITEINANDER, DIE IN EINEM UMFASSENDEN BLOCK, DEM   "
"SOG. SYSTEMRAHMEN, VEREINBART WERDEN. DIESER RAHMENBLOCK ENT-  "
"HÄLT DIE FÜR ALLE GLOBALEN UND FORMALEN PARAMETER VON KOMB SO- "
"WIE FÜR DIE AUFRUFSVERWALTUNG BENÖTIGTEN GRÖSEN.               "
§PAGE§
§BEGIN "ANFANG DES PROGRAMMS UND DES SYSTEMRAHMENS"
       §IDENT SYPROZ, SYRÜCK = §LOC §LABEL;
       §IDENT SYNIVEAU, SYPAR1, SYPAR2, SYPAR3 = §LOC §INT;
       §IDENT SYFELD = §LOC 1 §INT; SYNIVEAU .= 0;
§BEGIN "RAHMEN DER PROZEDUR KOMB"
       §IDENT KOMB = SYPROZ;
       §IDENT FOR.RSPR, AKT.RSPR = SYRÜCK;
       §IDENT NIV = SYNIVEAU;
       §IDENT FP1, AP1 = SYPAR1;
       §IDENT FP2, AP2 = SYPAR2;
       §IDENT FP3, AP3 = SYPAR3;
       §IDENT A = SYFELD, KOMB .= ANF.KOMB;
       §GOTO NACH.KOMB; §IDENT ANF.KOMB = §HERE;
§BEGIN "PROZEDUR KOMB"
       §IDENT RSPR = §LOC 1 §LABEL;
       §IDENT MM, UG, OG = §LOC 1 §INT;
       NIV .= NIV + 1; RSPR(NIV) .= FOR.RSPR;
       MM(NIV) .= FP1; UG(NIV) .= FP2; OG(NIV) .= FP3;
       §IDENT I = §LOC §INT; "FÜR I IST KEIN VEKTOR NÖTIG"
       A(UG(NIV)) .= A(UG(NIV)-1);
       §IDENT WEITER = §HERE;
       A(UG(NIV)) .= A(UG(NIV))+1;
       §GOTO §IF A(UG(NIV)) > OG(NIV) §THEN ENDE §ELSE §HERE §FI;
       §GOTO §IF MM(NIV) > 1 §THEN §HERE §ELSE DRUCK §FI;
       AP1 .= MM(NIV)-1; AP2 .= UG(NIV)+1; AP3 .= OG(NIV)+1;
       AKT.RSPR .= NACH.AUFRUF1; §GOTO KOMB;
       §IDENT DRUCK = §HERE; §OUT '§076'; I .= 1;
```

```
        §IDENT NOCH.DRUCK = §HERE;
        §GOTO §IF I > UG(NIV) §THEN NACH.DRUCK §ELSE §HERE §FI;
        §IDENT ZWR = §LOC §CHAIN;
        ZWR .= §IF §DIG A(I) = 1 §THEN '   ' §ELSE §IF §DIG A(I)
               = 2 §THEN '  ' §ELSE ' ' §FI §FI;
        §OUT ZWR, A(I); I .= I+1; §GOTO NOCH.DRUCK;
        §IDENT NACH.AUFRUF1, NACH.DRUCK = §HERE;
        §GOTO WEITER; §IDENT ENDE = §HERE;
        NIV .= NIV-1; §GOTO RSPR(NIV+1)
§END "PROZEDUR KOMB";
        §IDENT NACH.KOMB = §HERE
§END "RAHMEN DER PROZEDUR KOMB";
§LINE2§
§BEGIN "RAHMEN DES HAUPTPROGRAMMS"
        §IDENT PROZ = SYPROZ;
        §IDENT AKT.RSPR = SYRÜCK;
        §IDENT AP1 = SYPAR1;
        §IDENT AP2 = SYPAR2;
        §IDENT AP3 = SYPAR3;
        §IDENT A = SYFELD;
§BEGIN "HAUPTPROGRAMM"
        §IDENT M, N = §LOC §INT; §INP M, N;
        §GOTO §IF 1 <= M <= N §THEN §HERE §ELSE ENDE §FI;
        §IDENT UNDEF.INT = §LOC §INT;
        A(0) .= 0; A(M) .= UNDEF.INT;
        AP1 .= M; AP2 .= 1; AP3 .= N-M+1;
        AKT.RSPR .= NACH.AUFRUF1; §GOTO PROZ;
        §IDENT NACH.AUFRUF1, ENDE = §HERE
§END "HAUPTPROGRAMM"
§END "RAHMEN DES HAUPTPROGRAMMS"
§END "ENDE DES PROGRAMMS UND DES SYSTEMRAHMENS"
§EOP
```

B) Implementierung der Sprache PAL

Die Implementierung der Programmiersprache PAL erfolgte durch Studenten in Form von einzelnen Fortgeschrittenen-Praktika. Demgemäß wurde der Compiler für PAL in einzelne nicht zu umfangreiche Läufe gegliedert. Als Sprache zur Formulierung des Compilers wurde ALGOL 60 gewählt, da diese bei den Studenten als bekannt vorausgesetzt werden konnte, wenn auch andere Sprachen wie etwa PASCAL für diesen Zweck fraglos günstiger gewesen wären. Die Quellsprache ist, wie schon gesagt, PAL; als Zielsprache wurde wegen seiner Maschinennähe BCPL verwendet. Im übrigen wurde nicht angestrebt, einen besonders effizienten Compiler zu erstellen; im Zweifelsfall wurde, da das Projekt ein Studienobjekt ist, logischen Gesichtspunkten höhere Priorität eingeräumt. Dieser Zielsetzung entsprechend wurde außerdem versucht, weitestmöglich gehende Maschinenunabhängigkeit zu verwirklichen.

I. Generelle Festsetzungen

1) Zur Kommunikation der Läufe

Die Läufe kommunizieren über Listen miteinander, die am Ende eines Laufs in Dateien eingetragen werden und vom nächsten Lauf aus diesen Dateien wieder gelesen werden. Eine Liste besteht aus einer oder mehreren Spalten und wird (aus ALGOL-Sicht) entsprechend durch ein oder mehrere eindimensionale ganzzahlige Felder mit unteren Grenzen 0 und gemeinsamer oberer Grenze realisiert. Die Läufe kommunizieren über folgende Listen miteinander: Die (allgemeine) Kommunikationsliste KOLI, die Bezeichnungsliste BZLI, die Identifikatorenliste IDLI und die einzelnen Zwischensprachenlisten ZWLIi. Die Struktur von KOLI wird im folgenden beschrieben, die von BZLI im Lauf 1, die von IDLI im Lauf 2 und die von ZWLIi im Lauf i.

KOLI ist eine einspaltige Liste und dient der allgemeinen Kommunikation der Läufe untereinander. Sie ist am Ende eines jeden

Laufs i konsistent definiert und enthält dann folgende Angaben:

Zelle	Inhalt
0	Verweis auf die letzte relevant besetzte Zelle von BZLI
1	Relevante Länge von ZWLIi
2	0 oder 1, je nachdem, ob Lauf i oder ein vorangegangener Lauf keinen oder einen Fehler des PAL-Programmierers fand
3	Anzahl der PAL-Zeichen je BZLI-Zelle ("CHARANZ")
4	- (10 ↑ (8. globaler Übersetzungsparameter)) ("ZLVORBES")
5	Obere Grenze für Vereinbarung von BZLI
6	Obere Grenze für Vereinbarung von IDLI
7	Anzahl der PAL-Zeichen der Implementation des Compilers
8	Maximale Anzahl von Indexausdrücken einer indizierten Größe
9	Verweis auf den ersten Hilfszellen-Eintrag in IDLI

Hierzu folgende Anmerkungen:

a) KOLI ist momentan nicht selbständig, sondern belegt die ersten zehn Zellen von BZLI, was schon von der Logik her gesehen sehr unschön ist

b) Die Angabe in Zelle 0 sollte besser in BZLI eingetragen werden; die Angabe in Zelle 1 besitzt nur Kommentarwert und kann entfallen; die Angabe in Zelle 9 ist fast unmittelbar aus IDLI entnehmbar und sollte deshalb hier nicht auftreten

c) Die Angabe in Zelle 3 ist der einzige auftretende maschinenabhängige Parameter. Diese Parameter sollten aber alle in KOLI auftreten. An weiteren Parametern sind vorhanden: Maximale Anzahl von PAL-Zeichen je BCPL-Element, maximale Ziffernanzahl einer ganzen BCPL-Zahl, die Zuordnung Zentralcodezeichen / PAL-Zeichen für das Einlesen des Quellprogramms und seiner Daten (Vektor "CH") und der Verweis auf den Anfang von CH in KOLI, maximaler Zentralcodewert (= obere Grenze von CH), die Zen-

tralcodewerte der Zeichen CR (carriage return), NL (new line), NF (new form), SP (space) und 0 (zero). Falls CR, NL und NF nicht explizit vorhanden sind, sollten sie als explizit vorhanden simuliert werden. Falls die Zentralcodewerte der Ziffern kein geschlossenes Intervall bilden oder steigenden Ziffern keine steigenden Zentralcodewerte entsprechen, reicht die Angabe der Null allein nicht.

d) Die Angabe in Zelle 7 ist der einzige auftretende implementierungsabhängige Parameter. Diese Parameter sollten aber alle in KOLI auftreten. An weiteren Parametern sind vorhanden: Obere Grenze für Vereinbarung von KOLI, Puffergröße für den Transport der Listen in Dateien und zurück, maximal ausnutzbare Breite einer Druckerzeile, Darstellung der PAL-Zeichen auf dem Drucker (Vektor "CR") und Verweis auf den Anfang von CR in KOLI, symbolische Gerätenummern (logische E/A-Kanäle) für E/A-Transporte des Compilers und der Runtime

e) Die Angaben in den Zellen 5 und 6 sind Angaben des PAL-Programmierers. Es wurde vergessen, noch zwei weitere derartige Angaben zu verlangen, nämlich eine gemeinsame obere Grenze für die Vereinbarung der Keller der Läufe und die Größe der während der Runtime benötigten Halde

f) Die Angaben in den Zellen 2, 4 und 8 werden von einzelnen Läufen erzeugt. Man sollte noch ein paar Reservezellen für weitere derartige Angaben vorsehen.

Aus den gemachten Anmerkungen resultiert folgender Vorschlag für eine revidierte Kommunikationsliste KOLI, auf die wir uns im folgenden stets beziehen werden. Es gilt: Nicht erwähnte Zellen sind Reservezellen; explizit genannte Zahlenwerte sind empfohlen, nicht verbindlich; SGNR bedeutet "symbolische Gerätenummer".

Zelle	Inhalt
0	Verweis auf Vektor CH (80)
1	Verweis auf Vektor CR (340)
10	Obere Grenze für Vereinbarung von KOLI (479)

Zelle	Inhalt
11	Obere Grenze für Vereinbarung von BZLI
12	Obere Grenze für Vereinbarung von IDLI
13	Obere Grenze für Vereinbarung von Kellern
14	Größe der Halde (in BCPL-Elementen)
20	Puffergröße beim Transport der Listen in Dateien und zurück (128)
21	maximal ausnutzbare Breite einer Druckerzeile (132)
22	Anzahl der PAL-Zeichen der Implementation des Compilers (hier 140)
30	SGNR für das Quellprogramm samt seinen unmittelbar folgenden Daten (kurz: für die PAL-Quelle) (10)
31	SGNR für das Protokoll der PAL-Quelle samt sonstiger Daten sowie für alle Meldungen des Compilers (15)
32	SGNR für Zwischenspeicherung aller Daten des Programms in Datei (16)
33	SGNR für Lesen von KOLI aus Datei (20)
34	SGNR für Schreiben von KOLI in Datei (25)
35	SGNR für Lesen von BZLI aus Datei (30)
36	SGNR für Schreiben von BZLI in Datei (35)
37	SGNR für Lesen von IDLI aus Datei (40)
38	SGNR für Schreiben von IDLI in Datei (45)
39	SGNR für Lesen von ZWLI(i-1) aus Datei (50)
40	SGNR für Schreiben von ZWLIi in Datei (55)
41	SGNR für Lesen des generierten BCPL-Programms durch den BCPL-Compiler aus Datei (60)
42	SGNR für Schreiben des generierten BCPL-Programms durch Lauf 4 in Datei (65)
43	SGNR für Lesen der zusätzlichen Daten des Programms durch Lauf 1 (0 oder 70)

Zelle	Inhalt
44	SGNR für Lesen der Daten des Programms zur Runtime aus Datei (71)
45	SGNR für Schreiben des Outputs des Programms zur Runtime (0 oder 75)
50	0 oder 1, je nachdem, ob Lauf i oder ein vorangegangener Lauf keinen oder einen Fehler des PAL-Programmierers fand
51	- (10 ↑ (8. globaler Übersetzungsparameter)) ("ZLVORBES")
52	Maximale Anzahl von Indexausdrücken einer indizierten Größe
53	Maximale Zeichenanzahl eines Eingabedatums
60	Anzahl der PAL-Zeichen je BZLI-Zelle ("CHARANZ"; 4)
61	Maximale Anzahl von PAL-Zeichen je BCPL-Element (hier 3)
62	Maximale Ziffernanzahl einer ganzen BCPL-Zahl (hier 7)
63	Maximaler Zentralcodewert (hier 255)
64	Zentralcodewert von CR (hier 22)
65	Zentralcodewert von NL (hier 21)
66	Zentralcodewert von NF (hier 23)
67	Zentralcodewert von SP (hier 175)
68	Zentralcodewert von 0 (hier 176)
80 ff.	Werte des Vektors CH (hier 256 Werte; näheres siehe bei Lauf 1)
340 ff.	Werte des Vektors CR (hier 140 Werte; näheres wird weiter unten erläutert)

Abschließend sei zu KOLI noch folgendes angemerkt:

1) Obere Grenzen für die Vereinbarung der ZWLIi sind nicht erforderlich, da die Zwischensprachen unmittelbar aus den bzw. in die betreffenden Dateien gelesen bzw. geschrieben werden können

2) Damit Lauf 1 bei seinem Start KOLI erstmalig lesen kann, ist er mit zwei "privaten" Daten zu versehen, nämlich mit der Lese-SGNR von KOLI und der Puffergröße. Es sind dies dieselben Werte wie in den Zellen 33 und 20 des Vorschlags für KOLI. Da KOLI i. a. in einer permanenten Datei liegen wird, ist KOLI von Lauf 1 zu regenerieren.

Über die Art und Weise der Kommunikation der Läufe wurde schon gesagt, daß die Listen während oder (meist) am Ende eines Laufs in Dateien abgelegt werden, aus denen sie vom jeweils nächsten Lauf wieder gelesen werden. Die für alle Fälle gültigen Konventionen für die Ablage von Listen in Dateien werden im folgenden genannt.

Im Rahmen des PAL-Compilers gibt es dreierlei Dateien:

a) Texthaltungsdateien

Eine derartige Datei gliedert sich in Sätze, welche aus Zentralcodezeichen bestehen, wobei jedoch keine Steuerzeichen außer NL als Satzendezeichen und EM (end of medium) als Datenendezeichen auftreten und NL und EM nur implizit vorhanden sind

b) Ausgabedateien

Eine derartige Datei ist nur für die Ausgabe auf dem Drucker bestimmt und gliedert sich in Sätze, welche aus Zentralcodezeichen bestehen. Jeder Satz besitzt als erstes Zeichen ein explizit vorhandenes Steuerzeichen und zwar ein Vorschubzeichen wie z. B. CR oder NL. Die restlichen Zentralcodezeichen sind keine Steuerzeichen; nur EM ist wieder implizit vorhanden. Das erste Zeichen des ersten Satzes ist (mindestens unter ALGOL) zwangsweise ein NF.

c) Ganzwortdateien

Eine derartige Datei gliedert sich in Sätze, welche aus Worten, d. h. den Inhalten (Bitmustern) von Speicherzellen, bestehen. Steuerzeichen sind, auch implizit, nicht vorhanden.

Als Texthaltungsdateien werden unterstellt die Dateien für die PAL-Quelle und für das generierte BCPL-Programm. Die einzige Eingabeprozedur, mit der man solche Dateien, deren Sätze ja von unbekannter variabler Länge sind, lesen kann, ist (im TR440-ALGOL) die Standardprozedur IN. Sie besitzt zwei Parameter. Der erste ist eine symbolische Gerätenummer, womit die zu lesende Datei festliegt; der zweite ist eine integer-Variable, der der Zentralcodewert des nächsten Zeichens zugewiesen wird; Satzende wird dabei als NL und Datenende als EM angeliefert. IN ist leider ziemlich langsam; wesentlich schneller wäre eine Variante davon, bei der der zweite Parameter ein Vektor ist, der auf einen Schlag mit den Zentralcodewerten der Zeichen des ganzen nächsten Satzes (oder noch besser mit frei wählbaren Codierungen davon) gefüllt würde und ggfs. mit entsprechend vielen NL's bzw. EM's aufgefüllt würde. In eine derartige Datei schreiben kann man mit allen Ausgabeprozeduren; um diese Schnittstelle aber klein zu halten, wurde festgelegt, nur die zu IN genau korrespondierende Prozedur OUT zu verwenden; diese Konvention wurde aber leider nicht immer eingehalten.

Als Ausgabedateien werden unterstellt die Datei für das Protokoll und die Meldungen und die für den Output des Programms. Eine derartige Datei ist wieder mit allen Ausgabeprozeduren beschreibbar und es sollte wieder nur OUT verwendet werden. Die Ausgabe eines Vorschubzeichens (es sind hier 16 davon vorhanden) durch OUT bewirkt den Abschluß des jüngst angefangenen Satzes und die Eröffnung des nächsten Satzes mit dem Vorschubzeichen als erstem Zeichen.

Als Ganzwortdateien werden die Dateien für alle noch nicht genannten Ein/Ausgabe-Ströme unterstellt; die Information befindet sich dabei in ALGOL-Vektoren vom Typ integer und soll geschrieben werden bzw. in solche Vektoren eingelesen werden. Dabei wird je Vektorkomponente ein Wort benötigt. Für diese Fälle wurde die Konvention getroffen (und auch eingehalten), die Information pufferweise zu schreiben und zu lesen. Der Puffer ist dabei ein ALGOL-Vektor integer array PU [0:x], wobei x um 1 kleiner ist als der

Wert in Zelle 20 von KOLI. Beim Schreiben wird zunächst die nächste Portion der betreffenden Liste in PU umgefüllt und anschließend PU auf einen Schlag als nächster Satz in die Datei geschrieben; gelesen wird entsprechend umgekehrt. Dazu werden die Standardprozeduren PUTS bzw. GETS verwendet, welche zweiparametrig sind. Der erste Parameter ist eine symbolische Gerätenummer und bestimmt so die Datei, der zweite Parameter ist PU. Das "S" der Namen dieser Prozeduren besagt, daß rein sequentiell gearbeitet wird; insgesamt wäre es geschickter gewesen, die ganz analogen Standardprozeduren PUT und GET zu verwenden, bei denen man durch einen weiteren Parameter die Nummer des anzusprechenden Satzes vorgeben kann.

Ergänzend muß gesagt werden, daß die Puffer PU eine Struktur besitzen: Die Zelle 0 von PU enthält nämlich keine Information des Ein/Ausgabe-Stromes, sondern gibt an, wieviel Worte in PU relevant besetzt sind; sie fungiert damit als "Verpackung" des Puffers. Es gilt folgende Konvention: PU[0] = y < x bedeutet: Es sind nur die Zellen 1 bis y von PU relevant gefüllt und PU ist der letzte Puffer (Satz) der Datei. Andernfalls ist PU[0] = x, alle x Informationszellen von PU sind relevant gefüllt und es existiert noch ein weiterer Puffer (Satz) (der möglicherweise leer ist, d. h. mit PU[0] = 0) in der Datei. Eine aus heutiger Sicht sehr unschöne Ausnahme wurde aus Bequemlichkeitsgründen für Puffer mit PU[0] = x zugelassen, die Information aus IDLI enthalten: Bei derartigen Puffern enthält die Zelle PU[x] entgegen der Angabe in PU[0] keine Information aus IDLI; auf diese Weise paßten genau 18 Zeilen der (7-spaltigen) IDLI in PU hinein.

Abschließend sei zum Dateiengebrauch noch festgehalten, daß die jeweilige Organisation von ALGOL-Operatoren bezüglich Dateien explizite Aufrufe der Prozeduren FORWIND und/oder REWIND erforderlich machen kann.

Zur Kommunikation der Läufe sei schließlich noch erwähnt, daß die Codierung der Delimiter in den Zwischensprachen in gewissen Grenzen (nämlich im Intervall [-99:-1]) frei wählbar ist; jeder Lauf

darf sich bei seinem Vorgänger bestellen, was immer er will.

2) Zur Darstellung der PAL-Zeichen

Es gibt wohl kaum einen Drucker, auf dem alle PAL-Zeichen direkt druckbar sind. Deshalb wurde generell die Konvention getroffen, daß alle PAL-Zeichen durch Übereinanderdruck zweier direkt druckbarer Zeichen dargestellt werden. Ist ein PAL-Zeichen direkt druckbar, so nehme man als zweites Druckerzeichen den Zwischenraum. In denjenigen Fällen, in denen ein PAL-Zeichen auch durch zwei direkt druckbare Zeichen nicht erkennbar dargestellt werden kann, gebe man das genau für diesen Zweck vorgesehene PAL-Zeichen ⓓ aus.

Die folgende Tabelle enthält alle PAL-Zeichen, die bezüglich ihrer Darstellung und/oder Bedeutung bemerkenswert sind.

PAL-Wert	Darstellung / Bedeutung
0	Darstellung durch 0 (Null) und / (Schrägstrich). Bedeutung: Beim Einlesen der Quelle: Ignore-Zeichen (wird sogleich weggeworfen); beim Protokolldruck, soweit es nicht als Füllzeichen eines Eintrags in BZLI benutzt wird: CR (carriage return)
1	Darstellung durch 1 (Eins) und /. Bedeutung: Beim Einlesen der Quelle: Quellzeilenende; beim Protokolldruck: NL (new line)
2	Darstellung durch 2 (Zwei) und | (Strich). Bedeutung: Beim Einlesen der Quelle: EM (end of medium); beim Protokolldruck: NF (new form)
3	Darstellung durch die Buchstaben O und I. Bedeutung: Darstellung eines in der aktuellen Implementierung nicht erkennbar darstellbaren (anderen) PAL-Zeichens
4	Darstellung durch die Buchstaben O und X. Bedeutung: Darstellung für einen Zentralcodewert, dem beim Einle-

PAL-Wert	Darstellung / Bedeutung
	sen der Quelle kein (anderes) PAL-Zeichen zugeordnet werden kann (substitute character)
5	Darstellung durch F und /. Bedeutung: Fluchtzeichen für Wortsymbole, Ersatzdarstellungen und Protokollvorschübe
6	Darstellung durch S und /. Bedeutung: Stringbegrenzer
7	Darstellung durch K und _ (Unterstrich). Bedeutung: In der Definition von PAL: Kommentarbegrenzer. Beim Protokolldruck: Anfang eines Kommentars, dem in derselben Quellzeile ein sichtbares Zeichen voranging
8	Darstellung durch K und ¯ (Überstrich). Bedeutung: (Beim Protokolldruck) Anfang eines Kommentars, dem in derselben Quellzeile kein sichtbares Zeichen voranging
9	Darstellung durch _ und ¯. Bedeutung: Keine (noch frei verfügbar)
51	Darstellung durch S und =. Bedeutung: Großes scharfes S
58	Darstellung durch $_{10}$ (Basiszehn).
61	Auf dem hiesigen Drucker Darstellung durch < (kleiner) und _.
62	Auf dem hiesigen Drucker Darstellung durch > (größer) und _.
64	Auf dem hiesigen Drucker Darstellung durch = und |.
68	Darstellung durch die öffnende spitze Klammer <
69	Darstellung durch die schließende spitze Klammer >
70	Auf dem hiesigen Drucker Darstellung durch = und _
75	Darstellung durch C und /
76	Darstellung durch N und /
77	Darstellung durch P und /
116	Auf dem hiesigen Drucker Darstellung durch ` (Gravis) und _ (auch wenn diese Darstellung die Grenze der Erkennbarkeit überschritten haben sollte)

PAL-Wert	Darstellung / Bedeutung
117	Auf dem hiesigen Drucker Darstellung durch ^ (Zirkumflex) und |
118	Auf dem hiesigen Drucker Darstellung durch v und |
119	Auf dem hiesigen Drucker Darstellung durch > und -
126	Bedeutung: accent aigue (Akut)
127	Bedeutung: accent grave (Gravis)
128	Bedeutung: accent circonflex (Zirkumflex)
129	Bedeutung: Öffnende Zeichenreihenklammer
130	Bedeutung: Schließende Zeichenreihenklammer
131	Bedeutung: Ringel (over dot)
132	Bedeutung: Tilde
133	Bedeutung: Unterstrich
134	Bedeutung: Überstrich
135	Auf dem hiesigen Drucker Darstellung durch < und >. Bedeutung: Multiplikationskreuz
136	Auf dem hiesigen Drucker Darstellung durch + und _. Bedeutung: Wer da? (Auslösung des fremden Namensgebers bei Fernschreibern)
138	Bedeutung: Kissen-Zeichen

Die Darstellung der PAL-Zeichen wird über den in KOLI enthaltenen Vektor CR gesteuert, woraus die prinzipiell vorhandene freie Wählbarkeit der Darstellungen der PAL-Zeichen folgt. Zu jedem PAL-Zeichen enthält CR die beiden zugehörigen Druckerzeichen Z1 und Z2 in der Form (Zentralcodewert von Z1)·i + (Zentralcodewert von Z2), wobei i der um 1 erhöhte Wert der Zelle 63 von KOLI ist.

3) Zur Maschinenunabhängigkeit und Dokumentation

Der Maschinenunabhängigkeit des Compilers dienen

a) eine ganze Reihe von Parametern in KOLI (siehe dort)

b) die Festsetzung, daß innerhalb des Compilers (auch zur Runtime) nur PAL-Zeichen bzw. deren Werte verwendet werden sollen (z. B. für auszugebende Meldungen)

c) die Schmalheit der E/A-Schnittstelle, die durch Beschränkung auf die Standardprozeduren IN und OUT sowie GETS und PUTS (bzw. GET und PUT) erreicht wird

Der Erreichung eines hohen Dokumentationswerts des Compilers dienen

a) die Benutzung von sinnreich benannten ALGOL-Größen. Insbesondere sollen für Codierungen und andere feste Werte entsprechend benannte Variable herangezogen werden, die mit den betreffenden Zahlenwerten belegt werden; niemals soll mit den Zahlenwerten selbst gearbeitet werden oder Eigenschaften von diesen explizit verwendet werden

b) die sog. Zwischendrucke ZDRi. Dabei handelt es sich um Hilfsprogramme, die den gesamten Output des jeweiligen Laufs i in direkt lesbarer logischer Form ausdrucken. Je schärfer sie dabei den Output kontrollieren, desto besser. Die Zwischendrucke sind auch zur Fehlersuche im Test- wie im Dauerbetrieb hervorragend geeignet, insbesondere dann, wenn man den Zwischendruck, statt ihn Material ausgeben zu lassen, von ihm früher erzeugtes Material, von dessen Richtigkeit man sich überzeugt hat, einlesen und auf Gleichheit mit dem aktuellen Material kontrollieren lassen kann. Auf diese Weise verlieren nachträgliche Änderungen des Compilers sehr viel von ihrer Problematik.

4) Zur Behandlung von Fehlern des PAL-Programmierers

Es wird ganz generell zwischen drei Sorten von Fehlern unterschieden:

a) Fehler, die zum sofortigen Abbruch der Übersetzung führen

b) Fehler, die nicht so gravierend sind, sodaß der aktuelle Lauf (bis auf weiteres) noch fortgesetzt werden kann. Nach diesem Lauf wird die Übersetzung dann jedoch abgebrochen, d. h. sie "gelten als Fehlersituation"

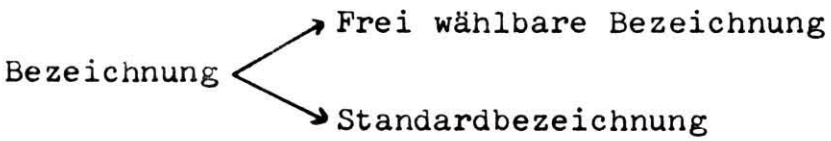
c) "Harmlose" Fehler, die nicht als Fehlersituation gelten, automatisch korrigiert und nicht reklamiert werden. Derartige Fehler gibt es nur sehr wenige und nur in sehr speziellen Kontexten; sie sollten dennoch durch Einreihung unter Fall b abgeschafft werden.

Erfährt Lauf i, $i \geq 2$, über die Zelle 50 von KOLI, daß einer seiner Vorgänger schon Fehler gefunden hat, so übergibt er seinem Nachfolger lauter leere Listen; nur KOLI bleibt unverändert. Das ist alles, was Lauf i in einem solchen Fall zu tun hat; er wird dann "so gut wie völlig kurzgeschlossen".

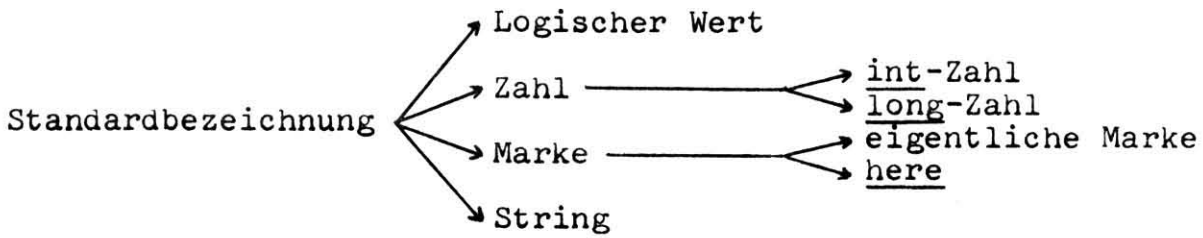
Der Compiler macht keine Anstrengungen, Fehler nach irgendwelchen Wahrscheinlichkeiten oder sonstigen Gesichtspunkten zu korrigieren; allenfalls wird der Keller ausreichend leergeräumt. Eine gute Fehlerbehandlung könnte allerdings im Rahmen von zusätzlichen Fortgeschrittenen-Praktika eingefügt werden.

II. Lauf 1

Da im folgenden der Begriff "elementare Einheit" sehr oft auftritt, sei daran erinnert, was darunter verstanden wird:



Frei wählbare Bezeichnung ⟶ Identifikator



Aufgabe von Lauf 1 (Scanner, lexikaler Analysator) ist es, eine PAL-Quelle von links nach rechts lesend in die entsprechenden elementaren Einheiten maximaler Länge zu zerlegen. Theoretisch gesehen ist er ein endlicher Automat mit Output (u. a. wird ein "schönes" Protokoll der Quelle erzeugt), der eine reguläre Menge zu erkennen hat, womit die zugrundeliegende Realisierungsidee genannt ist. Unter dem "Grundzustand" des Scanners wird dabei der Anfangszustand dieses Automaten verstanden. Auf Grund der Darstellung der Wortsymbole ergibt sich für den Übergangsgraphen des entsprechenden Automatenteils eine Baumstruktur. Diese wurde im Scanner exakt nachgebildet, was etlichen statischen Umfang des Scanners verursacht. Naheliegenderweise könnte man einem Wortsymbol sein Fluchtzeichen nicht voran-, sondern nachstellen, wodurch man den Umfang des Scanners nennenswert verkleinern könnte, was aber fol-

gende Darstellungsprobleme aufwirft: Folgt einem Identifikator ein Wortsymbol oder ein Protokollvorschub (eine Ersatzdarstellung kann nicht folgen), so müßte zunächst ein Zwischenraum folgen (ein Zeilenwechsel allein tut's nicht!), woraus folgt, daß Zwischenräume zur Trennung von elementaren Einheiten notwendig wären, was bisher nicht der Fall ist (Zeilenwechsel werden beliebig langen Identifikatoren zuliebe ignoriert). Deshalb wurde es vorgezogen, Wortsymbolen ihr Fluchtzeichen voranzustellen. Zu beachten ist dann allerdings, daß die FANO-Bedingung erfüllt ist.

Lauf 1 hat als Input KOLI und die PAL-Quelle und beginnt mit dem Einlesen der globalen Übersetzungsparameter in sehr maschinennaher Form, wozu ihn die Angaben in KOLI befähigen; dann folgt das Einlesen der Zeichenfolge () und ein Sprung zur Behandlung von Spezialzeichendefinitionen (diese muß ja generell vorhanden sein, da Spezialzeichendefinitionen auch überall sonst in der Quelle auftreten können). Ab den runden Klammern wird dabei mit den allgemeinen Leseprozeduren des Scanners eingelesen, was u. a. die sofortige Umcodierung der zunächst eingelesenen Zentralcodezeichen in PAL-Zeichen bewirkt. Für diese Umcodierung ist der in der Festlegung von KOLI erwähnte Vektor CH vorhanden. Er enthält zu jedem möglichen Zentralcodewert den zugehörigen PAL-Wert. Für die vorliegende Implementierung gilt dabei im einzelnen:

Zentralcodewert	TR440-Bedeutung	PAL-Wert	PAL-Bedeutung
16 - 31	Vorschübe	1	Quellen-Zeilenende
33	end of medium	2	end of medium
0 - 15, 32, 34 - 63, 255	Sonstige Steuerzeichen	0	Ignore-Zeichen
131	keine	70	Äquivalenzzeichen
135	keine	118	Pfeil nach unten
136	keine	119	Pfeil nach rechts
146	keine	135	Multiplikationskreuz
221	keine	51	Großes scharfes S

Zentralcodewert	TR440-Bedeutung	PAL-Wert	PAL-Bedeutung
sonstige	gewisse Bedeutung bzw. keine	entsprechend bzw. 4	dieselbe Bedeutung bzw. unzulässiges Zentralcodezeichen

Da PAL-Zeichen durch Spezialzeichen bezeichnet werden können, wird für alle Spezialzeichen nach der Codierung mit CH eine zweite Codierung mit einem scannerlokalen Vektor PH durchgeführt, um das letztlich gemeinte PAL-Zeichen zu ermitteln. Die Vorbesetzung von PH ist die Identität; genau durch Spezialzeichendefinitionen werden die Komponenten von PH geändert. In diesem Zusammenhang sei auf folgende wünschenswerte Erweiterung der Darstellungsmöglichkeiten hingewiesen: Man sollte auch diejenigen Delimiter, die nicht nur aus genau einem PAL-Zeichen bestehen, sowie alle Wortsymbole durch Spezialzeichen darstellbar machen und dies umso mehr, als man diesen elementaren Einheiten zweckmäßigerweise ohnehin PAL-Werte im Anschluß an diejenigen der je Implementierung vorhandenen PAL-Zeichen zuordnen wird; diese PAL-Werte wären dann in PH einzutragen. Den genannten elementaren Einheiten wurden hier folgende PAL-Werte zugeordnet:

PAL-Wert	Bedeutung	PAL-Wert	Bedeutung
140	abs	151	eop
141	begin	152	false
142	bool	153	fi
143	bop	154	goto
144	chain	155	here
145	dig	156	ident
146	dim	157	if
147	div	158	inp
148	:=	159	int
149	else	160	label
150	end	161	long

PAL-Wert	Bedeutung	PAL-Wert	Bedeutung
162	loc	168	sgn
163	lwb	169	sgp
164	mod	170	sign
165	out	171	then
166	.=	172	true
167	ref	173	upb

Die genannten PAL-Werte werden z. B. dazu verwendet, die Darstellungen dieser elementaren Einheiten für das Protokoll wie auch für die erste Zwischensprache ZW1 aufzufinden.

Der Output des Lauf 1 besteht aus 4 Strömen, nämlich der Bezeichnungsliste BZLI, der ersten Zwischensprache ZW1, den zum Programm gehörenden Daten und dem Protokoll der gesamten Quelle. Diese vier Ströme werden nun im einzelnen in dieser Reihenfolge besprochen.

BZLI ist eine einspaltige Liste. Lauf 1 trägt dort sämtliche Bezeichnungen, die im Quellprogramm auftreten, ein (daher der Name). Sie stehen damit auch allen nachfolgenden Läufen zur Verfügung. Ferner gibt es Einträge in BZLI, die nur während Lauf 1 und dann auch nur sehr kurzfristig existieren, nämlich Delimiter, Kommentare und Protokollvorschübe. Nur im Fehlerfall gibt es schließlich noch Einträge der Art error; dabei handelt es sich um fehlerhafte Teilzeichenketten der PAL-Quelle, wobei nur solche des Programms das Ende des Lauf 1 erleben und deshalb ggfs. vom folgenden Zwischendruck ZDR1 berücksichtigt werden müssen; im übrigen werden dann auf Grund der allgemeinen Fehlerphilosophie die weiteren Läufe nicht mehr gestartet bzw. so gut wie gänzlich kurzgeschlossen, sodaß auch derartige Einträge für die weiteren Läufe nicht existieren. Ansonsten gilt ganz generell, daß jeder Eintrag nur einmal auftritt; das impliziert für jeden Eintrag außer den kurzfristigen einen entsprechenden Suchprozeß. Schließlich ist zu erwähnen, daß BZLI mit true, false und here (in dieser Reihenfolge)

vorbesetzt ist.

Die Einträge in BZLI sind ihrer Natur nach von variabler Länge; je Eintrag wird deshalb eine variable Anzahl von Zellen benötigt. Die erste Zelle eines Eintrags enthält einen Verweis auf die erste Zelle des unmittelbar vorangehenden Eintrags; die Einträge sind auf diese Weise von hinten nach vorne verkettet. Die erste Zelle des ersten Eintrags enthält momentan ein undef in Form einer Null. Statt dieser Null sollte man besser einen Verweis auf die letzte relevant besetzte Zelle von BZLI hineinsetzen; diese Angabe ist nützlicher und entspricht der Anmerkung b zur momentanen Version von KOLI. Die letzte relevant besetzte Zelle ist die Zelle hinter dem letzten gültigen Eintrag; sie enthält einen Verweis auf die erste Zelle dieses Eintrags.

Die zweite Zelle jedes Eintrags enthält in den letzten vier Bits eine Codierung für die Art des Eintrags; es gibt dabei zehn Fälle:

Codierung	Art	Codierung	Art
1	identifier	6	chainconst
2	boolconst	7	error
3	intconst	8	comment
4	longconst	9	delimiter
5	labelconst	10	layout

Wie schon angedeutet, existieren für Lauf 1 alle zehn Arten (die zehnte Art ist die von Protokollvorschüben), für ZDR1 nur die ersten sieben und für alle sonstigen Läufe und Zwischendrucke nur die ersten sechs Arten.

Für Einträge der ersten neun Arten gilt, daß der restliche Teil der zweiten Zelle die Anzahl der PAL-Zeichen enthält, die die Protokolldarstellung des Eintrags aufweist; Zahlen jedoch weisen keine führenden Nullen und keine Einzelpunkte mehr auf. Weiter gilt, daß die restlichen Zellen des Eintrags die Werte der PAL-Zeichen

des Eintrags enthalten und zwar sind je Zelle soviele Werte aufgereiht, wie CHARANZ in KOLI[60] angibt. Die letzte Zelle des Eintrags ist nötigenfalls mit PAL-Zeichen Ø aufgefüllt. Als Feinheit ist noch zu erwähnen, daß bei String- und Kommentareinträgen die betreffenden Begrenzer Ś bzw. K selbst nicht in BZLI eingetragen werden.

Für Einträge der Art layout schließlich gilt, daß der restliche Teil der zweiten Zelle eine Dezimalzahl enthält, deren letzte Ziffer angibt, welche Zeichen explizit ins Protokoll einzuschieben sind. Dabei gibt es drei Fälle:

Codierung	Zeichen
1	Zwischenraum
2	Zeilenvorschub
3	Seitenvorschub

Die restlichen Ziffern der Dezimalzahl geben die Anzahl der Zeichen an, die einzuschieben sind. Weitere Zellen existieren bei einem derartigen Eintrag nicht. Es sei angemerkt, daß es konzeptuell konsequenter gewesen wäre, nur die letztgenannte Anzahl in den restlichen Teil der zweiten Zelle zu füllen und die folgende Zelle mit dem PAL-Wert der einzuschiebenden Zeichen (10 bzw. 1 bzw. 2) zu versehen.

Auch die ZW1 enthaltende Liste ZWLI1 ist eine einspaltige Liste. Sie enthält Verweise auf die Bezeichnungen in BZLI (statt der Bezeichnungen selbst) und Delimiter des Quellprogramms, beides in der Reihenfolge ihres Auftretens im Quellprogramm. Für spätere Fehlermeldungen ist ferner die Zeilenstruktur des Protokolls durch Einstreuung von Zeilenangaben vorhanden.

Die Einträge in ZW1 belegen jeweils eine Zelle (Normiertheit). Bezeichnungen bzw. die Verweise darauf sind positivwertige Einträge; derartige Zellen enthalten auf ihren letzten vier Bits die Art der Bezeichnung (in derselben Codierung wie in BZLI); der Rest

der Zelle enthält einen Verweis auf die erste Zelle des betreffenden Eintrags in BZLI. Angemerkt sei, daß es ökonomischer gewesen wäre, lediglich einen Verweis abzusetzen und zwar auf die zweite Zelle des betreffenden Eintrags in BZLI.

Delimiter müssen (wie schon erwähnt) durch negative ganze Zahlen > -100 codiert werden. Es wurde folgende Codierung festgelegt:

Codierung	Delimiter	Codierung	Delimiter
-1	¬	-22	∧
-2	abs	-23	∨
-3	sgp	-24	<
-4	sgn	-25	>
-5	sign	-26	≡
-6	dig	-27	&
-7	+	-28	(
-8	-	-29	)
-9	*	-30	if
-10	/	-31	then
-11	div	-32	else
-12	mod	-33	fi
-13	↑	-34	begin
-14	lwb	-35	end
-15	upb	-36	bop
-16	≮	-37	eop
-17	≯	-38	goto
-18	≤	-39	inp
-19	≥	-40	out
-20	=	-41	.=
-21	≠	-42	:=

Codierung	Delimiter	Codierung	Delimiter
-43	,	-49	bool
-44	;	-50	int
-45	ident	-51	long
-46	loc	-52	label
-47	ref	-53	chain
-48	dim		

Für die Codierung von Zeilenangaben sind (wie auch in den restlichen ZWi) die negativen ganzen Zahlen ≤ -100 reserviert. Zeilenangaben setzen sich additiv zusammen aus der negativwertigen Vorbesetzung ZLVORBES in KOLI[51] und der Nummer der Zeile (laut Protokoll) in negativer Form.

Die auf das Quellprogramm folgenden Daten werden momentan lediglich ohne jede Kontrolle in eine eigene (Texthaltungs-) Datei übertragen. Diese Behandlung ist (nicht nur) logisch gesehen sehr unbefriedigend, da der Scanner alle Hilfsmittel zur kompletten syntaktischen Kontrolle der Daten, nämlich die Behandlung von logischen Werten, Zahlen, Strings, Kommentaren, Protokollvorschüben, Spezialzeichendefinitionen und den Delimitern Plus (+), Minus (-), Komma (,) und Semikolon (;), zur Verfügung hat, wodurch die Leseprozeduren der Runtime praktisch kostenlos erheblich entlastet würden. Wie schon bei der Erläuterung der revidierten Fassung von KOLI angedeutet, sollte man deshalb jedes Datum in der Form, die es in BZLI hätte, pufferweise wie bei anderen einspaltigen Listen auch in eine eigene Ganzwortdatei eintragen; eine BZLI-analoge Verweisstruktur aufzubauen ist dabei jedoch unnötig; außerdem ist in diesem Zusammenhang KOLI[53] zu definieren. Es sei darauf hingewiesen, daß unterschiedliche Konventionen für ALGOL und BCPL eine Codeprozedur für das Lesen dieser Datei zur Runtime erforderlich machen können.

In diesem Zusammenhang sei noch folgende Empfehlung ausgesprochen: Man sollte dem PAL-Programmierer die Möglichkeit geben, durch An-

gabe der symbolischen Gerätenummer 0 oder 70 in Form eines weiteren globalen Übersetzungsparameters kundzutun, ob es eine (Texthaltungs-) Datei gibt, die noch zusätzliche Quelldaten enthält. Gibt er 70 an, werden die zusätzlichen Daten an die auf das Quellprogramm unmittelbar folgenden Daten von Lauf 1 angehängt; hieraus ergibt sich z. B. die Möglichkeit, den eigentlichen Daten ein paar Steuerdaten voranzustellen. Gibt er 0 (oder einen sonstigen Wert ≠ 70) an, sind nur die auf das Quellprogramm folgenden Daten vorhanden. Jedenfalls wird diese Gerätenummer in KOLI[43] notiert. Schon aus Symmetriegründen sollte dann für den Output des Quellprogramms zur Runtime eine spiegelbildliche Möglichkeit vorhanden sein durch Angabe der symbolischen Gerätenummer 0 oder 75 in Form eines weiteren globalen Übersetzungsparameters, welcher in KOLI[45] notiert und im Kapitel B.VI erläutert wird.

Das Protokoll der PAL-Quelle gemäß den 16 Angaben des PAL-Programmierers hierzu zu erstellen ist Aufgabe eines speziellen Teils des Scanners, des sog. Protokolldrucks. Dieser wurde als parameterlose Prozedur PROT realisiert, welche über globale Größen versorgt wird. PROT kann ohne weiteres kurzgeschlossen werden, ohne die Arbeit des restlichen Scanners (im folgenden kurz: Restscanner) oder sonstiger Teile des Compilers irgendwie zu beeinträchtigen; in dieser Hinsicht ist PROT also outputlos.

Die logische Zeilenbreite setzt sich additiv aus den Werten der globalen Übersetzungsparameter 7 - 12 (vgl. Seite 27) in folgender Weise zusammen:

7	8	9	10	11	12
LIRAND	ZLNUBR	ABST	MABR	EINRBR	PRGBR

Für die PAL-Quelle ohne ihre Markendefinitionen stehen genau die letzten beiden Intervalle EINRBR ("Einrückbreite") und PRGBR ("(min.) Programmbreite") zur Verfügung. Alle Einrückungen spielen sich dabei ausschließlich im Intervall EINRBR ab. Setzt man die Korrektheit der PAL-Quelle und die Abwesendheit von Protokollvorschüben voraus, so gilt weiter: Die erste elementare Einheit,

die vom Protokolldruck reproduziert wird, nämlich der Begrenzer bop, wird an den linken Rand des Intervalls EINRBR gesetzt (dieser Rand gilt als das momentane Einrückungsniveau für begin's). Folgt nun ein Handlungskommentar, so wird dieser i. a. nach Ausgabe von einem Zwischenraum mehr als Parameter 16 (siehe Seite 28) angibt, noch in dieselbe Zeile gesetzt (oder doch dort angefangen); folgt ein Zeilenkommentar, so wird dieser in die nächste Zeile i. a. unter Einrückung (relativ zu bop) gemäß Parameter 14 (vgl. Seite 28) zuzüglich Parameter 17 (vgl. Seite 28) gesetzt (oder doch dort angefangen). Von der Möglichkeit abgesehen, daß noch mehrere Kommentare folgen können, folgt schließlich (soweit es den Protokolldruck interessiert) ein begin, welches in die dann nächste Zeile i. a. unter Einrückung (relativ zu bop) gemäß Parameter 14 zuzüglich Parameter 13 (vgl. Seite 27) gesetzt wird, womit das nunmehr gültige Einrückungsniveau für begin's erreicht ist (Hervorhebung der Blockstruktur des PAL-Programms). Unter Einrückung relativ hierzu gemäß Parameter 14 findet man das jetzt gültige Einrückungsniveau für Handlungen * Markendefinitionen. Letztere werden stets am linken Rand des Intervalls MABR begonnen, im übrigen aber wie alle sonstigen Handlungen behandelt. Generell sei festgehalten, daß mit "Einrückungsniveau für Handlungen" immer dasjenige für Handlungen * Markendefinitionen gemeint ist. Soweit zur Illustration der Parameter 7 bis 18. Zu den Parametern 6 und 7 schließlich sei noch angemerkt, daß man mit ihrer Hilfe den bedruckten Teil einer Druckerseite beliebig nach unten und rechts verschieben kann.

Für den Protokolldruck gelten folgende weiteren Festsetzungen:

1) Situation: Es sei im Rahmen einer Handlung die nächste elementare Einheit E (z. B. ein Identifikator oder ein Handlungskommentar) ins Protokoll zu setzen. Paßt E noch in die angefangene Zeile, so wird E dort hingesetzt. Andernfalls wird geprüft, ob E unter Einrückung gemäß Parameter 15 (vgl. Seite 28) relativ zum momentanen Niveau für Handlungen (aber wie immer keinesfalls jenseits der Ränder des Intervalls EINRBR) zur Gänze in die nächste Zeile paßt. Wenn ja, wird E dorthin gesetzt. Wenn nicht, wird E noch in der angebrochenen Zeile begonnen und in

den nächsten Zeilen unter zusätzlicher Einrückung gemäß Parameter 15 fortgesetzt.

Speziell für Handlungskommentar gilt: Die Zwischenräume, die gemäß Parameter 16 dem Kommentar vorausgehen, werden am Anfang einer neuen Protokollzeile nicht reproduziert; die Zwischenräume, die gemäß Parameter 16 dem Kommentar folgen, werden nur soweit reproduziert, als sie in die angefangene Zeile noch hineinpassen.

2) Die Begrenzer bop, begin, end und eop werden stets in eine neue Zeile gesetzt und sofern sie nicht von Handlungskommentar und/oder Semikolon gefolgt werden, stehen sie allein in ihrer Zeile.

3) Handlungen werden stets in einer neuen Zeile begonnen.

4) Die Darstellung der Delimiter und Wortsymbole im Protokoll durch PAL-Zeichen ist frei wählbar; man kann sich so dem vorhandenen Drucker bestmöglich anpassen. Hier z. B. wurden zur Darstellung von Wortsymbolen Worte aus den betreffenden kleinen Buchstaben herangezogen, was eindeutig ist, da Identifikatoren und eigentliche Marken nur große Buchstaben aufweisen dürfen. Hat man keine kleinen Buchstaben, so wird man die Darstellung durch das Fluchtzeichen F und große Buchstaben wählen. Kann man gewisse Delimiter auch durch Doppeldruck nicht darstellen, so wird man die entsprechenden Wortsymbole zur Darstellung heranziehen; für alle Delimiter, die nicht direkt darstellbar sein könnten, gibt es Darstellungen durch Wortsymbole.

5) Von lexikalisch fehlerfreien PAL-Quellen Q muß der Protokolldruck ein Protokoll P erzeugen, für das im Prinzip gilt: Wird P auf einem Medium ausgegeben, von dem es als Quelle erneut einlesbar ist (etwa auf Lochkarten), so dürfen zwischen P und Q keinerlei syntaktische oder semantische Unterschiede vorhanden sein. Hieraus folgt sogleich, daß alle elementaren Einheiten, die von frei wählbarer Länge sind, über mehrere Zeilen brechbar sein müssen, da der Protokolldruck derartige Einheiten nicht unbedingt im Intervall PRGBR unterbringen kann.

Nehmen wir nun das Beispiel der Ausgabe des Protokolls auf Lochkarten. Dann wären die Intervalle LIRAND, ZLNUBR und ABST

wegzulassen und die Ausgabe von Zeilennummern zu unterdrücken. Im übrigen aber würde der PAL-Programmierer eine sehr übersichtlich gestaltete PAL-Quelle erhalten. Die Möglichkeit des sinnvollen Übereinanderdruckens bzw. -stanzens von einzelnen Zeichen hat man jedoch nicht; dementsprechend wird man diejenigen PAL-Zeichen, die im gewählten Lochkartencode nicht vorhanden sind, in ihre Ersatzdarstellungen auflösen bzw. die Darstellung durch Wortsymbole nehmen (je nach Kontext). Spezielle Schwierigkeiten macht die Darstellung der PAL-Zeichen F, S und $\underline{K}$. Der PAL-Programmierer hat dann anzugeben, durch welche (direkt stanzbaren) Spezialzeichen sie dargestellt werden sollen (nur auf nicht wieder einlesbaren Informationsträgern kann man sie direkt als F bzw. S bzw. $\underline{K}$ darstellen, also z. B. auf Druckerpapier). Wo diese darstellenden Spezialzeichen selbst sonst auftreten, hat man sie in ihre Ersatzdarstellungen aufzulösen. Insgesamt gesehen wird man dann sicher den Vektor CR in KOLI anders definieren und interpretieren und eine eigene Stanzpositionsrechnung durchführen.

Wir haben zu spät bemerkt, daß auch unser Protokolldruck in einem einzigen Punkt gegen diese Forderung verstößt (genauer gesagt: verstoßen muß): Ein Handlungskommentar, der an den Anfang einer neuen Zeile gesetzt wird, wird beim Wiedereinlesen zu einem Zeilenkommentar. Da es ohnehin konzeptuell wünschenswert ist, den Einfluß der Quellgestaltung (soweit sie frei wählbar ist, vgl. Seite 28/29) auf die Unterscheidung Handlungskommentar / Zeilenkommentar (und damit überhaupt völlig) auszuschalten, erscheint uns folgende Sprachänderung angebracht: Handlungskommentare sind durch $\underline{K}$ und Zeilenkommentare durch $\overline{K}$ zu begrenzen und mittels $\underline{KK}$ bzw. $\overline{KK}$ über mehrere Quellzeilen brechbar (in analoger Weise wie gehabt). Der Steuerbuchstabe K in Spezialzeichendefinitionen wird ersetzt durch die Steuerbuchstaben H (für Handlungskommentar, d. h. die Darstellung von $\underline{K}$) und Z (für Zeilenkommentar, d. h. die Darstellung von $\overline{K}$).

6) Zur Protokollierung der Daten ist lediglich zu sagen, daß sie die Intervalle EINRBR und PRGBR ausfüllen, ohne irgendwie eingerückt zu werden.

Dabei fällt auf, daß man die Daten z. B. wegen ihrer Menge vielleicht gar nicht protokolliert haben will; auch die Nichtprotokollierung eines (korrekten) Programms kann wünschenswert sein. Deshalb sollte man vom PAL-Programmierer einen weiteren globalen Übersetzungsparameter verlangen, über den er diesen Wunsch mitteilt. Vorschlag für eine Festlegung dieses Parameters:

Wert des Parameters	Bedeutung
0	Protokollierung von Programm und Daten
1	Protokollierung nur des Programms
2	Protokollierung nur der Daten
3	Keinerlei Protokollierung
sonst	Wie 0

7) Der Protokolldruck hat die zusätzliche Aufgabe, Zeilenangaben in ZW1 einzustreuen. Dazu setzt er durch Aufruf der Prozedur STORE jedesmal dann eine Zeilenangabe ab, wenn er eine neue Protokollzeile eröffnet. STORE sollte dabei so arbeiten, daß von mehreren unmittelbar aufeinanderfolgenden Zeilenangaben nur die jeweils letzte tatsächlich abgesetzt wird. Erst nach dem Aufruf des Protokolldrucks setzt der Restscanner ggfs. einen Verweis auf die betreffende elementare Einheit in ZW1 ab.

Die datenmäßige und funktionelle Schnittstelle zwischen dem Protokolldruck PROT und dem Restscanner ist wie folgt festgelegt: Zunächst werden vom Restscanner alle elementaren Einheiten außer Spezialzeichendefinitionen an das jeweilige Ende der BZLI angefügt gemäß den bereits genannten Konventionen, Zahlen jedoch zunächst mit ihren führenden Nullen und Einzelpunkten. Damit ist der jüngste Eintrag in BZLI, den man sich als zunächst nur provisorisch angefügt zu denken hat, ein Parameter für PROT. Für Strings und Kommentare gilt zusätzlich, daß die logische Variable ILLEGZEI anzeigt, ob der String bzw. Kommentar ein verbotenes Zentralcodezeichen enthielt oder nicht; für Kommentare gilt zu-

sätzlich, daß die Variable MERKKOM den Wert des PAL-Zeichens K bzw. $\overline{K}$ enthält, je nachdem, ob dem Kommentar in seiner Quellzeile ein sichtbares Zeichen vorangeht oder nicht; für Delimiter gilt zusätzlich, daß die Variable PRPAL den PAL-Wert des Delimiters enthält, da PROT ein paar Delimiter wie z. B. begin oder ";" explizit erkennen muß. Erst nach dem Protokolldruck wird die jüngste elementare Einheit vom Scanner weiter bearbeitet; z. B. wird als nächstes bei Zahlen die ursprüngliche Darstellung durch die Darstellung ohne führende Nullen und Einzelpunkte überschrieben; dann werden bei Einträgen der Art comment, delimiter und layout diese weggeworfen und bei allen sonstigen Einträgen gesucht, ob diese schon vorhanden sind und nur wenn dies nicht der Fall ist, werden sie (endgültig) an BZLI angefügt.

Es bleibt nun noch festzulegen, wie Lauf 1 in seinen speziellen Fehlersituationen reagiert.

Fehler mit sofortigem Abbruch sind:

1) Überlauf der Bezeichnungsliste BZLI. Meldung: "Abbruch: Bezeichnungsliste zu klein".

2) Unzulässige Zeichen beim Einlesen der globalen Übersetzungsparameter; dies ist (auf Grund der logischen Einteilung von Lauf 1) die einzige Fehlersituation im gesamten PAL-Compiler, in der keine spezielle Fehlermeldung gegeben wird. Wird andererseits für einen Parameter ein unzulässiger Wert angegeben, so wird der entsprechende Minimalwert als Wert genommen (keine Fehlersituation).

3) Auf die globalen Übersetzungsparameter folgt keine Spezialzeichendefinition. Meldung: "Abbruch: Programmanfang falsch".

4) Alle Fehler in Spezialzeichendefinitionen. Meldung: "Abbruch: Elementarer Fehler in Spezialzeichendefinition".

5) Das physikalische Ende der PAL-Quelle (end of medium) wird nicht erst nach eop gefunden. Meldung: "Abbruch: End-of-medium gefunden".

6) Die erste im Protokoll reproduzierte elementare Einheit ist *

bop. Meldung: "Abbruch: Programm beginnt nicht mit B O P".

7) Das Intervall ZLNUBR für die Zeilennumerierung ist zu klein bzw. die Zeilennummern werden zu groß. Meldung: "Abbruch: Platz für Zeilennumerierung zu klein".

Sonstige Fehler, soweit sie vom Scanner gefunden werden, sind:

1) Fehler innerhalb von Strings und Kommentaren:
   a) Falsche Ersatzdarstellungen (Ziffernanzahl < 3 oder unzulässiger Wert):
   F wird in sein darstellendes Spezialzeichen Z rückcodiert und Z wird im Protokoll reproduziert, gefolgt von den dann folgenden Ziffern. Gilt nicht als Fehlersituation; richtiger wäre es wohl, dies doch als Fehlersituation gelten zu lassen (Benutzung der Variablen ILLEGZEI) und F nicht rückzucodieren.

   b) Beim Brechen eines Strings oder Kommentars über mehrere Zeilen folgt auf SS bzw. KK als nächstes sichtbares Zeichen Z kein S bzw. K:
   Der String bzw. Kommentar wird normal beendet und mit Z im Grundzustand neu aufgesetzt. Gilt nicht als Fehlersituation; richtiger wäre es wohl, dies doch als Fehlersituation gelten zu lassen (Benutzung der Variablen ILLEGZEI) und ein S bzw. K explizit im BZLI-Eintrag aufzunehmen, sodaß im Protokoll dann SS bzw. KK erscheint und das unrichtige Brechen sichtbar wird.

   c) Es wird ein Zentralcodewert gefunden, dem kein PAL-Zeichen zugeordnet ist:
   Zunächst wird das PAL-Zeichen ⊠ in den String bzw. Kommentar aufgenommen, ILLEGZEI auf true gesetzt und weitergelesen. Nach Erreichen des Endes des Strings bzw. Kommentars setzt der Protokolldruck diesen in eine eigene Zeile, beginnend am linken Rand des Intervalls MABR; ferner wird das Intervall ABST mit möglichst vielen Plus-Zeichen gefüllt. Gilt als Fehlersituation.

2) Fehler in den restlichen elementaren Einheiten (Bezeichnungen außer Strings; Delimiter; Protokollvorschübe):

Sie gelten jedenfalls als Fehlersituation. Wurde die jüngst analysierte Einheit bereits als Identifikator, eigentliche Marke oder Zahl erkannt (es sind dies gerade diejenigen elementaren Einheiten, deren Ende nur durch ein Zeichen erkannt werden kann, welches nicht mehr zu ihnen gehört), so wird sie als solche ordnungsgemäß abgeschlossen; die übrig bleibenden Zeichen bzw. in allen sonstigen Fällen die bisher (in BZLI) aufgereihten Zeichen sowie in jedem Fall das fehlerverursachende Zeichen selbst werden nunmehr als außerordentliche elementare Einheit E der Art error behandelt, d. h. solange beim Weiterlesen nur Buchstaben, Ziffern, Zwischenräume, Einzelpunkte oder im Grundzustand verbotene Zeichen auftreten, werden diese an E angefügt, bis schließlich ein anderes Zeichen Z auftritt. Dann wird E an den Protokolldruck weitergereicht, der E in eine eigene Zeile, beginnend am linken Rand des Intervalls MABR, setzt und das Intervall ABST mit möglichst vielen Plus-Zeichen füllt. Dann wird die Analyse im Grundzustand mit dem Zeichen Z fortgesetzt.

In jedem Fall aber (außer, wie schon erwähnt, bei fehlerhaften globalen Übersetzungsparametern) verabschiedet sich Lauf 1 mit einer der drei folgenden Meldungen: Mit "Lauf 1 ohne Fehler beendet", wenn die Quelle lexikalisch richtig war oder der Fehler nicht als Fehlersituation gilt bzw. mit "Quelle lexikalisch falsch (siehe die mit + markierten Zeilen)", wenn solche existieren bzw. sonst mit "Quelle lexikalisch falsch".

III. Lauf 2

Generell kann man sagen, daß Lauf 2 die Aufgabe hat, den Lauf 3 zu einer vollständigen semantischen Analyse zu befähigen und ihm diese Aufgabe durch die Durchführung einer möglichst weitgehenden syntaktischen Analyse sowie, wo immer möglich und zweckmäßig, durch Ansammeln, Gliedern und Ergänzen von Information zu erleichtern. Im einzelnen hat Lauf 2 folgenden Katalog von Aufgaben zu erledigen:

1) Erstellung der Identifikatorenliste IDLI. Näheres siehe weiter unten.

2) Vollständige syntaktische Analyse des PAL-Programms (bzw. von ZW1), soweit sie bei den natürlicherweise noch fehlenden Informationen über die durch Identifikatoren bezeichneten Größen (wie z. B. Typen, Arten und Referenzstufen) erledigt werden kann. Inwieweit man bei Konstanten diese dann bekannten Informationen berücksichtigt, bleibt freigestellt; bei syntaktischen Strukturen, die Eintragungen in IDLI verursachen, kann und sollte man jedoch sofort erschöpfend kontrollieren. Speziell bei Operatoren soll nur deren unäre bzw. binäre Verwendung kontrolliert werden, wobei (vorübergehend) alle binären Operatoren dieselbe Priorität haben, die allerdings kleiner ist als diejenige der unären Operatoren.

3) Kontrolle der Markendefinitionen auf nicht mehrmaliges Definiertwerden je Marke.

4) Prüfung auf nicht mehrmaliges Deklariertwerden eines Identifikators in ein- und demselben Block (ohne Unterblöcke).

5) Folgende Einzelheiten:

   a) Ersetzung der here's durch generierte eigentliche Marken der Gestalt . <digit sequence> , welche außerdem an BZLI anzufügen sind sowie Erzeugung entsprechender Markendefinitionen (mit Eintragung in IDLI). Damit ist die Behandlung von here's auf diejenige von eigentlichen Marken zurückgeführt. Mit <digit sequence> sind die ganzen Zahlen 0, 1, 2 usw. (je nach Bedarf) gemeint.

b) Voranstellen des Delimiters labdef vor Markendefinitionen (dies gilt auch für diejenigen von Absatz a).

c) Auflösen der Listen hinter inp und out durch Erzeugung entsprechend vieler Einzelanweisungen.

d) Aufspalten von + und - in unäre und binäre Operatoren.

e) Aufspalten der runden Klammern in arithmetische (weiterhin runde) und Indexklammern (eckige Klammern).

f) Setzen genau eines Semikolons hinter jede Markendefinition, jede Anweisung und jeden Block.

g) Transformation von bop in begin und eop in end (von Lauf 1 her ist sichergestellt, daß, von Zeilenangaben abgesehen, ZW1 mit bop beginnt und mit eop endet).

Dem Lauf 2 stehen als Input KOLI, BZLI und ZWLI1 zur Verfügung; diese Listen wurden bereits besprochen. Als Output liefert Lauf 2 (neben Meldungen) eine i. a. verlängerte BZLI, ferner die Zwischensprache ZW2 und die Identifikatorenliste IDLI. Diese drei Ströme werden nun im einzelnen in dieser Reihenfolge besprochen.

In BZLI sind alle auf Grund von Aufgabe 5a generierten Marken gemäß den allgemein gültigen Konventionen eingetragen worden. Insbesondere ist der einleitende Punkt (wie bei allen Marken) explizit in BZLI vorhanden.

ZWLI2 ist eine einspaltige Liste und enthält die Bezeichnungen und Zeilenangaben in unveränderter Form; für letztere gilt wieder, daß man von einer ununterbrochenen Folge von Zeilenangaben nur die letzte wirklich absetzen sollte. Die Delimiter (alte und neue) wurden diesmal wie folgt codiert:

Codierung	Delimiter	Codierung	Delimiter
-1	+u	-4	sgp
-2	-u	-5	sgn
-3	abs	-6	sign

Codierung	Delimiter	Codierung	Delimiter
-7	dig	-28	≡
-8	¬	-29	&
-9	lwb	-30	(
-10	upb	-31	)
-11	↑	-32	[
-12	*	-33	,
-13	/	-34	]
-14	div	-35	if
-15	mod	-36	then
-16	+b	-37	else
-17	-b	-38	fi
-18	<	-39	begin
-19	>	-40	end
-20	≤	-41	.=
-21	≥	-42	:=
-22	=	-43	goto
-23	≠	-44	labdef
-24	∧	-45	inp
-25	∨	-46	out
-26	<	-47	;
-27	>		

Im Vergleich zu ZW1 fällt vor allem auf, daß alle Deklaratoren verschwunden sind; das liegt daran, daß es in PAL keine Deklarationen gibt, die programmgenerierend wären, d. h. die im Zielprogramm die Generierung eines Programmstücks erforderlich machen würden. Im übrigen unterscheidet sich ZW2 von ZW1 nicht wesentlich (vgl. Aufgabenkatalog).

Die Identifikatorenliste IDLI besteht aus den sieben Spalten DINF, DENTITY, DBEZUG, DZEILE, DTYP, DART und DDIM. Sie enthält die Blockstruktur des PAL-Programms in Form von "Struktursymbol-Einträgen" sowie alle deklariert werdenden Identifikatoren, alle definiert werdenden Marken (auch die von Aufgabe 5a) und alle Standardbezeichnungen * Marken des PAL-Programms in Form von "Größen-Einträgen". Im folgenden wird der genaue Aufbau dieser Einträge festgelegt und die betreffenden Codierungen angegeben. Allgemein gilt: Die Codierung von undef ist 0.

In Struktursymboleinträgen steht ein begin oder end des Programms in folgender Form:

DINF[i] = structsy (1)

DENTITY[i] = begin (1) oder end (2)

DBEZUG[i] = Verweis auf das zugehörige andersartige Struktursymbol in IDLI

DZEILE[i] = Nummer der Protokollzeile des Struktursymbols

Anmerkung: DZEILE wurde erst nachträglich eingefügt und so wurde übersehen, daß für das von bop herrührende begin der Fall DZEILE[i] = 0 möglich ist und nicht undef bedeutet. Abhilfe: Man nehme die Zeilennummer in der Codierung von ZW1. Oder: Man codiere undef als -1 und nehme here als ersten BZLI-Eintrag (damit wäre dann die Codierung von undef umkehrbar eindeutig).

DTYP[i] = undef

DART[i] = undef

DDIM[i] = undef

In Größeneinträgen steht ein Identifikator oder eine Standardbezeichnung. Für Identifikatoren gilt:

DINF[i] = entity (2)

DENTITY[i] = Verweis auf den deklariert werdenden Identifikator I in BZLI (statt des Identifikators selbst)

DBEZUG[i] = undef, falls die information der Deklaration (vgl. Seite 20) mit loc beginnt. Andernfalls ist sie eine Bezeichnung B und es wird zunächst ein negativer Verweis auf B in BZLI hier abgesetzt, im Fall B = here jedoch gleich ein negativer Verweis auf die here ersetzende eigentliche Marke in BZLI (vgl. Aufgabe 5a). Am Ende des Lauf 2 wird statt dessen ein positiver Verweis auf denjenigen Eintrag E in IDLI abgesetzt, der mit B (im Fall, daß B ein Identifikator ist: letztlich) gemeint ist, was daran kenntlich ist, daß DBEZUG von E gleich undef ist.

DZEILE[i] = Nummer der Protokollzeile der Deklaration des Identifikators I

DTYP[i] = bool (1) bzw. int (2) bzw. long (3) bzw. label (4) bzw. chain (5)

DART[i] = const (1) bzw. var (2) bzw. array (3) bzw. varptr (4) bzw. arrptr (5)

DDIM[i] = Dimension bei Feldern, sonst undef

Für Standardbezeichnungen gilt:

DINF[i] = entity (2)

DENTITY[i] = Verweis auf die Standardbezeichnung in BZLI (statt der Bezeichnung selbst)

DBEZUG[i] = undef

DZEILE[i] = Nummer der Protokollzeile der Definition bei Marken (auch bei denen von Aufgabe 5a), sonst undef

DTYP[i] = bool (1) bzw. int (2) bzw. long (3) bzw. label (4) bzw. chain (5)

DART[i] = const (1)

DDIM[i] = undef

Mit Rücksicht auf Lauf 3 ist die Zeile 0 von IDLI wie folgt zu besetzen:

DINF[0] = entity; die restlichen Spalten sind auf undef zu setzen. Mit diesem Eintrag werden (im Fehlerfall) alle nichtdeklarierten Identifikatoren und alle nichtdefinierten Marken simuliert. In Zeile 1 steht das von bop herrührende begin. Dann folgen alle Standardbezeichnungen, die keine Marken sind. Dann folgt dasjenige begin, welches das eigentliche PAL-Programm einleitet. Dann folgen alle sonstigen Strukturymbole sowie alle deklariert werdenden Identifikatoren in der Reihenfolge von ZW1. Nach dem das eigentliche PAL-Programm abschließenden end folgen alle definiert werdenden Marken (auch die von Aufgabe 5a); sie müssen zunächst am Ende von IDLI angesammelt werden und können erst am Ende von Lauf 2 nach hier übertragen werden. Dann folgt das von eop herrührende end.

Dies sind die Konventionen für IDLI, soweit sie Lauf 2 betreffen. In Lauf 3 wird sich herausstellen, daß wegen der Menge der als Relationen zugelassenen Teilausdrücke (vgl. Seite 15, <relation>) die von ihm kreierten Hilfszellen nicht immer ein Intervall bilden; sie werden deshalb einzeln verwaltet und zwar mit Hilfe von folgenden (zusätzlichen) Größeneinträgen in IDLI, die von Lauf 3 hinter das letzte end gesetzt werden:

DINF[i] = entity (2)

DENTITY[i] = undef, da für Hilfszellen keine expliziten PAL-Namen kreiert werden (wohl aber BCPL-Namen)

DBEZUG[i] = free (1); intermediär tritt während Lauf 3 hier auch set (2) auf, d. h. es wird angemerkt, ob die Hilfszelle gerade frei ist oder nicht

DZEILE[i] = undef

DTYP[i] = undef

DART[i] = aux (6)

DDIM[i] = undef

Lauf 2 erledigt seine Aufgaben nach der üblichen Kellermethode (für die genannten Aufgaben ist mir keine andere, ähnlich effiziente und übersichtliche Methode bekannt) unter Zuhilfenahme eines

Kellers, der die drei Spalten KPUSHED, KSTATE und KREF aufweist. Im einzelnen gilt:

KPUSHED[i] = bop (1) bzw. decl (2) bzw. labdef (3) bzw. block (4) bzw. begin (5) bzw. assig (6) bzw. goto (7) bzw. out (8) bzw. inp (9) bzw. if (10) bzw. then (11) bzw. else (12) bzw. "(" (13) bzw. "[" (14) bzw. unop (15) bzw. binop (16) bzw. entity (17)

KSTATE[i] = eop (1) bzw. secoend (2) bzw. unit (3) bzw. leftpart (4) bzw. expr (5)

KREF[i] = Verweis auf das zugehörige begin in IDLI im Fall eines gekellerten bop oder begin, sonst empty (0)

Die dritte Spalte wird nur dazu benötigt, in IDLI die Verweise korrespondierender Struktursymbole aufeinander einzutragen; davon abgesehen sind alle Kellereinträge Paare (KPUSHED[i], KSTATE[i]). An möglichen Paaren sind vorhanden:

(bop, eop)
(decl, secoend)
(labdef, secoend)
(block, secoend)
(begin, unit)
(assig, leftpart)
(assig, expr)
(goto, expr)
(out, expr)
(inp, leftpart)
(if, expr)
(then, expr)
(else, expr)
((, expr)
([, expr)
(unop, expr)
(binop, expr)
(entity, leftpart)
(entity, expr)

Hieran sieht man deutlich, daß KSTATE (2. Komponente) stets angibt, was in Zusammenhang mit dem gekellerten Symbol (1. Komponente) erwartet wird. Speziell an assig und entity ist gut zu erkennen, ob man sich noch links (leftpart) oder schon rechts (expr) vom Zuweisungszeichen befindet. Dem Eintrag entity kann im

Quellprogramm nicht nur eine Bezeichnung, sondern auch eine indizierte Variable und im Fall von expr ein ganzer Ausdruck entsprechen.

Lauf 2 beginnt seine Tätigkeit nach einer Serie von Vorbesetzungen im wesentlichen mit dem Einlesen des initialen bop (stets vorhanden) und kellert dementsprechend (bop, eop); dieser Eintrag wird erst beim Lesen des finalen eop entfernt. Dann liest Lauf 2 ein begin ein (Kontrolle nötig) und kellert dementsprechend (begin, unit). Damit hat Lauf 2 seine Hauptschleife

```
SPEICHERN:   STORE( INPCOMP);
LESEN:       READ( INPCOMP);
KASKADE:     goto KASK[ ASSOCINT( INPCOMP)];
```

erreicht. Dabei enthält INPCOMP den jeweils nächsten ZW1-Eintrag, der keine Zeilenangabe ist (letztere werden innerhalb von READ abgehandelt). ASSOCINT ist eine Funktionsprozedur, die jedem Wert von INPCOMP eine natürliche Zahl zuordnet und damit diese Werte in (hier: 32) Klassen einteilt. Von der Behandlung einer jeden Klasse springt Lauf 2 dann nach SPEICHERN, LESEN oder KASKADE zurück.

Nun sind noch ein paar Worte zur Fehlerbehandlung des Lauf 2 zu sagen. Generell gilt, daß im Fehlerfall der Fehlerort in Form einer Zeilenangabe mit ausgegeben wird.

Fehler mit sofortigem Abbruch sind:

1) Überlauf der Bezeichnungsliste BZLI. Meldung: "Abbruch in Zeile x: BZLI im Lauf 2 zu klein"
2) Überlauf der Identifikatorenliste IDLI. Meldung: "Abbruch in Zeile x: Identifikatorenliste im Lauf 2 zu klein"
3) Überlauf des Kellers. Meldung: "Abbruch in Zeile x: Keller des 2ten Laufs zu klein"

Für die sonstigen Fehler (sie gelten alle als Fehlersituationen) gibt es eine logische Variable CORRACT, die angibt, ob der bisherige Teil der jüngsten Handlung korrekt war. Nur wenn er dies war,

wird eine Fehlermeldung gegeben und CORRACT auf false gesetzt, um weitere (meist Folge-) Fehlermeldungen zu dieser Handlung zu unterdrücken. Als Ausnahme zu dieser Konvention werden jedoch folgende Fehler stets gemeldet:

1) Fehler im Zusammenhang mit den Delimitern bop, begin und eop. Meldungen: "Zeile x: Weiteres B O P gefunden" bzw. "Zeile x: B E G I N ergibt Fehler" bzw. "Zeile x: Falsche Blockstruktur" (d. h. eop kommt zu früh oder zu spät)

2) Fehler im Zusammenhang mit Deklarationen der Form

   ident <identifier 1> = <proper label> oder
   ident <identifier 1> = <identifier 2> .

   Dann ist im Fall von <proper label> dieses nicht definiert bzw. im Fall <identifier 2> stößt man bei der Suche nach der letztlich durch <identifier 1> bezeichneten Größe auf eine nichtdefinierte Marke oder einen nichtdeklarierten Identifikator. Meldung: "Zeile x: Unausführbare Deklaration".

3) Ein- und dieselbe Marke wird mehrfach definiert (innerhalb des gesamten Programms) oder ein- und derselbe Identifikator wird mehrfach deklariert (innerhalb desselben Blocks ohne Unterblökke). Meldungen: "Zeile x: Marke mehrfach definiert" bzw. "Zeile x: Mehrfachdeklaration im selben Block".

Sonstige Fehler unterliegen, wie gesagt, der Regulierung durch CORRACT. Zwei davon ergeben eine besondere Meldung:

1) Auf bop folgt kein begin. Meldung: "Zeile x: Falscher Programmanfang"

2) Eine Deklaration ist syntaktisch falsch oder die Dimensionsangabe eines Feldes ist gleich 0. Meldung: "Zeile x: Fehlerhafte Deklaration".

In den übrigen Fällen erfolgt die Meldung "Zeile x: y ergibt Fehler", wobei y diejenige (jüngst in ZW1 gelesene) elementare Einheit ist, die nicht zu einem obersten Abschnitt des Kellerinhalts paßt. Wünschenswert ist es, y stets direkt in Klartext anzugeben,

wobei man bei Bezeichnungen die Erläuterung "Bezeichnung" einfügen sollte; momentan wird bei Bezeichnungen und Operatoren (außer + und -) statt des konkreten y nur der Standardtext "Identifikator" bzw. "Konstante" bzw. "Unärer Operator" bzw. "Binärer Operator" ausgegeben.

Im Korrektheitsfall andererseits verabschiedet sich Lauf 2 mit der Meldung "Lauf 2 ohne Fehler beendet".

Abschließend sei noch eine Sprachänderung empfohlen, deren Berücksichtigung im wesentlichen nur Lauf 2 und allenfalls noch den Protokolldruck von Lauf 1 betrifft.

Es hat sich als unpraktisch herausgestellt, den Rücksprungort für jeden zu simulierenden Prozeduraufruf explizit durch eine (jeweils neue) Bezeichnung benennen zu müssen, was nicht nötig wäre, könnte man dabei here verwenden. Dazu folgender Vorschlag: Man füge auf Seite 25 vor der Produktion für unit die Produktion

<action list> ::= <action> | <action list> , <action>

ein, ersetze die Produktion für unit durch

<unit> ::= <action list> | <block>

und lege für here fest (vgl. Seite 13), daß jedes Exemplar davon denjenigen Programmort bezeichnet, der unmittelbar hinter derjenigen Handlungsliste (nicht mehr: Handlung) liegt, in der das Exemplar von here auftritt. Dieser Vorschlag löst das genannte Problem auf elegante Weise, erfordert allerdings aus Eindeutigkeitsgründen, daß man input- und output-Listen, sofern sie echte Listen sind, in runde Klammern setzt, d. h. die Produktionen für input und output (vgl. Seite 24) ersetzt durch

<input> ::= inp <left part> | inp (<inlist>) bzw.

<output> ::= out <right part> | out (<outlist>) .

Ferner hat Lauf 2 dann handlungstrennende Kommata wie Semikolons zu behandeln mit Ausnahme der Tatsache, daß die Erzeugung der Definition der here ersetzenden generierten Marke wegfällt, d. h. in praxi, bis zum nächsten Semikolon oder end aufgeschoben wird.

Die Aufgaben und der Output von Lauf 2 bleiben von dieser Sprachänderung unberührt.

Ändert man die momentane Implementierung des Protokolldrucks nicht, so werden wegen seiner Abfragen auf Semikolons und end's nicht mehr Handlungen, sondern nur noch Handlungslisten jeweils in einer neuen Zeile begonnen, was unserer Meinung nach auch wünschenswert ist. Will man nach wie vor jede Handlung in eine neue Zeile setzen, so muß man handlungstrennende Kommata als solche erkennen. Das geht z. B. so, daß man einen Zähler bei jeder öffnenden bzw. schließenden runden Klammer um 1 erhöht bzw. erniedrigt. Bei korrekten PAL-Programmen gilt dann, daß genau die Kommata bei Zählerstand 0 handlungstrennende Kommata sind.

IV. Lauf 3

Zum Verständnis der Aufgaben von Lauf 3 ist es notwendig, die Struktur und die Eigenschaften von ZW2 vor Augen zu haben.

ZW2 ist, von begin's, end's und Semikolons abgesehen, eine Sequenz von Handlungen, die keine Deklarationen sind, d. h. von Anweisungen (Sprüngen, Zuweisungen und atomaren Ein- und Ausgabeanweisungen) und Markendefinitionen. Durch die syntaktischen Kontrollen des Lauf 2 ist weiterhin sichergestellt, daß jede dieser Handlungen wie auch ZW2 insgesamt eine formal (d. h. syntaktisch) zulässige Sequenz von Delimitern und Bezeichnungen ist.

Lauf 3 leistet nun die vollständige Dekomposition der genannten Handlungen in "Befehle" mit unterschiedlichen Anzahlen von Adressen und/oder Direktoperanden in der Weise, daß die entstehende klammerlose Befehlssequenz wirkungsmäßig zum Quellprogramm äquivalent ist. Im folgenden werden die möglichen Befehle aufgeführt und erläutert (womit gleichzeitig die logische Struktur von ZW3 beschrieben wird). Dazu sind jedoch zunächst noch ein paar Festsetzungen nötig, vor allem bezüglich der Interpretation der Befehle.

Für jeden in ZW2 auftretenden Delimiter mit Ausnahme von begin, end, Semikolon und den runden Klammern ist mindestens ein Befehl vorgesehen, der dann beim Abarbeiten des Delimiters zu generieren ist; nur für die eckigen Klammern und das (Indizes trennende) Komma sowie für if, then, else und fi, jeweils zusammen, existiert nur ein Befehl. Die Operanden der Befehle werden wie folgt abgekürzt:

OP: Bezeichnung eines Operanden (schlechthin). Ein ggfs. folgender dritter Buchstabe spezifiziert den Operanden näher; ggfs. wird OP auch indiziert.

KO: Bezeichnung einer PAL-Konstanten.

HZ: Bezeichnung einer Hilfszelle.

DIM: Natürliche Zahl (Dimension bzw. Indexanzahl).

BED: Bezeichnung des Werts eines auf if folgenden logischen Ausdrucks.

Alle außer diesen Abkürzungen in einem Befehl auftretenden Zeichen und Zeichenreihen ergeben zusammen die Bezeichnung des Befehls selbst; sie wurden so gewählt, daß sie direkt an die betreffenden Delimiter erinnern. Im übrigen wird die auch sonst bei Befehlen üblicherweise in bestimmten Kontexten vorhandene implizite Dereferenzierung unterstellt; so bedeutet LOAD a nicht, daß die Adresse a, sondern der Inhalt der Zelle mit der Adresse a in den Akkumulator zu laden ist, wohingegen bei STORE b keine implizite Dereferenzierung durchzuführen ist. Um bei als Operanden auftretenden PAL-Konstanten die implizite Dereferenzierung nicht immer explizit verhindern zu müssen, sind alle PAL-Konstanten (im Rahmen der Befehlsinterpretation) als Bezeichnungen für schon vor Start des Programms entsprechend initialisierte Variable aufzufassen, die während des Programms unverändert bleiben. Von den so vorhandenen Initialisierungen tauchen in ZW3 nur diejenigen für Marken explizit auf (wegen deren Ortsabhängigkeit). So ergibt sich, daß DIM der einzige vorhandene Direktoperand ist.

Einadreßbefehle:

goto OP ;

Infolge der soeben festgelegten Befehlsinterpretation bezeichnet OP letztlich eine Variable, die ihrerseits die anzuspringende Adresse enthält; es wird hier also eine implizite Dereferenzierung unterstellt

labdef KO ;

Dieser Befehl bezeichnet genau die für Marken wegen ihrer Ortsabhängigkeit nötige Initialisierung: Der (letztlich vorhandenen) Variablen KO wird die dem Programmort dieses Befehls entsprechende Maschinenadresse zugewiesen; letztere ist damit ein impliziter Operand dieses Befehls, der durch die Ortsabhängigkeit dargestellt wird. Für KO selbst wird keine implizite Dereferenzierung unterstellt

inp OP ;
Schema von vier verschiedenen Befehlen; es wird keine implizite Dereferenzierung unterstellt

out OP ;
Schema von vier verschiedenen Befehlen; es wird eine implizite Dereferenzierung unterstellt

Zweiadreßbefehle:

HZ := unop OP ;
unop steht für einen unären Operator; Schema von 15 verschiedenen Befehlen

OPD := OPS ;
D wie destination und S wie source; Schema von 7 verschiedenen Befehlen

dim DIM OP ;
Leistet die Dimensionskontrolle bei indizierten Vorkommen von Feldzeigern; OP gibt den Namen des Feldes an, DIM ist die Soll-Dimension von OP

Dreiadreßbefehle:

HZ := OPL binop OPR ;
binop steht für einen binären Operator; L wie left und R wie right; Schema von 34 verschiedenen Befehlen

HZ := OPL binop OPR save ;
Schema von 6 verschiedenen Befehlen. Diese Sonderform ist aus Rücksicht auf die zur Runtime vorzusehende garbage collection nötig und kann nur auftreten, wenn binop ein Relationsoperator und OPR eine Hilfszelle vom (momentanen) Typ long ist; sie besagt dann, daß OPR entgegen den sonst gültigen Gesetzen einen später noch benötigten Wert enthält. Ein ganz analoger Effekt kann auch dann auftreten, wenn OPR vom Typ int statt long ist; weil save für die garbage collection dann aber uninteressant ist, wird in ZW3 diese Sonderform dann nicht abgesetzt. Diese Entscheidung ist zwar ökonomisch, logisch gesehen aber sehr unschön, da es damit in Lauf 3 eine (einzige)

Stelle gibt, bei der letztlich spezielle Einzelheiten von Runtime-Darstellungen durchschlagen

Vieradreßbefehle:

HZ := cond BED OPT OPE ;
cond wie condition, T wie then und E wie else; Schema von 8 verschiedenen Befehlen

Multiadreßbefehle:

HZ := indvar OP_0 OP_1 ... OP_n ;
Liefert die Adresse einer indizierten Variablen mit n Indizes; OP_0 gibt den Namen des Feldes an, OP_1 bis OP_n liefern die Werte der Indizes; die Korrektheit des Wertes von n ist sichergestellt

Lauf 3 benützt für die Dekomposition der Handlungen in Befehle einen Keller, der alle dazu nötigen Informationen enthält; er enthält darüber hinaus noch weitere Informationen, die es gestatten, ZW2 semantisch zu kontrollieren wie z. B. die Kontrolle jedes Vorkommens einer Bezeichnung auf gemäß Kontext zulässigen Typ, Art und Referenzstufe. Alle semantischen Kontrollen (außer der weiter unten genannten) finden hier genau immer dann statt, wenn reduziert wird, d. h. wenn ein oberster Kellerabschnitt durch einen einzelnen Eintrag ersetzt wird und ein entsprechender Befehl in ZW3 abgesetzt wird. Die gerade erwähnte Ausnahme ist die Kontrolle auf Deklariertheit bei Identifikatoren (Berücksichtigung der Blockstruktur) bzw. auf Definiertheit bei Marken; diese Kontrolle könnte man zwar auf den Augenblick der Reduktion verschieben, sie wird hier jedoch schon nach dem Lesen der Bezeichnung in ZW2 durchgeführt, da jeder PAL-Bezeichnung sogleich ihr internal identifier, d. h. ein Verweis auf ihre Eintragung in IDLI, zugeordnet wird und dann in Lauf 3 nur noch mit diesem internal identifier gearbeitet wird; im übrigen ist diese Zuordnung aus Eindeutigkeitsgründen ohnehin notwendig und IDLI enthält ja auch alle Informationen über die Bezeichnung (oder doch Verweise darauf). Klappt die Zuordnung Bezeichnung - internal identifier nicht, so liegt (genau dann) ein nichtdeklarierter Identi-

fikator bzw. eine nichtdefinierte Marke vor, womit der Lokalisierungsteil der sogleich ausgegebenen Fehlermeldung besonders genau wird.

Damit sind die beiden Aufgaben von Lauf 3 (Dekomposition und semantische Analyse) beschrieben. Als Detailaufgabe ist hinzuzufügen, daß Lauf 3 KOLI[52] zu definieren hat.

Dem Lauf 3 stehen als Input KOLI, BZLI, IDLI und ZWLI2 zur Verfügung; diese Listen wurden bereits besprochen. Als Output liefert Lauf 3 (neben Meldungen) eine unveränderte BZLI, eine i. a. verlängerte IDLI und die Zwischensprache ZW3. An IDLI werden von Lauf 3 nur Hilfszellen-Einträge angefügt; ihr Aufbau wurde bereits beschrieben. Bleibt noch ZW3 zu besprechen.

ZWLI3 ist eine einspaltige Liste, welche im wesentlichen die einzelnen Bestandteile der bereits besprochenen Befehle der Reihe nach enthält; das Ende eines Befehls ist über das stets vorhandene Semikolon einfach zu finden.

Zeilenangaben werden unverändert übernommen; von mehreren in ZW3 unmittelbar aufeinanderfolgenden Zeilenangaben wird man wieder nur die letzte tatsächlich absetzen.

Die Operanden der Befehle werden als die Bezeichnungen von ZW3 aufgefaßt. Sie werden als nichtnegative Werte codiert, nämlich direkt als die entsprechenden internal identifier. Die einzige Ausnahme bildet der Operand DIM des Befehls dim DIM OP ; ; für ihn wird diejenige natürliche Zahl abgesetzt, die sein Wert ist. Alle sonstigen Bestandteile der Befehle werden als Delimiter von ZW3 aufgefaßt. Sie wurden wie folgt codiert:

Codierung	Delimiter	Codierung	Delimiter
-1	+ u int	-4	- u long
-2	+ u long	-5	abs int
-3	- u int	-6	abs long

Codierung	Delimiter	Codierung	Delimiter
-7	sgp int	-34	> int
-8	sgp long	-35	> long
-9	sgn int	-36	≤ int
-10	sgn long	-37	≤ long
-11	sign int	-38	≥ int
-12	sign long	-39	≥ long
-13	dig int	-40	= int
-14	dig long	-41	= long
-15	¬	-42	≠ int
-16	lwb	-43	≠ long
-17	upb	-44	∧
-18	10 int	-45	∨
-19	10 long	-46	<
-20	* int	-47	>
-21	* long	-48	≡
-22	/ int	-49	&
-23	/ long	-50	save
-24	div int	-51	val
-25	div long	-52	ref
-26	mod int	-53	dim
-27	mod long	-54	indvar
-28	+ b int	-55	cond
-29	+ b long	-56	:= bool
-30	- b int	-57	:= int
-31	- b long	-58	:= long
-32	< int	-59	:= label
-33	< long	-60	:= chain

Codierung	Delimiter	Codierung	Delimiter
-61	:= var	-68	inp long
-62	:= arr	-69	inp chain
-63	:= ptr	-70	out bool
-64	goto	-71	out int
-65	labdef	-72	out long
-66	inp bool	-73	out chain
-67	inp int	-74	;

Hier fällt vor allem auf, daß die meisten Delimiter, die in ZW2 vorhanden waren und nun wieder vorhanden sind, in mehrere Fälle aufgesplittet wurden, so daß Typ oder Art der von ihnen bearbeiteten Werte unmittelbar ersichtlich sind. + u int z. B. bedeutet, daß dies dasjenige unäre Plus ist, welches nur auf Werte vom Typ int angewendet werden kann. Weiter gilt: Bei dem Befehl inp OP ; z. B. wurde gesagt, daß dies ein Schema von 4 Befehlen ist; diese Befehle lauten: inp bool OP ; , inp int OP ; , inp long OP ; und inp chain OP ; . Analoges gilt für die Befehle out OP ; und OPD := OPS ; . Bei dem Befehl HZ := cond BED OPT OPE ; wurde von 8 verschiedenen Befehlen gesprochen; da es nur ein einziges cond gibt, folgt daraus, daß für das Zuweisungszeichen alle 8 Möglichkeiten vorhanden sind. Bei den Befehlen HZ := unop OP ; und HZ := OPL binop OPR ; überlagern sich diese beiden Effekte: Operator und Zuweisungszeichen geben den Typ der betroffenen Werte an. Die 6 Fälle der Sonderform HZ := OPL binop OPR save ; schließlich lauten: HZ := bool OPL long rel op OPR save ; , wobei für long rel op die sechs Relationsoperatoren für long-Werte einzusetzen sind.

Folgende Delimiter bedürfen einer Erläuterung:

a) Es gibt insgesamt 8 Zuweisungszeichen. Die ersten 5 geben an, daß ein Wert (im Sinn von PAL) zu speichern ist (im Fall := label z. B. eine als Sprungadresse benutzbare Maschinenadresse). Die nächsten beiden sind für die Speicherung von Be-

zügen auf Variable bzw. Felder vorgesehen; ein Bezug auf eine Variable ist dabei ihre Adresse; ein Bezug auf ein Feld ist die Adresse der betreffenden "Feldzelle"; letztere verweist auf den Anfang des zum Feld gehörenden Informationsvektors, der seinerseits u. a. auf die erste Zelle des Feldes selbst verweist (der internal identifier eines Feldes wird entsprechend als interne Bezeichnung eben dieser Feldzelle gehandhabt). Das letzte Zuweisungszeichen mit der Codierung -63 schließlich tritt nur in Verbindung mit dem cond-Befehl auf und zwar nur bei Alternativen, deren then- und else-Teil (blanke) Zeiger sind und bedeutet die Speicherung eines Bezugs auf einen Zeiger. Ein solcher Bezug ist dabei die Adresse des Zeigers.

b) Der Delimiter val wird einem Operand immer dann vorangestellt, wenn dieser zusätzlich explizit zu dereferenzieren ist (an impliziten Dereferenzierungen haben wir nur noch diejenigen auf Grund der Befehlsinterpretation). In diesem Zusammenhang sei daran erinnert, daß in Verbindung mit der Definition von Zuweisungen (vgl. Seite 23) automatische Dereferenzierungen (aus der Sicht des PAL-Programmierers) zugesagt wurden. Diese Dereferenzierungen, abzüglich derer durch die Befehlsinterpretation, sind gerade diejenigen, die Lauf 3 explizit zu generieren hat. Allen Operanden OP, OPD, OPS, OPL, OPR, BED, OPT, OPE und OP_i werden u. U. val's vorangestellt, in keinem Fall aber mehr als zwei val's.

c) Der Delimiter ref dient der expliziten Verhinderung einer durch die Befehlsinterpretation implizit gegebenen Dereferenzierung, sodaß statt des Werts des Operanden seine Bezeichnung (d. h. letztlich seine Adresse) genommen wird. Nur Operanden OPS, OPT und OPE wird u. U. (maximal) ein ref vorangestellt.

Anmerkungen zu b und c:

1) Die vorangestellten val's und ref's sind im Rahmen der Befehlsinterpretation als mögliche Bestandteile der Operanden aufzufassen; sie treten in ZW3 erst durch die Aufspaltung der Befehle als selbständige Bestandteile in Erscheinung. Der (eigentliche) Operand ohne seine val's oder ref's ist es, welcher

durch seinen internal identifier codiert wird.

2) Lauf 3 hält bei der Analyse eines Ausdrucks die der Reihe nach anfallenden Teilausdrücke auf möglichst hoher Referenzstufe und dereferenziert ggfs. erst so spät wie möglich. Diese Strategie der maximalen Information erlaubt es, ohne Änderungen an Lauf 3 PAL dahingehend zu erweitern, daß man zur Indizierung alle Ausdrücke zuläßt, die Feldnamen als Ergebnis haben (und nicht nur Feldnamen selbst); außerdem könnte man als linke Seiten von Zuweisungen alle Ausdrücke zulassen, die Variable bzw. Zeiger als Ergebnis haben (und nicht nur Variable bzw. Zeiger selbst). Diese beiden Sprachänderungen würden nur für Lauf 2 eine (vertretbare) Belastung mit sich bringen.

Zu ZW3 sei generell noch angemerkt, daß sie eine zwar mosaikhafte, aber sehr informationsreiche Schnittstelle ist. Dies ist durchaus gewollt: Lauf 4 soll sich nach eigenem Ermessen die von ihm benötigte Information so beschaffen können, daß er möglichst wenig Arbeit hat. In sehr vielen Fällen benötigt Lauf 4 sogar die in ZW3 enthaltene Information in vollem Umfang, da er sie anders gar nicht mehr erschließen könnte.

Es wurde schon erwähnt, daß die jeweils relevant besetzten Hilfszellen nicht immer ein Intervall bilden. Dieser Zustand kommt allerdings nur in Verbindung mit Relationen vor (vgl. deren Syntax auf Seite 15). Das folgende Beispiel hat nicht nur die Aufgabe, die Wahrheit dieser Behauptung zu zeigen, sondern soll auch die Arbeitsweise des im Lauf 3 verwendeten Kellers illustrieren; die auftretenden PAL-Größen seien alles Variable vom Typ long und ◊ fungiere als fiktives Ausdruck-Endezeichen. Der zu übersetzende Ausdruck lautet: A+B < C+D > E+F = G+H ◊.

Zeile	Keller	jüngst gelesen	ZW3
1	A+B	<	HZ1 := A+B ;
2	HZ1 < C+D	>	HZ2 := C+D ;
3	HZ1 < HZ2	>	HZ1 := HZ1 < HZ2 save ;

Zeile	Keller	jüngst gelesen	ZW3
4	HZ1 ∧ HZ2 > E+F	=	HZ3 := E+F ;
5	HZ1 ∧ HZ2 > HZ3	=	HZ2 := HZ2 > HZ3 save ;
6	HZ1 ∧ HZ2 ∧ HZ3	=	HZ1 := HZ1 ∧ HZ2 ;
7	HZ1 ∧ HZ3 = G+H	◊	HZ2 := G+H ;
8	HZ1 ∧ HZ3 = HZ2	◊	HZ2 := HZ3 = HZ2 ;
9	HZ1 ∧ HZ2	◊	HZ1 := HZ1 ∧ HZ2 ;
10	HZ1	◊	

In Zeile 7 der Kellerspalte bilden die relevant besetzten Hilfszellen kein Intervall; in Zeile 6 sieht man, wie dieser Zustand zustande kommt: Es ist (ausnahmsweise) der nicht-oberste Kellerabschnitt HZ1 ∧ HZ2 zu reduzieren; dies ist erfreulicherweise die einzige derartige Situation; sie ist dadurch möglich, daß man den Relationsoperator vor HZ3 in ein logisches Und verwandeln mußte (vgl. die Festsetzungen auf Seite 16). Natürlich war dies nur ein einfaches Beispiel; wie kompliziert das Hilfszelleninterval bei beliebiger Klammerung und if-then-else-fi-Schachtelung zerreißen kann, wäre noch zu untersuchen. Damit, daß man jede Hilfszelle für sich verwaltet, ist man andererseits sicher aus dem Schneider. Im übrigen erkennt man, daß unter allen freien Hilfszellen stets diejenige mit kleinstem internal identifier genommen wird, um die Anzahl an benötigten Hilfszellen klein zu halten und ferner, daß Hilfszellen (jedenfalls bei dieser von uns gewählten Methode) keinen festen Typ aufweisen: HZ2 z. B. ist ab dem zweiten Befehl vom Typ long, ab dem fünften vom Typ bool, ab dem siebten vom Typ long, ab dem achten vom Typ bool und ab dem neunten Befehl irrelevant besetzt. Auch die Notwendigkeit von save ist hier sehr gut zu sehen: Zur Durchführung z. B. des vierten Befehls könnte (zur Runtime) eine garbage collection nötig werden; dazu ist es dann notwendig, zu wissen, daß HZ2 einen noch benötigten long-Wert enthält, der ja natürlicherweise nur im heap abgelegt werden kann. Das Besondere an save aber ist nur, wie schon gesagt, die Tatsache, daß eine Hilfszelle trotz Benutzung ihres Inhalts noch nicht frei geworden ist.

Nach dieser Einführung in die Arbeitsweise des Kellers behandeln wir diesen Keller nun genauer, wobei in Klammern angegebene Zahlen wieder die Codierung der entsprechenden Einträge angeben, welche wie stets von sonstigen Codierungen völlig unabhängig ist und auch sein soll. Einheitlich gilt, daß empty wieder durch 0 codiert wurde.

Der Keller des Lauf 3 besteht aus den fünf Spalten KINF, KPUSHED, KMODE, KTYPE und KKIND und kann Delimitereinträge (sog. Zustandseinträge) und Größeneinträge enthalten; dementsprechend gilt:

KINF[i] = delimiter (1) oder entity (2) .

Bei Delimitereinträgen ist nur noch KPUSHED relevant besetzt und gibt den gekellerten Delimiter an:

KPUSHED[i] = +u (1) bzw. -u (2) bzw. abs (3) bzw. sgp (4) bzw. sgn (5) bzw. sign (6) bzw. dig (7) bzw. ¬ (8) bzw. lwb (9) bzw. upb (10) bzw. ₁₀ (11) bzw. * (12) bzw. / (13) bzw. div (14) bzw. mod (15) bzw. +b (16) bzw. -b (17) bzw. < (18) bzw. > (19) bzw. ≤ (20) bzw. ≥ (21) bzw. = (22) bzw. ≠ (23) bzw. ∧ (24) bzw. ∨ (25) bzw. < (26) bzw. > (27) bzw. ≡ (28) bzw. & (29) bzw. ((30) bzw. [(31) bzw. , (32) bzw. if (33) bzw. then (34) bzw. else (35) bzw. .= (36) bzw. := (37) bzw. goto (38) bzw. labdef (39) bzw. inp (40) bzw. out (41) bzw. begin (42) .

Damit sind gleichzeitig auch diejenigen Delimiter aufgezählt, die überhaupt gekellert werden.

Für Größeneinträge werden alle fünf Spalten benötigt:

KPUSHED[i] = internal identifier der Größe, d. h. ein Verweis auf die Größe in IDLI (dies gilt, wie schon mehrfach angedeutet, auch für Hilfszellen)

KMODE[i] = normal (1) bzw. aux (2) (für Hilfszellen)

KTYPE[i] = bool (1) bzw. int (2) bzw. long (3) bzw. label (4) bzw. chain (5)

KKIND[i] = const (1) bzw. var (2) bzw. array (3) bzw. varptr (4)
bzw. arrptr (5) bzw. ptrvarptr (6) bzw. ptrarrptr (7)

Zu den beiden letztgenannten Möglichkeiten für KKIND sei angemerkt, daß sie nur bei Hilfszelleneinträgen und in Verbindung mit Alternativen auftreten, deren then- und else-Teil (blanke) Zeiger sind; sie bedeuten dann, daß die Hilfszelle (momentan) ein Zeiger auf einen Variablenzeiger bzw. auf einen Feldzeiger ist. Dementsprechend wird in ZW3 ein cond-Befehl abgesetzt, dessen Zuweisungszeichen der Delimiter := ptr ist. Noch höhere intermediäre Referenzstufen wurden nicht vorgesehen; außerdem gibt es grundsätzlich keine Hilfszellen, die Adressen von Hilfszellen zugewiesen erhalten.

Nun ist zur generellen Arbeitsweise des Lauf 3 noch etliches zu sagen.

PAL weist relativ viele Operatoren auf (vor allem relativ zur Anzahl derjenigen Delimiter, die in ZW2 sonst noch möglich sind) und ist überdies syntaktisch und semantisch einfach strukturiert. Dieser Sachverhalt legt es nahe, die Dekomposition in Befehle gemäß der Operatorpräzedenz-Methode vorzunehmen. Da sich auch die sonst noch vorhandenen Delimiter zwanglos in dieses Schema einfügen lassen und hier auch eingefügt werden, sollte man vielleicht besser von einer Delimiterpräzedenz-Methode sprechen; Ausnahmen von dieser Methode (es sind nur sehr wenige, z. B. die in Verbindung mit der Dekomposition von Relationen) werden durch explizites Abfragen ermittelt und geregelt.

In Anwendung dieser Methode ordnen wir jedem Delimiter bei seinem Einlesen aus ZW2 einen Lese- oder Zeichenrang zu; dabei gilt allgemein: Je größer der Leserang, desto größer die Wahrscheinlichkeit, daß reduziert wird (daß der Keller geleert wird). Für Operatoren folgt daraus: Je kleiner die Priorität (Vorrang) des Operators, desto größer sein Leserang. Es wurden folgende Leseränge vergeben:

Leserang	Delimiter mit diesem Leserang
1	begin
2	.= | := | goto | labdef | inp | out
3	(| [| if
4	+u | -u | abs | sgp | sgn | sign | dig | ¬
5	lwb | upb | ₁₀
6	* | / | div | mod
7	+b | -b
8	< | > | ≤ | ≥ | = | ≠
9	∧
10	∨
11	< | >
12	≡
13	&
14	) | , |] | then | else | fi
15	;
16	end

Ferner wird allen gekellerten Delimitern ein Keller- oder Zustandsrang zugeordnet; dieser wird gerade so gewählt, daß das Reduzieren des Kellers in Verbindung mit dem Zeichenrang möglichst ausnahmslos funktioniert: Im wesentlichen wird der Keller solange reduziert, solange der Kellerrang des momentan obersten gekellerten Delimiters nicht (echt) größer ist als der Leserang des jüngst gelesenen Delimiters, aber keinesfalls weiter als bis einschließlich der dem jüngst gelesenen Delimiter entsprechenden (im Keller befindlichen) öffnenden Klammer (sofern vorhanden); die letztgenannte Beschränkung kann dabei nur dann aktuell sein, wenn der Kellerrang des obersten Kellerdelimiters gleich dem Leserang des jüngst gelesenen Delimiters ist; nur in diesem Gleichheitsfall treten auch sonstige Besonderheiten auf, denn es wurden folgende Kellerränge vergeben:

Kellerrang	Delimiter mit diesem Kellerrang
4	+u | -u | abs | sgp | sgn | sign | dig | ¬
5	lwb | upb | ↑
6	* | / | div | mod
7	+b | -b
8	< | > | ≤ | ≥ | = | ≠
9	∧
10	∨
11	< | >
12	≡
13	&
14	(| [| , | if | then | else
15	.= | := | goto | labdef | inp | out
16	begin

Hieran sieht man zunächst, daß es Kellerränge < 4 nicht gibt, woraus folgt, daß man alle Delimiter mit Leserang < 4 sogleich nach ihrem Einlesen kellern wird. Ferner sieht man, daß genau die unären Operatoren den Lese- und Kellerrang 4 haben; auf Grund der in PAL gewählten Präfixnotation für unäre Operatoren gilt dann, daß man auch unäre Operatoren nach ihrem Lesen sofort kellern wird. Mit dieser Bemerkung ist gleichzeitig eine der "sonstigen Besonderheiten" genannt, die im Gleichheitsfall Kellerrang = Leserang zu berücksichtigen sind. Im übrigen sei gleich hier noch erwähnt, daß die syntaktischen Kontrollen des Lauf 2 sicherstellen, daß im Fall der Notwendigkeit einer Reduktion stets der oberste Kellereintrag eine Größe und der zweitoberste Kellereintrag den obersten Kellerdelimiter enthalten, woraus folgt, daß sicher keine Reduktion nötig ist, falls der oberste Kellereintrag ein Delimiter ist.

Im folgenden wird Lauf 3 grob skizziert. Den Löwenanteil an Lauf 3 bildet (statisch wie dynamisch) die Prozedur CLEAROP ("clear ope-

rator"), die man wohl besser CLEARDEL ("clear delimiter") genannt hätte; sie führt sämtliche Reduktionen durch und damit auch die Generierung sämtlicher Befehle für ZW3 sowie fast die gesamte semantische Kontrolle. Da der jüngst eingelesene Delimiter eine ganze Kette von aufeinanderfolgenden Reduktionen auslösen kann, wurde der Rumpf von CLEAROP zu einer prinzipiell unendlichen Schleife gemacht; CLEAROP entscheidet völlig selbständig, wann es diese Schleife verlassen muß (wegen der voraussetzbaren syntaktischen Korrektheit wird die Schleife immer verlassen), sei es, daß der oberste Kellereintrag ein Delimiter ist (z. B. ein begin), sei es, daß der Kellerrang größer als der Leserang geworden ist oder sei es, daß der nunmehr oberste Kellerdelimiter von sich aus signalisiert, daß (meist nach einer Reihe von abschließenden Tätigkeiten) CLEAROP zu verlassen ist. Der letztgenannte Fall tritt auf, wenn entweder gewisse Handlungsteile (geklammerte Ausdrücke, indizierte Variable oder Bedingungsteile) erschöpft sind (oberster Kellerdelimiter: Runde Klammer auf bzw. eckige Klammer auf bzw. if bzw. then bzw. else) oder die gesamte Handlung zu Ende geht (oberster Kellerdelimiter: Zuweisungszeichen bzw. goto bzw. labdef bzw. inp bzw. out). Das bisher Gesagte legt schon nahe, daß die Schleife von CLEAROP im wesentlichen darin besteht, in Abhängigkeit des obersten Kellerdelimiters die entsprechende Marke eines Verteilers anzuspringen. Im Fall eines Operators gilt dann (sogar genau dann), daß nach dem Aufruf einer von vier zu CLEAROP lokalen Prozeduren (viele Operatoren verhalten sich sehr gleichartig) an den Schleifenanfang zurückgesprungen wird; dort wird die Schleife dann möglicherweise sofort verlassen. Zu den innerhalb von CLEAROP durchgeführten semantischen Kontrollen sei angemerkt, daß die Kontrolle auf zulässige Typen und Arten der von einer Reduktion betroffenen Größen (vgl. die Kellerspalten KTYPE und KKIND) mit Hilfe zweier passend vorbesetzter logischer Matrizen UNZULTYPE und UNZULKIND in Abhängigkeit des obersten Kellerdelimiters durchgeführt wird; dabei ist je Typ bzw. Art eine Spalte vorhanden und je Delimiter bis zu zwei Zeilen, wobei gleiche Zeilen aber nur einmal auftreten. Bei else und := als obersten Kellerdelimitern wird noch eine weitere logische Matrix PASSKIND herangezogen, die zu entscheiden gestattet, ob eine Art

durch Dereferenzierungen in eine andere Art überführbar ist. Alle sonstigen semantischen Kontrollen wie z. B. die auf korrekte Referenzstufen oder die auf bei den meisten binären Operatoren erforderliche Typgleichheit werden in CLEAROP explizit an Ort und Stelle durchgeführt, nur die Kontrolle auf Deklariertheit von Identifikatoren bzw. Definiertheit von Marken findet nicht in CLEAROP statt.

CLEAROP wird im Rahmen der Hauptschleife des Lauf 3 angesprungen:

```
LESEN:
     READ( INPCOMP );
     goto if INPCOMP < 0 then KASK[ - INPCOMP ] else ENTITY;
CLEAR AND PUSH:
     CLEAROP;
PUSH DELIMITER:
     PUSHDEL( ZW2KL[ INPCOMP ] );
     goto LESEN;
CLEAR ONLY:
     CLEAROP;
     goto LESEN;
ENTITY:
     :
     goto LESEN;
BEGIN:
     :
     goto LESEN;
END:
     :
     goto LESEN;
```

Zunächst wird also der nächste ZW2-Eintrag gelesen und in INPCOMP abgelegt; Zeilenangaben werden wie immer innerhalb von READ abge-

arbeitet. Enthält INPCOMP eine Bezeichnung, so wird die Marke ENTITY, andernfalls der Verteiler KASK angesprungen. Letzterer ordnet jedem Delimiter eine anzuspringende Marke zu, nämlich CLEAR AND PUSH, PUSH DELIMITER oder CLEAR ONLY. Nur im Fall eines begin's oder end's wird zu den weiter unten angedeuteten gleichnamigen Marken gesprungen. Der aktuelle Parameter von PUSHDEL deutet die Notwendigkeit der Umcodierung der Delimiter an.

Bei der Marke ENTITY sucht die Prozedur SEARCH ENTITY die eben gelesene Bezeichnung in IDLI und liefert den entsprechenden internal identifier als Resultat ab; damit ist automatisch die Kontrolle auf Deklariertheit bzw. Definiertheit der Größe verbunden. Anschließend wird die Größe noch gekellert.

Bei der Marke BEGIN wird ein begin gekellert und die beiden Zeiger D und DR aktualisiert, die SEARCH ENTITY zu ihrer Bezeichnungssuche in IDLI braucht; dabei zeigt D auf das dem jüngst gelesenen Struktursymbol entsprechende Struktursymbol in IDLI und DR auf das begin des momentan aktuellen Blocks in IDLI.

Bei der Marke END wird der oberste Kellereintrag gelöscht (der mit Sicherheit ein begin ist) und die beiden Zeiger D und DR aktualisiert. Zeigt D nun auf das letzte end in IDLI, so ist Lauf 3 zu beenden, andernfalls wird nach LESEN zurückgesprungen.

Abschließend seien noch ein paar Worte zur Fehlerbehandlung des Lauf 3 gesagt. Generell gilt wieder, daß im Fehlerfall der Fehlerort in Form einer Zeilenangabe mit ausgegeben wird. Im Gegensatz zu Lauf 1 und 2 kann es aber jetzt durchaus vorkommen, daß die Zeilenangabe zu groß ausfällt: Bis ein Delimiter wieder aus dem Keller entfernt wird, können schon viele weitere Zeilenangaben gelesen worden sein. Dem wäre relativ einfach abzuhelfen: Man erweitere den Keller um eine Spalte KZEILE, die nur für Delimiter definiert ist und dann die Protokollzeile des Delimiters angibt. Ergibt sich beim Entfernen des Delimiters aus dem Keller ein Fehler, so gebe man als Fehlerort die gekellerte Zeilenangabe an. Letztere läßt sich in analoger Weise dann auch noch dazu verwen-

den, die Zeilenangaben in ZW3 genauer zu machen: Man stelle jedem Befehl eine Zeilenangabe voran, wobei man insgesamt jedoch möglichst wenige Zeilenangaben absetze. Speziell bei indizierten Größen und Bedingungen nehme man als Zeilenangabe die Zeile der öffnenden Indexklammer bzw. des if.

Fehler mit sofortigem Abbruch sind:

1) Überlauf der Identifikatorenliste IDLI. Meldung: "Abbruch in Zeile x: Identifikatorenliste für Lauf 3 zu klein".
2) Überlauf des Kellers. Meldung: "Abbruch in Zeile x: Keller für Lauf 3 zu klein".

Die sonstigen Fehler (sie gelten alle als Fehlersituationen) werden alle in Klartext gemeldet. Zur Vermeidung von Folgefehlermeldungen wurden der Typ empty und die Art (kind) empty eingeführt, wobei dieser Typ bzw. diese Art bei allen entsprechenden Abfragen und insbesondere bei den Matrizen UNZULTYPE, UNZULKIND und PASSKIND immer zulässig sind bzw. passen. Ein nichtdeklarierter Identifikator oder eine nichtdefinierte Marke z. B. besitzen diesen Typ und diese Art (und den internal identifier 0). Im übrigen wird dieser Typ und diese Art auf Grund von Kontextzusammenhängen möglichst bald wieder aus dem Keller entfernt. Eine Division / z. B., bei der der linke Operand vom Typ empty und der rechte Operand vom Typ long ist, wird reduziert zu einem Ergebnis des Typs long.

Im Korrektheitsfall andererseits verabschiedet sich Lauf 3 mit der Meldung "Lauf 3 ohne Fehler beendet".

V. Lauf 4

Die Hauptaufgabe von Lauf 4 ist die eigentliche Codegenerierung, d. h. die Befehle der ZW3 in korrekte und gemäß der Befehlsinterpretation wirkungsgleiche BCPL-Anweisungsfolgen umzuformen und diese als das Zielprogramm ZW4 in eine Texthaltungsdatei abzusetzen. Die in ZW3 auftretenden Bezeichnungen sind dabei in BCPL-Namen umzuformen. Für alle so vorhandenen BCPL-Größen sind im "Kopf" (Anfangsteil) des Zielprogramms passend initialisierte Deklarationen zu generieren, wobei die Konstanten so behandelt werden, wie es für die Befehlsinterpretation vereinbart wurde. Im Kopf sind auch die Deklarationen aller vorhandenen (eigentlich nur: aller aktuell benötigten) Runtime-Unterprogramme aufzulisten sowie die initialisierten Deklarationen der von ihnen benötigten nichtlokalen BCPL-Größen. Ferner sind all diejenigen Kontrollen explizit zu generieren, die zur Runtime durchgeführt werden müssen und nicht innerhalb von Runtime-Unterprogrammen durchgeführt werden. Hierunter fallen z. B. alle Definiertheitskontrollen. Schließlich sind im Zusammenhang mit Hilfszellen BCPL-Anweisungen zu generieren, die diejenige Information auf dem laufenden halten, welche die garbage collection bezüglich Hilfszellen benötigt. All diese Aufgaben werden im folgenden näher ausgeführt. Dabei wird für den ganzen Rest dieser Arbeit die Bekanntheit mit der Programmiersprache BCPL vorausgesetzt. Dabei sei sogleich auf eine Besonderheit der BCPL-Implementierung auf dem TR440 hingewiesen: Der TR440 und leider auch die TR440-BCPL unterscheiden zwischen einer negativen Null (alle Bits gesetzt) und einer positiven Null (alle Bits gelöscht) und es gilt: $-1 < -0 < +0 < +1$.
In der Regel kann man diese Tatsache unberücksichtigt lassen; nur Vergleiche muß man i. a. entsprechend umständlich formulieren:

Vergleich	Formulierung
Z EQ 0	-1 LS Z LS 1
Z NE 0	Z GE 1 LOGOR Z LE -1
Z LS 0	Z LE -1
Z GR 0	Z GE 1

Vergleich	Formulierung
Z LE 0	Z LS 1
Z GE 0	Z GR -1

Allgemeine Vergleiche X δ Y sind i. a. durch X - Y δ 0 darzustellen (wobei auf X - Y δ 0 noch obige Tabelle anzuwenden ist). Dies ist jedoch nur nötig, wenn X und Y gleichzeitig gleich 0 sein können; andernfalls arbeitet der direkte Vergleich X δ Y auch korrekt.

Die aus den in ZW3 auftretenden Bezeichnungen (den internal identifiern) zu bildenden BCPL-Namen haben folgendes Aussehen: X.Y.i. X und Y geben Art und Typ der Bezeichnung gemäß den Informationen in IDLI an. Im einzelnen gilt für X:

X = K für Konstante
V für Variable
A für Felder (arrays)
P.V für Zeiger (pointer) auf Variable
P.A für Zeiger auf Felder
HZ für Hilfszellen

Für Y gilt:

Y = B für Typ bool
I für Typ int
L für Typ long
M für Typ label (Marke)
C für Typ chain

Für Hilfszellen jedoch ist Y und der nachfolgende Punkt wegzulassen, da Hilfszellen keinen festen Typ besitzen (jedenfalls nicht bei der von uns gewählten Vorgehensweise des Lauf 3).

i schließlich ist der in ZW3 auftretende internal identifier genau dann, wenn der zur Bezeichnung gehörige Bezugseintrag in IDLI undef enthält; andernfalls ist für i gerade dieser Bezugseintrag zu nehmen. Der Bezugseintrag free von Hilfszellen ist in diesem Zusammenhang stets wie undef zu werten.

Die Erzeugung von BCPL-Namen ist Bestandteil der Umformung der Befehlsoperanden. Zu diesen gehören der Befehlsinterpretation entsprechend auch noch die ggfs. vorangestellten ref's und val's. Für deren Umformung gilt:

1) Jedes ref ist in den unären BCPL-Operator LV umzusetzen.

2) Jedes val ist in den unären BCPL-Operator RV umzusetzen.

Eine wirklich vollständige Runtime-Überwachung kann auf Definiertheitskontrollen nicht verzichten; deshalb soll Lauf 4 diese Kontrollen generieren, wenn auch das Zielprogramm dadurch nicht unwesentlich belastet wird. Die Belastung rührt daher, daß das BCPL-Manual (mindestens das des TR440) keine definierte Reaktion auf die Benutzung von undefinierten Werten festlegt. So müssen alle entsprechenden Abfragen explizit generiert werden. Dazu braucht man ein Bitmuster, welches in BCPL-Variablen von übersetzten PAL-Programmen (sonst) nicht auftreten kann. Im Kopf des Zielprogramms findet sich die folgende MANIFEST-Deklaration, die ein solches Bitmuster mit U ("undefiniert") benennt:

```
MANIFEST BEGIN U = $H800000 END
```

Weiter ist es möglich, einer BCPL-Variablen V dieses Bitmuster zuzuweisen (V := U) sowie durch eine Gleichheitsabfrage zu prüfen, ob V das Bitmuster U enthält (... V EQ U ...). Letzteres ist möglich, da bei Gleichheitsabfragen direkt auf Bitmuster-Identität abgefragt wird. So können wir die Benutzung von undefinierten Werten auf definierte Weise abfangen: Lauf 4 generiert ein mit AL ("Alarm") markiertes Programmstück, welches die Meldung "Abbruch in Zeile x: Undefinierte Größe unzulässig" ausgibt und das Zielprogramm beendet. Im übrigen werden alle zu prüfenden BCPL-Variablen (und nicht nur sie) bei ihrer Entstehung zur Laufzeit oder ihrer Deklaration mit U initialisiert. Die Belastung des Zielpro-

gramms rührt von den vielen zu generierenden Definiertheitsabfragen her:

IF <Operand> EQ U DO GOTO AL;

Die Runtime-Darstellungen aller Operanden wurde andererseits so festgelegt, daß alle Operanden auf diese einfache Weise auf Definiertheit geprüft werden können.

Genau die expliziten und impliziten Dereferenzierungen sind es, welche durch solche Anweisungen zu überwachen sind (bei mehreren gleichzeitigen Dereferenzierungen sind zuerst die expliziten und dann die implizite ausgeführt zu denken). Die Definition von PAL (vgl. Seite 22) erzwingt allerdings eine (einzige) Ausnahme: Im Zuweisungsbefehl (OPD := OPS ;) darf die implizite Dereferenzierung von OPS nicht überwacht werden. Da dieser Befehl nur in Verbindung mit Zuweisungszeichen des Quellprogramms generiert wird, folgt, daß OPD letztlich eine PAL-Größe und damit keine Hilfszelle bezeichnet. Damit folgt, daß der Inhalt von Hilfszellen stets definiert ist und deshalb bei seiner Benutzung nicht geprüft zu werden braucht; dasselbe gilt a forteriori natürlich auch für alle BCPL-Variablen, die aus PAL-Konstanten erzeugt wurden.

Beispiel: Seien P, Q und R Zeiger auf Variable vom Typ int mit den internal identifiern i, j und k; sei B eine Variable vom Typ bool mit internal identifier l, besitze 1 den internal identifier m und sei n der internal identifier einer Hilfszelle. Die PAL-Anweisung

P .= if B then Q else R fi

wird von Lauf 3 übersetzt in die beiden Befehle

n := ptr cond l ref j ref k ;
val i := int val val n ;

Lauf 4 hat dann zu generieren:

```
IF V.B.1 EQ U DO GOTO AL;
HZ.n := V.B.1 COND LV P.V.I.j , LV P.V.I.k;
IF P.V.I.i EQ U DO GOTO AL;
IF RV HZ.n EQ U DO GOTO AL;
```

```
RV P.V.I.i := RV RV HZ.n;
```

Die PAL-Anweisung

```
P .= if B then Q else R fi + 1
```

wird von Lauf 3 übersetzt in die drei Befehle

```
n := ptr cond 1 ref j ref k ;
n := int val val n + b int m ;
val i := int n ;
```

Lauf 4 hat dann zu generieren:

```
IF V.B.1 EQ U DO GOTO AL;
HZ.n := V.B.1 COND LV P.V.I.j , LV P.V.I.k;
IF RV HZ.n EQ U DO GOTO AL;
IF RV RV HZ.n EQ U DO GOTO AL;
HZ.n := RV RV HZ.n + K.I.m;
IF P.V.I.i EQ U DO GOTO AL;
RV P.V.I.i := HZ.n;
```

Auf die Benutzung von undefinierten Werten durch den PAL-Programmierer könnte man natürlich auch wesentlich anders reagieren; die Definition von PAL schreibt in dieser Hinsicht ja nichts vor. Es sei deshalb im folgenden ein zum geschilderten Vorgehen genau entgegengesetztes kurz skizziert, ohne allerdings letzteres unbedingt empfehlen zu wollen.

goto U ;
: Es wird nicht gesprungen, sondern mit dem nächsten Befehl fortgefahren

labdef U ;
: Kann nicht auftreten

inp U ;
: Der Eingabezeiger überspringt das als nächstes anstehende Eingabedatum, sofern dieses existiert; andernfalls erfolgt wie bei inp OP mit definiertem OP ein Fehlerabbruch mangels Eingabedaten

out U ;

Es wird ein bis jetzt noch nicht vorhandenes PAL-Zeichen Ø ausgegeben; Ø könnte etwa den PAL-Wert 9 haben

Befehle, die unop, binop, cond oder indvar aufweisen:

HZ erhält U zugewiesen; daraus folgt, daß bei diesem Vorgehen auch Hilfszellen auf Definiertheit abzufragen sind

U := OPS ;

Es wird nichts zugewiesen, sondern sogleich mit dem nächsten Befehl fortgefahren

OPD := U ; mit OPD ≠ U

OPD wird U zugewiesen

dim DIM U ;

Es wird keine Dimension kontrolliert, sondern sogleich mit dem nächsten Befehl fortgefahren

Man sieht, daß bei diesem Vorgehen das Programm in jedem Fall in definierter Weise fortgesetzt wird; der PAL-Programmierer kann damit eine Reihe von zusätzlichen Effekten erreichen. Im übrigen ist er ohnehin schon in der Lage, PAL-Variable und -Zeiger auf "undefiniert" zu setzen; auf "undefiniert" abfragen kann er im Augenblick aber noch nicht. Wollte man das ermöglichen, würde man wohl am besten einen zusätzlichen unären Operator undef einführen, der auf alle Operanden angewendet werden darf und je nach Fall true oder false als Ergebnis liefert.

Damit können wir uns der Transformation der Befehle selbst zuwenden. Die Umformung der Operanden einschließlich der nötigen Definiertheitsprüfungen ist geklärt und wird deshalb im folgenden weggelassen.

Vorweg noch sei die Umformung von Zeilenangaben behandelt. Für jede Zeilenangabe ist eine Wertzuweisung an die BCPL-Variable Z ("Zeile") zu erzeugen, deren rechte Seite die betreffende Zeilennummer direkt angibt. Auf diese Weise sind zur Runtime das mit AL markierte Programmstück sowie die sonstigen Fehlerreaktionen in der Lage, bei ihren Meldungen die Fehlerzeile anzugeben: Sie ist der momentane Wert von Z.

Zur Transformation der Befehle werden manchmal Hilfszellen H1, H2, ... herangezogen. Sie sind im Kopf des Zielprogramms zu deklarieren und sind im übrigen immer nur sehr kurzfristig relevant besetzt. Mit den von Lauf 3 generierten Hilfszellen haben sie nichts zu tun.

Einadreßbefehle:

goto OP ;

```
    GOTO OP EQ U COND AL , OP;
```

labdef KO ;

```
    KO:
```

inp <type> OP ;

```
    H1 := INP<TYPE>(); OP := H1;
```

out <type> OP ;

```
    OUT<TYPE>( OP );
```

Zweiadreßbefehle:

HZ := unop OP ;

unop	Transformation
+ u int	HZ := + OP;
+ u long	HZ := UNPLUS(OP);
- u int	HZ := - OP;
- u long	HZ := UNMINUS(OP);
abs int	HZ := OP LS 0 COND - OP , OP;
abs long	HZ := ABS(OP);
sgp int	HZ := OP LE -1 COND -1 , 1;
sgp long	HZ := SGP(OP);
sgn int	HZ := OP GE 1 COND 1 , -1;
sgn long	HZ := SGN(OP);
sign int	HZ := OP GE 1 COND 1 , OP LE -1 COND -1 , 0;
sign long	HZ := SIGN(OP);

unop	Transformation
dig int	H1 := OP LS 0 COND - OP , OP; HZ := 0; BEGIN H1 := H1 / 10; HZ := HZ + 1 END REPEATUNTIL H1 LS 1;
dig long	HZ := OP ! 2; Dazu ist anzumerken, daß Variable (auch alle Konstanten werden wie Variable behandelt) vom Typ long durch BCPL-Vektoren realisiert werden; der internal identifier der Variablen bzw. dessen BCPL-Äquivalent fungiert als Bezeichnung für die Leitzelle des Vektors (sie enthält die physikalische Anfangsadresse des eigentlichen Vektors und wird mit U initialisiert). Mit OP!2 wird folglich die dritte Komponente des eigentlichen Vektors angesteuert (womit eine zu kontrollierende implizite Dereferenzierung verbunden ist); diese Komponente enthält die Anzahl der Ziffern des long-Werts gerade in der Weise, wie sie für dig benötigt wird
¬	HZ := NOT OP;

OPD := OPS ;
OPD := OPS;

dim DIM OP ;
IF DIM NE RV OP DO GOTO AD;
Dazu ist anzumerken:

1) AD ("Alarm Dimension") markiert ein (von Lauf 4 zu generierendes) Programmstück, welches die Meldung "Abbruch in Zeile x: Falsche Anzahl von Indizes" ausgibt und das Zielprogramm beendet.
2) Der Informationsvektor eines Feldes wurde mit Hilfe eines BCPL-Vektors realisiert; dessen Leitzelle ist gerade die schon erwähnte "Feldzelle" des Feldes; der internal identifier des Feldes bzw. sein BCPL-Äquivalent fungiert als Bezeichnung für die Feldzelle.
   Ein Informationsvektor enthält der Reihe nach: Dimension n

des Feldes; physikalische Anfangsadresse des Feldes (mit U vorbesetzt; das Feld selbst liegt in der Halde); die n momentanen unteren Indexgrenzen (mit 0 vorbesetzt); die n momentanen Kantenlängen (mit 0 vorbesetzt). Logisch gesehen müßte also generiert werden: IF DIM NE OP ! 0 ... . Das wirkungsgleiche "RV OP" ist jedoch schneller. Im übrigen sind die Feldzelle und alle Zellen eines Informationsvektors stets wohldefiniert; lediglich OP!1 enthält solange das Bitmuster U, solange das Feld noch nicht aufgerufen wurde und deshalb noch nicht in der Halde existiert.

Dreiadreßbefehle:

HZ := OPL binop OPR [save] ;

binop	Transformation
lwb	IF OPL LS 1 LOGOR OPL GR RV OPR DO GOTO ALB1; IF OPR ! 1 EQ U DO GOTO ALB2; HZ := OPR ! (1 + OPL); Dazu ist anzumerken: 1) ALB1 ("Alarm lower bound 1") markiert ein Programmstück, welches die Meldung "Abbruch in Zeile x: Untere Feldgrenze existiert nicht" ausgibt und das Zielprogramm beendet. 2) ALB2 markiert ein Programmstück, welches die Meldung "Abbruch in Zeile x: Untere Feldgrenze undefiniert" ausgibt und das Zielprogramm beendet. 3) Siehe Anmerkung 2 zum dim-Befehl
upb	IF OPL LS 1 LOGOR OPL GR RV OPR DO GOTO AUB1; IF OPR ! 1 EQ U DO GOTO AUB2; HZ := OPR ! (1 + OPL) + OPR ! (1 + OPL + RV OPR) - 1; Dazu ist anzumerken: 1) AUB1 ("Alarm upper bound 1") markiert ein Programmstück, welches die Meldung "Abbruch in Zeile x: Obere Feldgrenze existiert nicht" ausgibt und das Zielprogramm beendet.

binop	Transformation
	2) AUB2 markiert ein Programmstück, welches die Meldung "Abbruch in Zeile x: Obere Feldgrenze undefiniert" ausgibt und das Zielprogramm beendet. 3) Siehe Anmerkung 2 zum dim-Befehl
10 int	H1 := OPR LS 0 COND -1 , 1; H2 := 1; FOR I = 1 TO H1 * OPR DO H2 := H2 * 10; H3 := OPL LS 0 COND -1 , 1; HZ := H1 LS 0 COND OPL * H3 / H2 * H3 , OPL * H2;
10 long	HZ := SCALE(OPL , OPR);
* int	HZ := OPL * OPR;
* long	HZ := MULT(OPL , OPR);
/ int	IF -1 LS OPR LS 1 DO GOTO ADU; H1 := OPL / OPR; H2 := OPL - H1 * OPR; H3 := H2 LS 0 COND -1 , 1; H4 := H3 * H2 * 2; H5 := OPR LS 0 COND -1 , 1; H6 := H5 * OPR; HZ := H4 LS H6 LOGOR H4 EQ H6 LOGAND H1 / 2 * 2 + 1 EQ H1 + 1 COND H1 , H1 + H3 * H5; Dazu ist anzumerken, daß ADU ("Alarm bei / (durch)") ein Programmstück markiert, welches die Meldung "Abbruch in Zeile x: Division / durch 0" ausgibt und das Zielprogramm beendet.
/ long	HZ := DURCH(OPL , OPR);
div int	IF -1 LS OPR LS 1 DO GOTO ADI; H1 := OPL / OPR; H2 := OPL - H1 * OPR; H3 := OPR LS 0 COND -1 , 1; H4 := H2 GE 1 COND 1 , H2 LE -1 COND -1 , H3; HZ := H4 EQ H3 COND H1 , H1 - 1; Dazu ist anzumerken, daß ADI ("Alarm bei div") ein Programmstück markiert, welches die Meldung "Abbruch in Zeile x: Division div durch 0" ausgibt und das Zielprogramm beendet.

binop	Transformation
div long	HZ := DIV(OPL , OPR);
mod int	HZ := -1 LS OPR LS 1 COND OPL , VALOF BEGIN H1 := OPL / OPR; H2 := OPL - H1 * OPR; H3 := OPR LS 0 COND -1 , 1; H4 := H2 GE 1 COND 1 , H2 LE -1 COND -1 , H3; RESULTIS H4 EQ H3 COND H2 , H2 + OPR END;
mod long	HZ := MOD(OPL , OPR);
+ b int	HZ := OPL + OPR;
+ b long	HZ := PLUS(OPL , OPR);
- b int	HZ := OPL - OPR;
- b long	HZ := MINUS(OPL , OPR);
< int	HZ := OPL - OPR LE -1;
< long	HZ := LESS(OPL , OPR);
> int	HZ := OPL - OPR GE 1;
> long	HZ := GRT(OPL , OPR);
≤ int	HZ := OPL - OPR LS 1;
≤ long	HZ := LEEQ(OPL , OPR);
≥ int	HZ := OPL - OPR GR -1;
≥ long	HZ := GREQ(OPL , OPR);
= int	HZ := -1 LS OPL - OPR LS 1;
= long	HZ := EQL(OPL , OPR);
≠ int	HZ := OPL - OPR GE 1 LOGOR OPL - OPR LE -1;
≠ long	HZ := NEQ(OPL , OPR);
∧	HZ := OPL LOGAND OPR;
∨	HZ := OPL LOGOR OPR;

binop	Transformation
<	HZ := OPL LOGOR NOT OPR;
>	HZ := NOT OPL LOGOR OPR;
≡	HZ := OPL EQV OPR;
&	HZ := CONC(OPL , OPR);

Vieradreßbefehle:

```
HZ := cond BED OPT OPE ;
    HZ := BED COND OPT , OPE;
```

Multiadreßbefehle:

```
HZ := indvar OP0 OP1 ... OPn ;
    S ! 0 := OP0; H1 := LV OP0 ! 1;
    S ! 1 := OP1 - H1 ! 1;
          :
    S ! n := OPn - H1 ! n;
    HZ := IV();
```

Dazu ist anzumerken, daß die parameterlose Funktion IV ("indizierte Variable") die Adresse der aufgerufenen indizierten Variablen berechnet und ggfs. eine Feldvergrößerung durchführt; der Vektor S, der mit KOLI[52] + 1 Elementen zu vereinbaren ist, enthält mit der Anfangsadresse des betreffenden Informationsvektors und den reduzierten Indizes alle dazu nötigen Informationen.

Zu den Befehlen sei generell noch angemerkt, daß es wünschenswert ist, unter Heranziehung von weiteren Hilfszellen H7, H8 ... möglichst viele explizite Dereferenzierungen der Operanden zu sparen, dies insbesondere in Verbindung mit den Definiertheitskontrollen der Operanden, auch wenn die Codeerzeugung dadurch etwas erschwert wird.

Nun ist es zweckmäßig, den Kopf des Zielprogramms zu skizzieren. Die hier gewählte Reihenfolge der einzelnen Deklarationen ist da-

bei völlig unerheblich, soweit nicht ausdrücklich BCPL-Konventionen dagegen stehen. Der genaue Aufbau von angedeuteten Zahlenlisten

<integer> , ... , <integer>

wird für jeden einzelnen Fall im nächsten Kapitel erläutert.

```
BEGIN
NONREC 1 : <E/A-Prozedur> , ... , <E/A-Prozedur>
EXTERNAL BL.EA : <E/A-Prozedur> , ... , <E/A-Prozedur>
GLOBAL BEGIN START : 1 END
MANIFEST BEGIN U = $H800000 END
STATIC BEGIN
   K.B.2 = TRUE;
   K.B.3 = FALSE;
   :
   K.I.i = <integer>;
   :
   K.L.i = TABLE U, U, <integer>, ..., <integer>;
   :
   Bei Markenkonstanten ist hier nichts explizit zu tun; sie wer-
   den vom BCPL-Compiler selbst in analoger Weise behandelt
   :
   K.C.i = TABLE U, U, <integer>, ..., <integer>;
   :
   V.B.i = U;
   :
   V.I.i = U;
   :
   V.L.i = U;
   :
   V.M.i = U;
```

```
:
V.C.i = U;
:
A.B.i = TABLE <dimension>, U, 0 REP <2*dimension>;
:
A.I.i =                          "
:
A.L.i =                          "
:
A.M.i =                          "
:
A.C.i =                          "
:
P.V.B.i = U;
:
P.V.I.i = U;
:
P.V.L.i = U;
:
P.V.M.i = U;
:
P.V.C.i = U;
:
P.A.B.i = U;
:
P.A.I.i = U;
:
P.A.L.i = U;
```

```
    :
    P.A.M.i = U;
    :
    P.A.C.i = U;
    :
    HZ.i = U;
    :
    Z = U;
    HEAP = TABLE U REP <heap-Größe>;
    . } Alle nichtlokalen Größen, die von Runtime-Unterprogrammen
    : } benötigt werden (z. B. von der garbage collection GC)
END
LET GC( FW , FN , F1 , F2 ) BE BEGIN ... END
LET INPBOOL() = VALOF BEGIN ... END
LET INPINT() = VALOF BEGIN ... END
LET INPLONG() = VALOF BEGIN ... END
LET INPCHAIN() = VALOF BEGIN ... END
LET OUTBOOL( FP ) BE BEGIN ... END
LET OUTINT( FP ) BE BEGIN ... END
LET OUTLONG( FP ) BE BEGIN ... END
LET OUTCHAIN( FP ) BE BEGIN ... END
LET UNPLUS( FP ) = VALOF BEGIN ... END
LET UNMINUS( FP ) = VALOF BEGIN ... END
LET ABS( FP ) = VALOF BEGIN ... END
LET SGP( FP ) = VALOF BEGIN ... END
LET SGN( FP ) = VALOF BEGIN ... END
LET SIGN( FP ) = VALOF BEGIN ... END
LET SCALE( FL , FR ) = VALOF BEGIN ... END
LET MULT( FL , FR ) = VALOF BEGIN ... END
LET DURCH( FL , FR ) = VALOF BEGIN ... END
LET DIV( FL , FR ) = VALOF BEGIN ... END
LET MOD( FL , FR ) = VALOF BEGIN ... END
LET PLUS( FL , FR ) = VALOF BEGIN ... END
LET MINUS( FL , FR ) = VALOF BEGIN ... END
LET LESS( FL , FR ) = VALOF BEGIN ... END
```

```
LET GRT( FL , FR ) = VALOF BEGIN ... END
LET LEEQ( FL , FR ) = VALOF BEGIN ... END
LET GREQ( FL , FR ) = VALOF BEGIN ... END
LET EQL( FL , FR ) = VALOF BEGIN ... END
LET NEQ( FL , FR ) = VALOF BEGIN ... END
LET CONC( FL , FR ) = VALOF BEGIN ... END
LET IV() = VALOF BEGIN ... END
START: .  } Vorbesetzungen, die erst zur Laufzeit des Zielprogramms
       .  } errechenbar sind, z. B. solche für die garbage collec-
       .  } tion GC
```

In den ersten beiden Deklarationen sind diejenigen E/A-Prozeduren aufzuzählen, die man in den Runtime-Unterprogrammen verwendet. Die Aufzählung der Runtime-Unterprogramme ist erschöpfend, es sei denn, man ruft in diesen Unterprogrammen außer den E/A-Prozeduren und GC weitere Funktionen und/oder Routinen auf, die man parallel zu ihnen vereinbaren will.

In praxi wird man letztlich wohl nicht so vorgehen, wie es soeben skizziert wurde, sondern man wird den Deklarationen der Runtime-Unterprogramme die fünf Deklarationen

```
NONREC 1 : <E/A-Prozedur>, ..., <E/A-Prozedur>
EXTERNAL BL.EA : <E/A-Prozedur>, ..., <E/A-Prozedur>
EXTERNAL INPBOOL, INPINT, ..., CONC, IV
EXTERNAL Z, HEAP, <Liste der nichtlokalen Größen der Runtime-
        Unterprogramme>
MANIFEST BEGIN U = $H800000 END
```

voranstellen und sie einmal vorweg übersetzen, wodurch ein entsprechendes Montageobjekt MOUPPAL entsteht. Im Kopf sind dann die ersten beiden Deklarationen zu ersetzen durch

```
EXTERNAL INPBOOL, INPINT, ..., CONC, IV
EXTERNAL Z, HEAP, <Liste der nichtlokalen Größen der Runtime-
        Unterprogramme>
```

und die Deklarationen der Runtime-Unterprogramme sind wegzulassen. Die Übersetzung des Zielprogramms liefert dann ein Montageobjekt MOPAL; die Montage von MOPAL und MOUPPAL liefert dann den lauf-

fähigen Operator. Das geschilderte Vorgehen ist ein mögliches für den TR440; auf anderen Computern gehe man entsprechend vor.

Es bleibt nun noch die Generierung derjenigen BCPL-Anweisungen zu besprechen, die diejenige Information auf dem laufenden halten, welche die garbage collection bezüglich Hilfszellen benötigt.

Nach meinem heutigen Erkenntnisstand könnte man diese Aufgabe des Lauf 4 ersatzlos abschaffen, da dies auf einfache Weise ohne nennenswerte Nachteile möglich ist. Dazu hat man lediglich die Hilfszellenbehandlung des Lauf 3 etwas zu ändern: Lauf 3 hätte nun doch Hilfszellen von festem Typ und fester Art zu kreieren und die Spalten DTYP und DART von IDLI entsprechend zu definieren; die Art aux (6) würde dabei entfallen (daß es sich um eine Hilfszelle handelt, ist immer noch an der absoluten Größe des entsprechenden internal identifier erkennbar), statt dessen wären noch die beiden zusätzlichen Arten ptrvarptr (6) und ptrarrptr (7) aufzunehmen. Jedesmal, wenn Lauf 3 eine Hilfszelle benötigt, gibt er Typ und Art dieser Hilfszelle an (mit Hilfe seines Kellers kann er das unschwer); unter allen Hilfszellen wird dann gesucht, ob eine dieses Typs und dieser Art frei ist; ggfs. ist eine entsprechende neue Hilfszelle zu kreieren. Bezüglich der garbage collection werden Hilfszellen des Typs long oder chain und der Art var (d. h. Hilfsvariable dieser Typen) wie PAL-Variable dieser Typen und Hilfszellen der Art varptr (d. h. Hilfszeiger auf Variable) wie PAL-Zeiger auf Variable behandelt: Sie sind wie die entsprechenden PAL-Größen in Permanenz zur garbage collection angemeldet und werden damit von ihr erfaßt; alle andersartigen Hilfszellen sind von der garbage collection ohnehin nicht betroffen. Dabei wird es dann zwar passieren, daß die garbage collection long- und chain-Werte, die in momentan irrelevant besetzten Hilfszellen "enthalten" sind, aufhebt und nicht wegwirft; es ist aber äußerst unwahrscheinlich, daß dies zu einer nennenswerten Belastung der Halde führt. Im übrigen wird man dann auch die Benennung der Hilfszellen durch BCPL-Namen den Konventionen für PAL-Größen anpassen: Die für Y gemachte Ausnahme und der Fall X = HZ entfallen und für X gilt zusätzlich:

X = HV für Hilfszellen der Art var (Hilfsvariable)

HP.V für Hilfszellen der Art varptr (Hilfszeiger auf Variable)

HP.A für Hilfszellen der Art arrptr (Hilfszeiger auf Felder)

HHP.V für Hilfszellen der Art ptrvarptr (Hilfshyperzeiger auf Variable)

HHP.A für Hilfszellen der Art ptrarrptr (Hilfshyperzeiger auf Felder)

Die momentane Implementierung des Lauf 4 jedoch hat diese Aufgabe zu erledigen. Zu diesem Zweck ist in die STATIC-Deklaration des Kopfs des Zielprogramms folgende Vereinbarung zu setzen:

```
VHZ = TABLE <Anzahl der Hilfszellen+2>, U REP <Anzahl der Hilfs-
          zellen+2>;
```

VHZ ist ein globaler Parameter von GC und es gilt, daß jeder Komponente von VHZ außer den ersten dreien eine Hilfszelle zugeordnet ist: VHZ!3 ist die Hilfszelle mit dem kleinsten internal identifier zugeordnet, VHZ!4 die Hilfszelle mit dem zweitkleinsten internal identifier usw. . Unter den mit der Marke START im Kopf des Zielprogramms markierten Vorbesetzungen finden sich auch diejenigen für VHZ!3, VHZ!4 usw. : VHZ!i enthält die Adresse seiner zugehörigen Hilfszelle in negativer Form. Es gilt nun folgende Konvention: Eine Hilfszelle ist zur garbage collection genau dann angemeldet, wenn die zugehörige Komponente von VHZ die Adresse der Hilfszelle in positiver Form enthält. Die soeben erwähnten Vorbesetzungen besagen also, daß zunächst alle Hilfszellen abgemeldet sind. Die von Lauf 4 zu generierenden An- und Abmeldungen haben die einheitliche Form

```
VHZ ! i := - VHZ ! i;
```

Das liegt daran, daß eine an- bzw. abzumeldende Hilfszelle stets gerade ab- bzw. angemeldet ist. Weiter gilt, daß Abmeldungen dem umgeformten Befehl voranzustellen sind und Anmeldungen entsprechend nachzustellen sind (bei inp-Befehlen ist eine Abmeldung zwischen den Aufruf von INP<TYPE> und die nachfolgende Zuweisung

zu setzen); dabei wird man eine Hilfszelle, die ab- und gleich darauf wieder angemeldet wird, weder ab- noch anmelden (sondern angemeldet lassen). Somit bleibt nur noch zu klären, unter welchen Umständen eine An- oder Abmeldung zu generieren ist.

Eine Hilfszelle ist anzumelden, wenn ihr ein long- oder chain-Wert oder die Adresse einer Variablen (die dann sicher keine Hilfszelle ist) zugewiesen wird. Lauf 4 hat dazu all diejenigen Befehle zu überwachen, die ein Zuweisungszeichen aufweisen, dem der internal identifier einer Hilfszelle und kein val vorangeht. Ist das Zuweisungszeichen gleich := long, := chain oder := var, so ist genau dann eine Anmeldung zu generieren.

Wann eine Hilfszelle abzumelden ist, ist für Lauf 4 schwieriger herauszufinden, im Prinzip aber gleich einfach: Sobald der Inhalt einer Hilfszelle HZ, die einen long- oder chain-Wert oder den Bezug auf eine Variable enthält, im aktuellen Befehl benutzt wird, ist HZ abzumelden. Einzige Ausnahme: Weist der Befehl ein save auf und enthält OPR den internal identifier einer Hilfszelle HZ, so ist HZ nicht abzumelden. Diese Regel gilt auch dann, wenn Lauf 3, wie empfohlen, auch bei integer-Vergleichen ggfs. ein save anliefert. Im übrigen ist ZW3 natürlich so informationsreich, daß man die Sorte des Inhalts von Hilfszellen immer ermitteln kann, wenn dazu auch etliche Fallunterscheidungen nötig sind.

Bei der Erläuterung der soeben besprochenen Aufgabe wurde eingangs die Möglichkeit der Abschaffung der Aufgabe erwähnt, wobei die garbage collection manche irrelevanten long- und chain-Werte nicht wegwerfen würde. Dieser Nachteil läßt sich allerdings von Lauf 4 unter sehr geringem Mehraufwand vermeiden: Er generiert doch An- und Abmeldungen für die soeben besprochenen Fälle, zieht für die nötigen Entscheidungen aber nicht den Kontext des Befehls, sondern direkt die (dann vorhandenen) Angaben zu Typ und Art der Hilfszelle heran. Dies Vorgehen wäre, auch von der Logik her, fraglos die sauberste Lösung und sei deshalb letztlich empfohlen.

Dem Lauf 4 stehen als Input KOLI, BZLI, IDLI und ZWLI3 zur Verfügung; er benötigt alle diese Listen, verändert sie aber nicht,

wobei es ihm jedoch freigestellt ist, etwa Spalten in IDLI vorübergehend für eigene Zwecke zu benutzen. Der eigentliche Output von Lauf 4 ist somit nur ZW4. ZW4 ist in der besprochenen Form des Zielprogramms als String in eine Texthaltungsdatei abzusetzen, sodaß der BCPL-Compiler diesen String ohne Fehlermeldung übersetzen kann. Im übrigen gibt es für Lauf 4 weder Listenüberläufe noch sonstige Fehlersituationen: Lauf 3 hat den Rest der zur Übersetzungszeit durchführbaren Kontrollen erledigt; alle sonstigen Kontrollen können erst zur Laufzeit erfolgen. Deshalb verabschiedet sich Lauf 4 in jedem Fall mit der Meldung "Lauf 4 ohne Fehler beendet".

VI. Runtime-System

Hier wird zunächst auf die Darstellung der PAL- und Hilfsgrößen durch BCPL-Größen des Zielprogramms eingegangen; dann werden die Runtime-Unterprogramme mehr oder weniger ausführlich charakterisiert.

Ein BCPL-Vektor besteht aus einem einzelnen BCPL-Element L, welches auf das erste BCPL-Element einer zusammenhängenden Sequenz V von BCPL-Elementen verweist. L haben wir schon früher "Leitzelle" getauft und V heiße der "eigentliche Vektor" des BCPL-Vektors. Unter dem i-ten Element eines BCPL-Vektors wird stets das i-te Element des eigentlichen Vektors verstanden; seine BCPL-Bezeichnung lautet: L!(i-1), d. h. die Indizierung in BCPL beginnt bei 0.

PAL-Größen werden im Rahmen des Zielprogramms wie folgt dargestellt, wobei daran erinnert sei, daß PAL-Konstante wie einfache Variable desselben Typs behandelt werden, jedoch außer ihrer Initialisierung in der STATIC-Deklaration des Kopfs des Zielprogramms keinen Wert mehr zugewiesen erhalten.

1) Konstante und einfache Variable vom Typ bool:
   Sie werden durch BCPL-Elemente dargestellt, die TRUE bzw. FALSE enthalten.

2) Konstante und einfache Variable vom Typ int:
   Sie werden durch BCPL-Elemente dargestellt, die ganze Zahlen enthalten.

3) Konstante und einfache Variable vom Typ long:
   Sie werden durch BCPL-Vektoren dargestellt; solange die einfache Variable noch undefiniert ist, existiert nur die mit U vorbesetzte Leitzelle. Der eigentliche Vektor liegt nur bei einfachen Variablen im heap, sodaß nur sie an der garbage collection teilnehmen. In jedem Fall aber ist der eigentliche Vektor wie folgt aufgebaut: Die ersten beiden Elemente sind für Zwecke der garbage collection reserviert (bei Konstanten sind sie nur aus Einheitlichkeitsgründen vorhanden); im dritten Element steht die Anzahl der Dezimalziffern in der Form,

wie sie für den Operator dig benötigt wird; im vierten Element steht das Vorzeichen in der Form, wie es für den Operator sign benötigt wird. Bei long-Werten gleich 0 ist dies alles; ansonsten steht in den restlichen Elementen rechtsbündig der Betrag des long-Werts, je Element sechs Dezimalziffern (ggfs. Auffüllung des fünften Elements mit führenden Nullen). Vereinbarung: Unter der Adresse einer Konstanten oder einer einfachen Variablen vom Typ long wird die Adresse der betreffenden Leitzelle verstanden.

4) Konstante und einfache Variable vom Typ label:
Sie werden durch BCPL-Elemente dargestellt, die Programmadressen enthalten. Im Fall von Konstanten übernimmt der BCPL-Compiler selbst die Kreierung entsprechender (gleichnamiger) statischer Variabler samt Initialisierung. Jedenfalls sind all diese BCPL-Elemente auch auf der rechten Seite von BCPL-Zuweisungen benutzbar.

5) Konstante und einfache Variable vom Typ string:
Sie werden durch BCPL-Vektoren dargestellt; solange die einfache Variable noch undefiniert ist, existiert nur die mit U vorbesetzte Leitzelle. Der eigentliche Vektor liegt nur bei einfachen Variablen im heap, sodaß nur sie an der garbage collection teilnehmen. In jedem Fall aber ist der eigentliche Vektor wie folgt aufgebaut: Die ersten beiden Elemente sind für Zwecke der garbage collection reserviert (bei Konstanten sind sie nur aus Einheitlichkeitsgründen vorhanden); im dritten Element steht die Anzahl der Zeichen des Strings. Bei leeren chain-Werten ist dies alles; ansonsten stehen in den restlichen Elementen linksbündig die Zeichen selbst, je Element drei PAL-Zeichen (ggfs. Auffüllung des letzten Elements mit Ignore-Zeichen Ø), aufgereiht wie in BZLI-Einträgen des Typs chain. Vereinbarung: Unter der Adresse einer Konstanten oder einfachen Variablen vom Typ chain wird die Adresse der betreffenden Leitzelle verstanden.

6) Felder
Sie werden durch BCPL-Vektoren V dargestellt, deren zweites Element Leitzelle eines BCPL-Vektors F ist. Die Leitzelle von

V ist die Feldzelle des Feldes; der eigentliche Vektor von V ist der Informationsvektor des Feldes und der eigentliche Vektor von F enthält die Komponenten des Feldes in lexikographischer Reihenfolge, wobei noch zwei zusätzliche Elemente für Zwecke der garbage collection angefügt sind. Für die Komponenten des Feldes gelten dieselben Konventionen wie für einfache Variable desselben Typs; der Informationsvektor enthält der Reihe nach: Die Dimension n des Feldes; Verweis auf das erste Element des eigentlichen Vektors von F (physikalische Anfangsadresse des Feldes); die n unteren Grenzen der Reihe nach; die n Kantenlängen der Reihe nach. Die unteren Grenzen und die Kantenlängen sind mit 0 vorbesetzt, die Leitzelle von F mit U; sobald erstmalig eine Komponente des Feldes aufgerufen wird, wird der Informationsvektor entsprechend aktualisiert und der eigentliche Vektor von F eingerichtet; letzterer ist der einzige Teil des Feldes, der im heap liegt.

7) Zeiger
Sie werden durch BCPL-Elemente dargestellt, die Bezüge (letztlich: Adressen) auf andere BCPL-Elemente enthalten: Zeiger auf Variable vom Typ bool, int und label enthalten Bezüge auf die entsprechenden BCPL-Elemente; Zeiger auf Variable vom Typ long und chain enthalten Bezüge auf die Leitzellen der entsprechenden BCPL-Vektoren; Zeiger auf Felder schließlich enthalten Bezüge auf die entsprechenden Feldzellen.

8) Hilfszellen
Sie werden genau so wie die entsprechenden PAL-Größen gehandhabt, vgl. 1 - 5 und 7. Hilfszellen der Art ptrvarptr und ptrarrptr werden analog zu 7 behandelt: Sie werden durch BCPL-Elemente dargestellt, die Bezüge auf Zeiger enthalten.

Für die Unterprogramme des Runtime-Systems gilt generell, daß all diejenigen Unterprogramme außer GC, die Platz im heap benötigen, zunächst feststellen müssen (und auch können), wieviel Platz sie im heap benötigen; ggfs. ist GC aufzurufen, was Neudefinitionen von Größen des rufenden Unterprogramms erforderlich machen kann. Erst dann können sie den Platz im heap zur Informationsablage benutzen, wobei sie sich natürlich stets an die noch zu nennenden

Konventionen für die Halde halten müssen.

Am TR440 ist es zweckmäßig, den vier Einlesefunktionen eine gemeinsame Hilfsprozedur INDAT zur Verfügung zu stellen, die die von Lauf 1 abgelegten Quelldaten aufbereitet und zeichenweise anliefert, da die Konventionen für ALGOL und BCPL leider erheblich voneinander abweichen. INDAT dürfte unter Heranziehung entsprechend vieler BCPL-Standardprozeduren zwar in BCPL formulierbar sein, Effizienzgründe legen es aber dringend nahe, INDAT als Codeprozedur in TAS (Assemblersprache des TR440) zu formulieren. Auch das Argument der Maschinenunabhängigkeit sticht hier nicht, da die Menge von vorhandenen BCPL-Standardprozeduren ebenfalls hochgradig maschinenabhängig ist. INDAT hat die Aufgabe, in den Puffer INPU die Zeichen des nächsten Eingabedatums zu füllen: Die erste Komponente von INPU soll die Zeichenanzahl, die zweite den Typ und die restlichen Komponenten die Zeichen (je Komponente ein PAL-Zeichen) des Eingabedatums enthalten. INPU selbst ist ein BCPL-Vektor, der in der STATIC-Deklaration des Kopfs des Zielprogramms zu vereinbaren ist; die dazu nötige Vektorlänge ist um 2 höher als KOLI[53] angibt. Im übrigen wurde hier stillschweigend die revidierte Datenbehandlung des Lauf 1 unterstellt. Mittels INDAT fällt es den vier Eingabefunktionen leicht, den eingelesenen Wert gemäß den für Konstanten gültigen Konventionen aufzubauen; INPBOOL und INPINT liefern den aufgebauten Wert ab, INPLONG und INPCHAIN liefern einen Verweis auf den eigentlichen Vektor des aufgebauten Werts ab. Man beachte, daß INPLONG auch Werte vom Typ int einlesen kann, vgl. Seite 24.

Die Ausgaberoutinen bereiten (jedenfalls am TR440) keine besonderen Schwierigkeiten. Die Gestaltung des Druckbildes ist dabei ausschließlich Sache des PAL-Programmierers; er kann mittels der Operatoren dig und $_{10}$ seinen Output nach Wunsch gestalten; für die Länge von Strings ist er selbst verantwortlich. Die Ausgabe erfolgt unter Heranziehung der symbolischen Gerätenummer in KOLI[45]. Steht dort 0, wird der gesamte Output nicht ausgegeben, sondern unterdrückt. Andernfalls wird in die mit der symbolischen Gerätenummer 75 assoziierten (Ausgabe-) Datei D ausgegeben. Falls

der PAL-Programmierer nur M als Vorschubzeichen und sonst nur direkt druckbare Zeichen ausgibt, darf D auch eine Texthaltungsdatei sein. Diese wäre dann für PAL-Programme wieder einlesbar.

UNPLUS, UNMINUS und ABS erstellen im wesentlichen eine Kopie eines long-Werts und liefern einen Verweis auf diese Kopie ab.

SGP, SGN und SIGN erstellen im wesentlichen eine 1, 0 oder -1 vom Typ long und liefern einen Verweis darauf ab.

SCALE erstellt im wesentlichen eine Kopie eines long-Werts, wobei an dessen Ende Nullen angehängt bzw. Dezimalziffern abgeschnitten werden. Die Rechtsbündigkeit des Ergebnisses erfordert noch ein Umfüllen der Ziffern. Arithmetische long-Operationen sind jedenfalls nicht involviert. Abgeliefert wird ein Verweis auf das Ergebnis.

MULT leistet die Multiplikation zweier long-Werte und liefert einen Verweis auf das Ergebnis ab. Dabei wird man einleitend abfragen, ob das Ergebnis gleich 0 ist. Nur wenn dies nicht der Fall ist, wird man beide Faktoren FL und FR "spreizen", d. h. Kopien L und R erstellen, die rund doppelt so viel Speicherplatz belegen, da sie nur drei Ziffern je Element aufweisen. Dann kann man z. B. so vorgehen: Im ersten Schritt wird das letzte Element von L mit dem letzten Element von R multipliziert (die letzten drei Ziffern davon sind die letzten drei Ziffern des Ergebnisses); im zweiten Schritt wird die Summe aus den Produkten (letztes Element von L * vorletztes Element von R) und (vorletztes Element von L * letztes Element von R) gebildet; diese Summe wird zu dem im ersten Schritt gebildeten Produkt stellenrichtig addiert, womit die letzten sechs Ziffern des Ergebnisses feststehen und abgespalten werden können. Dieses Verfahren ist nun im dritten Schritt unter Einschluß der dritt- und im vierten Schritt unter Einschluß der viertletzten Elemente von L und R analog fortzusetzen, wodurch sich die vorletzten sechs Ziffern des Ergebnisses ergeben und abgespalten werden können usw. . Nichtvorhandene Elemente von L und R sind dabei als vorhanden und mit 0 besetzt zu

simulieren. Gilt n = (Anzahl der Elemente von L) + (Anzahl der Elemente von R), so ist der (n-1)-te Schritt der letzte bei diesem Verfahren. Im übrigen benutze man die Tatsache, daß man zwei BCPL-Elemente, die sechsziffrige Zahlen enthalten, ohne weiteres addieren kann (jedenfalls am TR440), da das Ergebnis sicher in 1 BCPL-Element hineinpaßt.

DURCH, DIV und MOD leisten im wesentlichen dieselbe Division zweier long-Werte; dabei ist die Division in allen überlangen Arithmetiken zeitmäßig die Achillesferse, gleichgültig, ob man sich für ein direktes oder ein iteratives Verfahren entscheidet. Ein direktes Verfahren erhält man z. B. durch die Nachbildung der Division von Hand; dazu würde man sich im heap zunächst eine Tabelle erstellen, die das 0- bis 10-fache des Nenners enthält. Ein iteratives Verfahren würde so aussehen, daß man durch eine grobe Division eine Näherung des Ergebnisses bestimmt; mit deren Hilfe errecnnet man durch eine Multiplikation und eine Subtraktion den dazugehörigen Divisionsrest; aus diesem und dem Nenner ergibt sich die erste Korrektur für die Näherung des Ergebnisses, damit ein zweiter Divisionsrest usw. . Je rascher die Beträge der Divisionsreste abnehmen, desto besser konvergiert das iterative Verfahren; man ist fertig, sobald der Betrag des jüngsten Divisionsrestes kleiner als der Betrag des Nenners ist. Jedenfalls aber ist es aus Effizienzgründen zu vermeiden, sonstige Runtime-Unterprogramme (außer GC) aufzurufen; man kopiere statt dessen die wesentlichen Teile davon. Abzuliefern ist schließlich ein Verweis auf das Ergebnis.

PLUS und MINUS leisten im wesentlichen dieselbe Addition zweier long-Werte und bereiten keine besonderen Schwierigkeiten; man benutze ggfs. die Unterstellung, daß nichtnegative Ergebnisse insgesamt gesehen wahrscheinlicher sind als negative. Abgeliefert wird ein Verweis auf das Ergebnis.

LESS, GRT, LEEQ, GREQ, EQL und NEQ leisten die entsprechenden Vergleiche zweier long-Werte und liefern als Ergebnis TRUE bzw. FALSE ab.

CONC leistet die Konkatenation zweier Strings und liefert einen Verweis auf das Ergebnis ab. Dabei wird man den Fall, daß mindestens einer der beiden Operanden der leere String ist, gesondert behandeln: Der andere Operand wird dann lediglich kopiert.

IV liefert die Adresse einer indizierten Variablen als Ergebnis an; dabei kann eine Feldvergrößerung nötig werden. IV wird über den Vektor S versorgt: S!0 ist (außerordentliche) Leitzelle zum Informationsvektor des aufgerufenen Feldes; S!1 bis S!n enthalten die reduzierten Werte der Indizes. Dazu ist anzumerken, daß man die Adresse einer indizierten Variablen grundsätzlich auf zwei Weisen berechnen kann: Durch Kombination der originalen Indexwerte mit der reduzierten Anfangsadresse des Feldes oder durch Kombination der reduzierten Indexwerte mit der physikalischen Anfangsadresse des Feldes. Wir beschreiten hier den letztgenannten Weg, um das Risiko der Nicht-Darstellbarkeit der reduzierten Anfangsadresse und damit zusätzlicher Einschränkungen für Indexwerte relativ zu ganzen Zahlen zu vermeiden.

Seien F das aufgerufene Feld und UG!I (OG!I, K!I) die momentanen unteren Grenzen (oberen Grenzen, Kantenlängen) von F; ferner mögen die Elemente T!I die Originalwerte der Indizes enthalten. Dann gilt:

UG!I ≤ T!I ≤ OG!I ⇔ 0 ≤ S!I < K!I

Liegt also jeder Wert S!I (I>0) zwischen 0 (einschließlich) und K!I (ausschließlich), so ist (genau dann) die aufgerufene Komponente des Feldes bereits vorhanden; ihre Adresse ergibt sich zu

LV S!0!1!(S!1) für n = 1 und
LV S!0!1!((...((S!1*K!2+S!2)*K!3+S!3)*K!4+...+S!(n-1))*K!n+S!n)
für n ≥ 2; dem Aufruf von S!1 gehen n - 1 Klammern voran.

Ist die Komponente des Feldes noch nicht vorhanden, so sind zwei Fälle zu unterscheiden:

S!0!1 = U :
Dann handelt es sich um den dynamisch ersten Aufruf des Fel-

des. Im heap ist (nach noch zu nennenden Konventionen) ein Feldeintrag einzurichten (ggfs. muß GC aufgerufen werden), dessen erstes Element V mit U zu initialisieren ist; in S!0!1 ist ein Verweis auf V abzusetzen; in S!0!2 bis S!0!(n+1) sind die Werte von S!1, S!2 usw. als untere Grenzen einzutragen; S!0!(n+2) bis S!0!(2n+1) schließlich sind auf 1 (jetzige Kantenlängen) zu setzen. Der Inhalt von S!0!1 ist als Ergebnis abzuliefern.

S!0!1 ≠ U :

Dann ist das schon vorhandene Feld erst auszuweiten und der Informationsvektor und der Vektor S zu aktualisieren, ehe man auf die eingangs genannte Weise die Adresse findet. Das Ausweiten des Feldes besteht darin, daß man eine passend gespreizte Kopie F_{neu} des Feldes $F = F_{alt}$ erstellt und dann F_{alt} wegwirft. Schließlich sind noch alle Zeiger auf F_{alt} "umzubiegen".

Dazu ermittle man zunächst die Werte der eingangs genannten Größen UG!I, T!I und OG!I sowie die neuen unteren und oberen Grenzen UH!I und OH!I. Dann prüfe man, ob eine garbage collection nötig ist und rufe ggfs. GC auf. Danach steht fest, wo im heap die Komponenten von F_{neu} einzurichten sind. Letztere sind nun lexikographisch aufzuzählen; dabei ist jede Komponente von F_{neu}, die in F_{alt} nicht auftritt, mit U zu initialisieren, andernfalls mit dem Wert der Komponente in F_{alt} (keine Definiertheitskontrolle durchführen). Durch vorherige Berechnung geeigneter Hilfsgrößen ist es möglich, zusammenhängende Bereiche von neuen wie auch schon vorhandenen Komponenten zu erkennen und die Initialisierung von F_{neu} dadurch erheblich zu beschleunigen. Ob man das Umbiegen der Zeiger auf F_{alt} in Zeiger auf F_{neu} vor oder nach dem Kopieren erledigt, ist gleichgültig; jedenfalls ist dazu folgendes zu tun: Mit Hilfe der bei der garbage collection beschriebenen Vektoren VPZ und VHZ ermittle man alle Zeiger und angemeldeten Hilfszellen, die auf ein Element von F_{alt} verweisen. Jeder dieser Verweise V ist zu einem entsprechenden Verweis W auf F_{neu} zu machen. Dazu ist der Reihe nach zu tun: Umrechnung von V in eine

Adresse V' relativ zum Anfang von F_{alt}; mit Hilfe der Kantenlängen von F_{alt} Ermittlung der entsprechenden reduzierten Indizes S_i; mit Hilfe der unteren Grenzen von F_{alt} Ermittlung der entsprechenden originalen Indizes T_i (diese gelten in gleicher Weise auch für F_{neu}); mit Hilfe der unteren Grenzen von F_{neu} Ermittlung der entsprechenden reduzierten Indizes S'_i; mit Hilfe der Kantenlängen von F_{neu} Ermittlung von W; Speicherung von W in den betreffenden Zeiger bzw. die betreffende Hilfszelle. Soweit das Umbiegen der Zeiger. Nun sei noch darauf hingewiesen, daß F_{neu} erst nach dem Kopieren in die Haldenstruktur eingefügt werden darf; näheres dazu bei der garbage collection.

GC schließlich leistet die garbage collection, d. h. die Säuberung der Halde (engl. heap) von irrelevanten Informationen. GC weist vier Parameter auf: Über den ersten Parameter wird GC mitgeteilt, ob es von IV aufgerufen wird (TRUE) oder nicht (FALSE); über den zweiten Parameter erfährt GC, wieviel Platz benötigt wird; GC prüft, ob nach der Bereinigung soviel freier Platz vorhanden ist; wenn nein, meldet GC "Abbruch in Zeile x: Halde zu klein" und beendet das Zielprogramm. Über den dritten und vierten Parameter können GC die Adressen von bis zu zwei zusätzlichen Hilfszellen der Art var und des Typs long oder chain mitgeteilt werden. Dadurch ist GC in der Lage, diese Hilfszellen mittels der Komponenten VHZ!1 und VHZ!2 des Vektors VHZ noch zur garbage collection anzumelden. Will man weniger als zwei Hilfszellen anmelden lassen, so rufe man GC mit entsprechend vielen aktuellen Parametern -1 auf; GC setzt VHZ!1 und/oder VHZ!2 dann auf -1, wodurch abgemeldete Hilfszellen simuliert werden. Dieser Mechanismus ist aus folgendem Grund vorhanden: GC wird stets innerhalb eines anderen Runtime-Unterprogramms UP aufgerufen; UP bringt über seine Parameter bis zu zwei long- oder chain-Werte mit; die entsprechenden Parameter von UP sind also Hilfszellen, die von der garbage collection betroffen sind. Im übrigen sei daran erinnert, daß in Unterprogrammen UP ≠ GC, die Platz im heap benötigen, als erstes entsprechende Platzberechnungen durchzuführen sind und ggfs. GC aufzurufen ist; damit ist sichergestellt, daß es im heap zum Zeit-

punkt der garbage collection keine Zwischenergebnisse von UP gibt, die ebenfalls anzumelden wären.

Als nächstes wird der genaue Aufbau der Halde geschildert. Sie wird realisiert durch den Vektor HEAP des Zielprogramms und enthält genau diejenigen BCPL-Größen des Zielprogramms, deren Speicherbedarf erst zur Laufzeit bekannt wird:

1) Die eigentlichen Vektoren aller Hilfszellen der Art var und aller PAL-Variablen, die vom Typ long oder chain sind (im folgenden kurz "long- und chain-Werte" genannt); die Werte von long- und chain-Konstanten dagegen befinden sich nicht in der Halde, da ihr Platzbedarf zur Übersetzungszeit bekannt ist und sie unverändert bleiben.

2) Die Bereiche für die Komponenten von PAL-Feldern (im folgenden kurz "Felder" genannt).

Daraus folgt, daß folgende Größen nach einer Bereinigung der Halde von GC aktualisiert werden müssen:

1) Die Informationsvektoren aller Felder, die mindestens einmal bereits aufgerufen wurden.

2) Alle einfachen PAL-Variablen, die vom Typ long oder chain sind und einen definierten Wert besitzen.

3) Alle indizierten PAL-Variablen, die vom Typ long oder chain sind und einen definierten Wert besitzen.

4) Alle PAL-Zeiger auf Variable, die einen definierten Wert besitzen, sofern dieser Wert die Adresse einer indizierten Variablen ist.

5) Alle Hilfszellen, die (zur Zeit) von der Art var und dem Typ long oder chain sind und einen definierten und relevanten Wert besitzen.

6) Alle Hilfszellen, die (zur Zeit) von der Art varptr sind und einen definierten und relevanten Wert besitzen, sofern dieser Wert die Adresse einer indizierten Variablen ist.

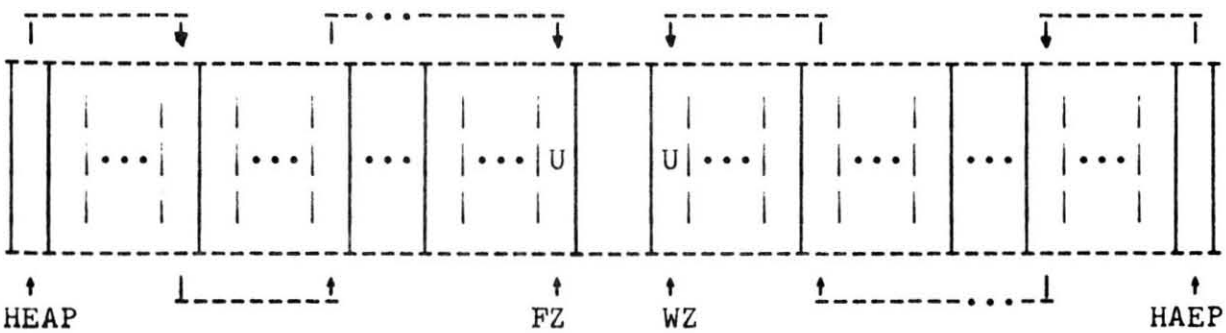
Anmerkung zu 4 und 6: Ob der PAL-Zeiger bzw. die Hilfszelle gera-

de einen Verweis auf eine einfache oder indizierte Variable enthält, ist zur Übersetzungszeit nicht entscheidbar und muß deshalb zur Laufzeit bei jeder garbage collection geprüft werden.

Wegen 3 und der Tatsache, daß indizierte Variable im heap gehalten werden müssen, teilen wir die Halde in drei disjunkte Bereiche F, L und W ein:

F ist der Bereich für die Felder, W der Bereich für die long- und chain-Werte und L enthält die momentan freien (leeren) BCPL-Elemente der Halde. F und W wachsen jeweils auf Kosten von L.

Die Einträge der Bereiche F und W sind jeweils durch lineare, geradlinige Verweisketten miteinander verbunden:



Ein Eintrag von F verweist also mit seinem letzten Element (dieses ist für die garbage collection reserviert) auf das letzte Element seines rechten Nachbarn; ein Eintrag von W verweist mit seinem ersten Element (dieses ist für die garbage collection reserviert) auf das erste Element seines linken Nachbarn. Damit ist auch festgelegt, wie jeweils eines der beiden je Haldeneintrag vorhandenen und für Zwecke der garbage collection reservierten BCPL-Elemente benutzt wird. FZ ("F-Zeiger"), WZ ("W-Zeiger") und HAEP sind zu GC globale BCPL-Elemente, die in der angegebenen Weise (wie HEAP) auf die Halde verweisen.

Bevor wir den prinzipiellen Ablauf einer garbage collection festlegen, erläutern wir die dazu nötigen BCPL-Größen.

Für GC sind folgende fünf Vektoren in der STATIC-Deklaration des Kopfs des Zielprogramms zu deklarieren:

```
VVY = TABLE <Anzahl der long- und chain-Variablen>,
            U REP <Anzahl der long- und chain-Variablen>;
VAX = TABLE <Anzahl der bool-, int- und label-Felder>,
            U REP <Anzahl der bool-, int- und label-Felder>;
VAY = TABLE <Anzahl der long- und chain-Felder>,
            U REP <Anzahl der long- und chain-Felder>;
VPZ = TABLE <Anzahl der Zeiger auf Variable>,
            U REP <Anzahl der Zeiger auf Variable>;
VHZ = TABLE <Anzahl aller Hilfszellen + 2>,
            U REP <Anzahl aller Hilfszellen + 2>;
```

Bei den mit START im Kopf des Zielprogramms markierten Vorbesetzungen sind u. a. die folgenden aufzunehmen:

```
HAEP := LV HEAP ! <heap-Größe - 1>;
RV HAEP := U; RV HEAP := U; FZ := HEAP; WZ := HAEP;
```

Ferner sind VVY!1, VVY!2 usw. mit den Adressen der long- und chain-Variablen vorzubesetzen, VAX!1, VAX!2 usw. mit den Adressen der Feldzellen der bool-, int- und label-Felder, VAY!1, VAY!2 usw. mit den Adressen der Feldzellen der long- und chain-Felder, VPZ!1, VPZ!2 usw. mit den Adressen der Zeiger auf Variable und VHZ!3, VHZ!4 usw. mit den Adressen der Hilfszellen. Die Adressen in VVY, VAX, VAY und VPZ müssen positive Form haben, die in VHZ negative Form, denn es gilt folgende Konvention: Diejenige Größe, deren Adresse in positiver Form in einem der fünf Vektoren steht, ist zur garbage collection angemeldet; andernfalls ist sie abgemeldet. Damit gilt: Alle von der garbage collection zu aktualisierenden PAL-Größen sind initial (und sogar permanent) zur garbage collection angemeldet, alle Hilfszellen initial abgemeldet, wobei wegen des An- und Abmeldens von Hilfszellen auf Lauf 4 verwiesen sei.

GC benötigt noch folgende in der STATIC-Deklaration des Kopfs des Zielprogramms zu deklarierenden Größen:

```
FANZ = U;
```

```
FANF = TABLE U REP <Anzahl aller Felder>;
FEND = TABLE U REP <Anzahl aller Felder>;
FTRA = TABLE U REP <Anzahl aller Felder>;
```

Eine garbage collection läuft nun prinzipiell wie folgt ab:

Schritt 1:

Mit Hilfe der Zeiger HEAP und HAEP und der Verkettungen wird in das jeweils noch freie garbage-collection-Element eines jeden Haldeneintrags ein U hineingesetzt. Bedeutung: Damit sind zunächst alle Einträge unmarkiert, d. h. können weggeworfen werden.

Schritt 2:

Mit Hilfe der Vektoren VVY, VAX, VAY und VHZ werden diejenigen heap-Einträge ermittelt, auf die zur Zeit ein damit auffindbarer Verweis existiert; diese Einträge werden markiert, indem das U von Schritt 1 ersetzt wird durch einen Verweis auf dasjenige BCPL-Element E, das auf den Eintrag verweist; bei long- und chain-Werten ist zusätzlich in E ein U hineinzusetzen. Bei eben diesen Werten ist außerdem zu beachten, daß es mehrere Verweise auf einen solchen Wert geben kann; in solch einem Fall ist dann eine Kette aufzubauen: Der long- oder chain-Wert zeigt auf E, E zeigt auf E' usw.; das Ende der Kette ist an $E^{(n)}$ = U erkennbar. Bei der Abarbeitung von VHZ sind jedoch nur diejenigen Hilfszellen zu nehmen, die angemeldet sind und in den Bereich W verweisen, d. h. einen long- oder chain-Wert "enthalten".

Parallel zur Abarbeitung von VAX und VAY sind in die Vektoren FANF und FEND die momentanen physikalischen Anfangs- und Endadressen der zur Zeit existenten Felder einzutragen; die physikalische Anfangsadresse im Informationsvektor wird mit dem betreffenden Index für FANF (, FEND und FTRA) überschrieben; abschließend ist in FANZ die Anzahl der existenten Felder einzutragen.

Schritt 3:

Die markierten Einträge des Bereichs W werden zusammengeschoben und WZ entsprechend aktualisiert. Parallel dazu werden mit

Hilfe der Verweise von Schritt 2 die entsprechenden Aktualisierungen durchgeführt.

Schritt 4:

Die markierten Einträge des Bereichs F werden zusammengeschoben und FZ entsprechend aktualisiert. Parallel dazu werden die entsprechenden Translationen in FTRA vermerkt. Der dazu benötigte Index für FTRA wurde in Schritt 2 im Informationsvektor vermerkt.

Schritt 5:

Nur falls GC von IV aufgerufen wurde und das von IV aufgerufene Feld F_{alt} existiert, ist nun F_{alt} (notfalls in mehreren Portionen, jedenfalls von hinten her) an das rechte Ende des jetzigen Bereichs L umzuspeichern und die betreffende Komponente von FTRA zu aktualisieren. Ferner sind die Felder rechts von F_{alt} im Bereich F an die restlichen Felder heranzuschieben und die betreffenden Komponenten von FTRA zu aktualisieren sowie FZ entsprechend zu vermindern. Diese Maßnahme stellt die bestmögliche Ausnutzung der Halde durch IV im Fall einer garbage collection sicher.

Schritt 6:

Mit Hilfe der Verweise von Schritt 2 werden die jetzigen physikalischen Anfangsadressen aller Felder in die zugehörigen Informationsvektoren eingetragen.

Schritt 7:

Mit Hilfe der Vektoren VPZ und VHZ sind diejenigen Zeiger und angemeldeten Hilfszellen zu korrigieren, die Verweise auf den ehemaligen Bereich F enthalten. Genau für diesen Zweck wurden in den vorangegangenen Schritten die Größen FANZ, FANF, FEND und FTRA definiert; sie enthalten die gesamte dafür nötige Information.

Schritt 8:

Ist nun WZ - FZ $\leq$ FN, so ist die Meldung "Abbruch in Zeile x: Halde zu klein" auszugeben und das Zielprogramm zu beenden; andernfalls erbrachte die garbage collection soviel Platz in der Halde, daß das Zielprogramm fortgesetzt werden kann.

Damit sind alle Unterprogramme des Runtime-Systems besprochen. Wenn das Zielprogramm in seinem Lauf nicht abgebrochen werden muß, verabschiedet es sich mit der Meldung "Programm ohne Fehler beendet".

VII. Simulation von ALGOL 60-Programmen

Zum Abschluß dieser Arbeit gehen wir nun noch auf die Simulation von ALGOL 60-Programmen durch PAL-Programme ein. Auf Seite 1 wurde unter Punkt 2 postuliert, daß es in einfacher Weise möglich sein soll, alle Sprachstrukturen von ALGOL 60 auszudrücken oder zu simulieren, wobei Prozeduren, die rekursiv sind oder nichtspezifizierte formale Parameter haben, außer Acht bleiben. Die folgenden Ausführungen sollen zeigen, daß dieses Ziel im wesentlichen erreicht wurde. Die Simulation von Prozeduren wird dabei als letztes besprochen; sie ist der schwierigste Punkt. Auch die Simulation von rekursiven Prozeduren wird abschließend erörtert.

Logische Werte, Strings und ganzzahlige Werte sind in PAL vorhanden; sollten int-Werte ihre Kapazitätsbeschränkung überschreiten, weiche man auf long-Werte aus. Reelle Werte sind nicht vorhanden; falls eine passende Dimensionierung nicht zu schwierig ist, kann man reelle Werte durch long-Werte darstellen. Andernfalls bilde man reelle Werte durch Paare (x,y) nach, wobei x die Mantisse (Typ long) und y der Exponent (Typ int) ist. Einfache Variable, indizierte Variable und Felder sind vorhanden. Verteiler sind, behandelt man sie korrekt, im wesentlichen nichts anderes als spezielle einparametrige Funktionsprozeduren; sie werden deshalb bei den Prozeduren behandelt. Standardfunktionen kann es in PAL nicht geben; ersatzweise wurden die Operatoren abs, sgp, sgn und sign aufgenommen. Die Transferfunktion entier ist in gewissen Grenzen durch den Operator div nachbildbar. Von den Operatoren ist ↑ nicht vorhanden, aber nachbildbar. ÷ ist auch nicht vorhanden, aber leicht simulierbar; man wird es nachbilden, wenn es auf die Nullpunktsymmetrie dieses Operators ankommt; andernfalls erfüllt das translationsinvariante div denselben Zweck. Die sonstigen Operatoren sind alle vorhanden. Alle Ausdrücke sind unschwer und meist sogar direkt nachbildbar. Zuweisungen und Sprünge sind vorhanden; die leere Anweisung nachzubilden ist auf Grund der Definition von PAL unnötig. Die Möglichkeit der Markierung ist wieder direkt vorhanden, ebenso wie Blöcke. Zusammengesetzte Anweisungen, bedingte Anweisungen und Laufanweisungen sind leicht nachbildbar. Die im-

pliziten Typtransfers integer ⟷ real, denen in PAL in etwa die Transfers int ⟷ long entsprechen, sind nicht vorhanden. Man könnte PAL entsprechend erweitern, z. B. durch weitere Operatoren, deren Bezeichnung andeutet, welche Transfers sie durchführen und/oder von welchem Typ ihr Ergebnis ist. Ansonsten kann man meist durch Einführung zusätzlicher PAL-Größen solche Transfers vermeiden. own in seiner statischen Interpretation ist sehr leicht, in seiner dynamischen Interpretation in Verbindung mit nur nichtrekursiven Prozeduren auch unschwierig, in Verbindung mit rekursiven Prozeduren aber nur schwer nachbildbar.

So bleiben noch die Prozeduren zu besprechen. Wir gehen zunächst auf die Simulation von nichtrekursiv aufgerufenen Prozeduren und dann auf die von rekursiv aufgerufenen Prozeduren ausführlich ein.

Nichtspezifizierte formale Parameter simulieren zu wollen ist bei einer so streng typ- und artgebundenen Sprache wie PAL nicht möglich; wir setzen deshalb ab jetzt voraus, daß sämtliche formalen Parameter spezifiziert sind. Dann bemerkt man sehr rasch, daß es immer noch eine ganze Hierarchie von Fällen gibt: Je besser man über die tatsächlich möglichen Aufrufe einer Prozedur in dem zu simulierenden ALGOL-Programm Bescheid weiß, desto wesentlich einfacher gestaltet sich die Simulation. Man könnte es durchaus als eine Schwäche von ALGOL 60 betrachten, daß es so wenige Möglichkeiten gibt, derartige Information dem ALGOL-Compiler zu Optimierungszwecken mitzuteilen. Man denke z. B. an die Mitteilung, ob eine Prozedur rekursiv aufgerufen wird oder nicht und an den krassen Informationsmangel, der bei der Simulation von Aufrufen formaler Prozeduren sofort auffällt. Wir beschränken uns im folgenden auf die Simulation von nichtrekursiv aufgerufenen Prozeduren und schlagen eine mögliche allgemeingültige Vorgehensweise vor. Wir weisen nochmal ausdrücklich darauf hin, daß es meist einfachere Möglichkeiten zur Simulation gibt, indem man spezielles Wissen über die Aufrufe einer Prozedur ausnützt; z. B. ist es oft möglich, einen Parameter value zu listen, ohne die Ergebnisse des ALGOL-Programms zu verändern oder statt eines call by name einen call by reference zu nehmen. Generell sei gleich noch festgehal-

ten, daß value gelistete formale Parameter einer Prozedur P (auch bei rekursiv aufgerufenen Prozeduren) im wesentlichen nichts anderes als deklarierte prozedurlokale Größen von P sind; sie haben lediglich die besondere Eigenschaft, im Rahmen eines Aufrufs von P initialisiert zu werden.

Die Parameterübergabe erfolgt im wesentlichen mit Hilfe einer Matrix PARAM aus programmglobalen PAL-Größen, die nur für diesen Zweck benutzt werden. Die PAL-Größen bilden nur logisch gesehen eine Matrix; tatsächlich sind es lauter nichtindizierte Größen. Jede Spalte besteht aus elf PAL-Größen, nämlich je PAL-Typ eine Variable dieses Typs und für die Typen bool, int und long je ein Zeiger auf Variable sowie auf Felder dieses Typs. Die Anzahl der Spalten ist gleich der maximalen Anzahl von aktuellen Parametern eines Prozeduraufrufs. Die Benutzung dieser Matrix sieht so aus: An der Aufrufstelle werden in den n ersten Spalten Informationen über die n vorhandenen aktuellen Parameter übergeben; je Parameter wird eine Spalte benutzt. Die Benutzung einer Spalte besteht dabei darin, daß in Abhängigkeit der Spezifikation des betreffenden formalen Parameters die entsprechende PAL-Größe mit passender Information definiert wird; nur sehr selten wird noch eine zweite Größe dieser Spalte definiert. Bei der Simulation von Aufrufen deklarierter Prozeduren kennt der menschliche Übersetzer die Spezifikationen der formalen Parameter und damit auch die zu definirenden PAL-Größen; bei der Simulation von Aufrufen formaler Prozeduren kann erst das simulierende PAL-Programm feststellen, welche deklarierte Prozedur momentan aufgerufen wird und mit deren Spezifikationen die zu definierenden PAL-Größen ermitteln. Zu diesem Zweck ist ein (Spalten-) Vektor PROZANF vorhanden, der die Anfangsadressen aller deklarierten Prozeduren enthält sowie eine Matrix SPEZ vom Typ int, die je Zeile die Spezifikationen der formalen Parameter der entsprechenden Prozedur in codierter Form enthält. Aus jeder Spezifikation muß dabei auch hervorgehen, ob der betreffende formale Parameter value gelistet ist oder nicht. Der menschliche Übersetzer hat dann ein PAL-Programmstück zu formulieren, welches die Anfangsadresse der aufgerufenen Prozedur in PROZANF und damit die aktuelle Zeile in SPEZ auffindet und dann

durch Abfrage dieser Spezifikationen die entsprechenden PAL-Größen so definiert wie es der menschliche Übersetzer bei Aufrufen von deklarierten Prozeduren von vornherein kann. Falls der menschliche Übersetzer genau weiß, welche deklarierten Prozeduren durch den betrachteten formalen Aufruf letztlich aufgerufen werden können und falls die formalen Parameter dieser deklarierten Prozeduren der Reihe nach in gleicher Weise value gelistet und spezifiziert sind, so kann er natürlich direkt wie bei Aufrufen deklarierter Prozeduren verfahren; diese Situation ist ein weiteres besonders schönes Beispiel dafür, wie zusätzliche Information die Simulation vereinfachen kann. Soweit die Ausführungen zu den Tätigkeiten an der Aufrufstelle. Auf der anderen Seite der Parameterübergabe, dem Rumpf der aufgerufenen (deklarierten) Prozedur, werden als erstes alle Angaben zu den aktuellen Parametern aus der Matrix PARAM in prozedurlokale Größen übertragen, womit PARAM für den nächsten Prozeduraufruf bereit ist. Logisch gesehen gehört zu PARAM noch eine einzelne weitere PAL-Variable vom Typ label; sie dient in analoger Weise der Übertragung eines weiteren, in ALGOL nicht explizit auftretenden Parameters, nämlich der Rücksprungadresse.

Außer PARAM gibt es noch weitere programmglobale PAL-Größen. Von diesen sei hier nur noch die wichtigste besprochen: Der simulierte Akkumulator. In ihm legen alle Funktionsprozeduren, d. h. die im ALGOL-Programm deklarierten Funktionsprozeduren, die noch zu besprechenden name procedures sowie diejenigen Funktionsprozeduren, durch die Verteiler nachgebildet werden, ihr Ergebnis ab. Der simulierte Akkumulator besteht aus elf PAL-Größen: Einer Variablen VALADDR vom Typ bool, je PAL-Typ eine Variable dieses Typs und je PAL-Typ ein Zeiger auf Variable dieses Typs. Die Funktionsprozedur gibt in VALADDR an, ob sie einen Wert (true) oder die Adresse einer Variablen (false) ermittelt hat und übergibt ihr Ergebnis in der entsprechenden Variablen bzw. dem entsprechenden Zeiger. Die Definition des Akkumulators darf i. a. natürlich erst unmittelbar vor Verlassen des Rumpfs der Funktionsprozedur erfolgen; ggfs. ist eine prozedurlokale Größe für Zwischenspeicherungen heranzuziehen.

Für das folgende stelle man sich einen Aufruf einer deklarierten Prozedur vor; wie Aufrufe formaler Prozeduren auf solche deklarierter Prozeduren zurückgeführt werden, wurde bereits besprochen. Insbesondere bei formalen Aufrufen achte man auf folgende Besonderheit: Ein aktueller Parameter, der eine vorzeichenlose ganze Zahl ist, kann auch ein integer-label sein!

Bei formalen Parametern, die value gelistet und real, integer, Boolean oder label spezifiziert sind, hat man den Wert des korrespondierenden aktuellen Ausdrucks auszurechnen und in PARAM abzusetzen.

Bei formalen Parametern, die string spezifiziert sind, verfahre man genauso wie eben, d. h. man ersetze den laut ALGOL gegebenen call by name durch einen call by value und setze dementsprechend den letztlich gemeinten korrespondierenden aktuellen String direkt in PARAM ab.

Die noch verbleibenden Fälle haben das folgende gemeinsam: Besteht der aktuelle Parameter aus genau einem Identifikator, der formaler und nicht value gelisteter Parameter einer Prozedur P ist, so ist die nach PARAM zu übertragende Information nicht wie im folgenden beschrieben erst zu bilden, sondern es ist die in den entsprechenden zu P lokalen Größen enthaltene Information direkt nach PARAM zu übertragen. Sprechweise: Der letztlich gemeinte aktuelle Parameter wird in die jüngst aufgerufene Prozedur durchgereicht. Im folgenden wird deshalb unterstellt, daß der aktuelle Parameter gegen eine der drei oben genannten Bedingungen verstößt.

Bei formalen Parametern, die value gelistet und (real) array, integer array oder Boolean array spezifiziert sind, übergebe man in PARAM einen Bezug auf das korrespondierende aktuelle Feld. Die aufgerufene Prozedur erstellt dann mittels lwb und upb eine prozedurlokale Kopie dieses Feldes. Ist dessen Dimension uneinheitlich oder für den menschlichen Übersetzer unbekannt, blähe man zuerst alle Felder des ALGOL-Programms, die über diesen formalen Parameter übergeben werden (schlimmstenfalls also alle Felder des

betreffenden Typs), durch Einführung zusätzlicher Indizes und Feldkanten der Länge 1 auf die gemeinsame maximale Dimension auf.

Bei formalen Parametern, die nicht value gelistet sind und die real, integer, Boolean oder label spezifiziert sind, hat man den korrespondierenden aktuellen Parameter als zusätzliche Funktionsprozedur (sog. name procedure) zu behandeln, die lokal zum kleinsten die betrachtete Aufrufstelle unmittelbar umfassenden Prozedurrumpf bzw. der dazugehörigen Prozedur ist. An PARAM wird in diesen Fällen lediglich die Anfangsadresse dieser Prozedur übergeben. Die Benutzung des formalen Parameters ist nichts anderes als der Aufruf einer parameterlosen formalen Funktionsprozedur.

Bei formalen Parametern, die nicht value gelistet sind und die (real) array, integer array oder Boolean array spezifiziert sind, übergebe man in PARAM einen Bezug auf das korrespondierende aktuelle Feld. Die aufgerufene Prozedur übernimmt dann einfach diesen Bezug.

Bei formalen Parametern, die procedure spezifiziert sind, übergebe man in PARAM die Anfangsadresse der korrespondierenden aktuellen (deklarierten, nicht unbedingt eigentlichen) Prozedur.

Bei formalen Parametern, die real procedure, integer procedure oder Boolean procedure spezifiziert sind, übergebe man in PARAM die Anfangsadresse der korrespondierenden aktuellen (deklarierten) Funktionsprozedur F. Dies reicht aber noch nicht; es gibt nämlich ALGOL-Programme, die Wertzuweisungen an Identifikatoren formaler Funktionsprozeduren enthalten, wobei es umstritten ist, ob der ALGOL-Report dies zuläßt oder nicht. Wir können das getrost zulassen und übergeben in derselben Spalte von PARAM ausnahmsweise eine zweite Information, nämlich die Adresse derjenigen PAL-Größe FF, die zu F lokal ist und zur Zwischenspeicherung des Ergebnisses von F dient (vgl. die Erläuterungen zum simulierten Akkumulator).

Anmerkung: Wollte man im Rahmen der Simulation auch die Korrektheit der Parameterübergaben prüfen, so könnte man die Adresse von

FF, oder ganz allgemein: die Adresse von FF in der momentan aktuellen Inkarnation von F, nur dann übergeben, wenn eine solche Inkarnation existiert; bei der Zuweisung an den formalen Funktionsidentifikator wäre dann erst zu prüfen, ob eine Adresse übertragen werden konnte.

Bei formalen Parametern schließlich, die switch spezifiziert sind, übergebe man in PARAM die Anfangsadresse der korrespondierenden aktuellen Funktionsprozedur, die man sich aus der switch-Deklaration gebildet zu denken hat. Jede Anwendungsstelle eines (deklarierten oder formalen) Verteilers verkörpert dementsprechend den Aufruf einer einparametrigen Funktionsprozedur, von der die Spezifikation integer ihres value zu listenden formalen Parameters auch dann direkt bekannt ist, wenn der Verteiler und damit die Prozedur formal ist: Der korrespondierende aktuelle Parameter ist nämlich der Indexausdruck des switch designators. Im Rumpf einer derartigen Funktionsprozedur ist folgendes zu tun: Ist der Wert des Indexausdrucks "out of range", so ist als Ergebnis im simulierten Akkumulator eine spezielle programmglobale PAL-Marke anzuliefern, die ausschließlich dazu da ist, festzuhalten, daß (letztlich) eine nicht vorhandene Verteilerkomponente anzuspringen ist, d. h. daß (letztlich) gar kein Sprung ausgeführt werden soll, sondern hinter der betreffenden ALGOL-Sprunganweisung fortzufahren ist. Dementsprechend weisen alle (simulierten) Sprunganweisungen einen entsprechenden bedingten Zielausdruck auf, es sei denn, man kann mit Sicherheit ausschließen, daß (im Rahmen dieser Sprunganweisung) eine nichtvorhandene Verteilerkomponente angesprungen wird. Andernfalls ist der dem Wert des Indexausdrucks entsprechende Zielausdruck auszuwerten; ergibt sich eine Marke, so ist diese als Ergebnis im simulierten Akkumulator anzuliefern; andernfalls ergibt sich eine Verteilerkomponente, die ihrerseits in gleicher Weise zu behandeln ist. Die geforderte Nichtrekursivität von Prozeduren verbietet in diesem Fall natürlich, daß ein Verteiler direkt oder indirekt sich selbst aufruft. Wird andererseits ein Verteiler nur durch seinesgleichen rekursiv aufgerufen, so läßt sich Rekursivität im simulierenden PAL-Programm einfach vermeiden: Man beende den aktuellen Aufruf ohne Ergebnis und rufe

erst dann die nächste derartige Funktionsprozedur auf.

Es sei noch darauf hingewiesen, daß man switch spezifizierte formale Parameter sinnvoll value listen kann, auch wenn das in ALGOL nicht erlaubt ist. In diesem Fall übergebe man in PARAM die Anfangsadresse der korrespondierenden aktuellen, aus dem Verteiler gebildeten Funktionsprozedur und außerdem in derselben Spalte von PARAM eine zweite Information, nämlich die Anzahl der Komponenten des korrespondierenden aktuellen Verteilers, wobei diese zweite Information auch dann zu übergeben ist, wenn der formale Parameter nicht value gelistet ist. Im Rumpf der aufgerufenen Prozedur P ruft man im Fall der value-Listung dann die Komponenten des Verteilers der Reihe nach auf und notiert die Ergebnisse in einem zu P lokalen Vektor V. Dieser Vektor ist der Input für ein ebenfalls zu P lokales PAL-Programmstück Q. Q ist eine vereinfachte Version der aus deklarierten Verteilern gebildeten Funktionsprozeduren: Der Fall, daß der Indexwert out of range ist, ist zwar wieder abzufangen; andernfalls findet man das Ergebnis aber direkt in der betreffenden Komponente von V. Im übrigen ist Q genauso wie die anderen derartigen Funktionsprozeduren zu behandeln.

Soweit die Erläuterungen zur Simulation von nichtrekursiven Prozeduren, denen nun noch ein Vorschlag zur Simulation von rekursiv aufgerufenen Prozeduren folgt. Seine Realisierung erfordert eine Spracherweiterung von PAL um weitere Objekte der Referenzstufe 2, nämlich um indizierte Zeiger aller zehn Sorten, d. h. Felder aus Zeigern. Es sei nicht verschwiegen, daß die Erweiterung von PAL um diese Zeigerfelder erhebliche Änderungen mit sich bringt, sowohl in der Definition wie auch in der Implementierung von PAL.

Es ist sicher möglich, wenn auch nicht eben einfach, ein komplettes ALGOL 60-Runtime-System in PAL zu simulieren. Ein für unsere Zwecke erheblicher Nachteil einer derartigen Simulation besteht aber darin, daß es praktisch unmöglich ist, zusätzliche Information über das zu simulierende ALGOL-Programm zur Vereinfachung und Beschleunigung der Simulation einzusetzen; andererseits hat

man als menschlicher Übersetzer in aller Regel zusätzliche Information in beachtlichem Umfang und könnte damit wie schon bei nichtrekursiven Prozeduren erhebliche Vereinfachungen der Simulation erreichen. Diesen Vorteil bietet die im folgenden vorgeschlagene Simulation, die eine direkte Verallgemeinerung des Vorschlags zur Simulation von nichtrekursiven Prozeduren darstellt und nun in ihrer allgemeingültigen und damit umständlichsten Version behandelt wird.

Dazu brauchen wir den Begriff des statischen Vorgängers einer ALGOL-Größe. Jede ALGOL-Größe, deren Gültigkeitsbereich durch die Blockstruktur geregelt ist (also Identifikatoren und integer labels), besitzt genau einen statischen Vorgänger, worunter wir den bereinigten erweiterten Rumpf einer Prozedur verstehen wollen. Der unbereinigte erweiterte Rumpf einer Prozedur besteht dabei aus dem formal parameter part samt folgendem Semikolon, dem value part, dem specification part und dem procedure body der Prozedur; der bereinigte erweiterte Rumpf einer Prozedur P geht aus dem unbereinigten Rumpf von P durch Streichung aller unbereinigten erweiterten Rümpfe von zu P lokalen Prozeduren hervor. Weiter besitzt jede ALGOL-Größe, deren Gültigkeitsbereich durch die Blockstruktur geregelt ist, eine Kreationsstelle: Bei deklarierten Größen ist das diejenige Stelle ihrer Deklaration, an der der Identifikator der Größe zum ersten Mal auftritt; bei Marken ist das diejenige Stelle, an der sie in markierender Weise auftreten; bei formalen Größen ist das diejenige Stelle, an der sie zum ersten Mal in der Liste der formalen Parameter derjenigen Prozedur auftreten, von der sie formale Parameter sind. Damit gilt: Der statische Vorgänger einer ALGOL-Größe, deren Gültigkeitsbereich durch die Blockstruktur geregelt ist, ist derjenige bereinigte erweiterte Rumpf einer Prozedur, der die Kreationsstelle der ALGOL-Größe enthält; sonstige ALGOL-Größen besitzen keinen statischen Vorgänger. Dabei wird das ALGOL-Programm insgesamt als unbereinigter erweiterter Rumpf einer deklarierten parameterlosen Prozedur betrachtet; Standardprozeduren sind dabei als dazu lokale und in einem das eigentliche ALGOL-Programm einschließenden Block deklarierte Prozeduren zu behandeln.

Eine Prozedur P rekursiv aufzurufen heißt, P während eines Aufrufs von P direkt oder indirekt erneut aufzurufen. Bei dem erneuten Aufruf fällt ein zweiter und i. a. ein weiterer Satz von aktuellen Parametern an, die man nicht in die betreffenden prozedurlokalen Größen übertragen kann, da diese schon besetzt sind und ihre Werte i. a. noch benötigt werden. Dies ist die Motivation für die folgende, die Simulation von nichtrekursiven Prozeduren erweiternde Idee: Man macht die genannten prozedurlokalen Größen (und wegen der Kopierregel auch alle sonstigen Größen, deren Kreationsstelle im bereinigten erweiterten Rumpf der aufgerufenen Prozedur liegt und die Variable, Zeiger oder Felder sind) zu Vektoren bzw. bei Feldern zu Feldern mit einer um 1 erhöhten Dimension (d. h. mit einer zusätzlichen, etwa nunmehr ersten Feldkante) und richtet eine zusätzliche, dem bereinigten erweiterten Rumpf der aufgerufenen Prozedur zugeordnete (prozedurlokale) int-Variable INKVAR ein, die sog. lokale Inkarnationsvariable, in der die Anzahl der momentan jeweils noch nicht beendeten Aufrufe der Prozedur vermerkt ist; Sprechweise: INKVAR gibt stets an, wieviele (aktive) "Inkarnationen" der Prozedur momentan existieren. INKVAR wird nun bei der Parameterübergabe als (ggfs. erster) Index benutzt: Wird P aufgerufen, so wird nach der Definition von PARAM INKVAR um 1 erhöht und in die so bestimmten Vektorkomponenten bzw. Feldbereiche der betreffenden Größen werden die Angaben über die aktuellen Parameter eingetragen; wird die Inkarnation von P normal (d. h. nicht durch einen expliziten Sprung) verlassen, wird INKVAR wieder um 1 erniedrigt.

Im weiteren werden "nur" noch die Auswirkungen dieser Idee geschildert, um zu einer in sich konsistenten Vorgehensweise zu kommen. Wie schon angedeutet, unterscheiden wir bei allen deklarierten Prozeduren (das sind alle im ALGOL-Programm deklarierten Prozeduren, alle name procedures sowie alle Verteiler nachbildenden Prozeduren) zwischen einzelnen Inkarnationen dieser Prozeduren. Eine Inkarnation ist dabei ein Exemplar eines bereinigten erweiterten Prozedurrumpfs und ist eindeutig bestimmt durch das Paar (Anfangsadresse, Inkarnationsnummer). Nicht nur Prozeduren, sondern sämtliche Größen des ALGOL-Programms, deren Gültigkeits-

bereich durch die Blockstruktur geregelt wird (bzw. die entsprechenden Größen des PAL-Programms) sind wegen der Kopierregel mit Inkarnationsnummern zu versehen. Was das für Variable, Zeiger und Felder bedeutet, wurde bereits ausgeführt; sonstige betroffene Größen sind nun als Paare zu behandeln, deren zweite Komponente eine Inkarnationsnummer ist. Die einer ALGOL-Größe zugeordnete Inkarnationsnummer ist dabei dieselbe wie die ihres statischen Vorgängers.

Um nun die momentan aktuellen Inkarnationsnummern (das sind durchaus nicht immer die momentan höchsten Inkarnationsnummern) stets direkt greifbar zu haben, existiert zusätzlich ein Satz von programmglobalen int-Variablen, den sog. globalen Inkarnationsvariablen. Jedem statischen Vorgänger ist eineindeutig eine solche Variable zugeordnet. Diese Variable müssen auf eine noch zu nennende Weise ständig aktualisiert werden und sind mit 0 zu initialisieren außer der dem ALGOL-Programm zugeordneten Variablen, die mit 1 zu initialisieren ist. Die in ihnen enthaltenen Werte sowie die Werte der lokalen Inkarnationsvariablen sind so wichtig, daß sie unmittelbar vor Beginn der eigentlichen Tätigkeiten einer Prozedur P in diejenigen Zeilen zweier zu P lokaler int-Matrizen INKG (G wie global) und INKL (L wie lokal) kopiert werden, die INKVAR gerade angibt.

Zu Zwecken der Parameterübergabe verlängere man jede Spalte von PARAM um eine int-Variable, in der man zusätzlich eine Inkarnationsnummer des aktuellen Parameters überträgt (diese existiert nur dann nicht, wenn der korrespondierende formale Parameter string spezifiziert ist oder value gelistet ist und real, integer oder Boolean spezifiziert ist; diese Fälle seien im folgenden ausgeschlossen); i. a. werden also nun je Spalte von PARAM mindestens zwei Informationen übertragen. Zur Ablage dieser Inkarnationsnummern in der aufgerufenen Prozedur hat man entsprechende zusätzliche Vektoren vorzusehen. Ist nun der aktuelle Parameter ein formaler und nicht value gelisteter Identifikator, so findet sich die zu übertragende Inkarnationsnummer in einem solchen zusätzlichen Vektor, der zum statischen Vorgänger P des aktuellen Parame-

ters lokal ist; der zugehörige Index ist der P zugeordneten globalen Inkarnationsvariablen zu entnehmen; sonstige nach PARAM zu übertragenden Informationen finden sich in den entsprechenden, zu P lokalen Vektoren unter demselben Index. Andernfalls ist der aktuelle Parameter ein formaler und value gelisteter oder ein deklarierter Identifikator (möglicherweise der einer name procedure); die zu übertragende Inkarnationsnummer ist der dem statischen Vorgänger des aktuellen Parameters zugeordneten globalen Inkarnationsvariablen zu entnehmen; die sonst zu übertragende Information ist dieselbe wie bei nichtrekursiven Prozeduren.

Damit ist die Parameterübergabe eines Prozeduraufrufs geklärt. Es ist allerdings noch ein weiterer Parameter zu übertragen in einer zusätzlichen int-Variablen AKTINK ("aktuelle Inkarnation"), die logisch gesehen zu PARAM gehört, nämlich eine Inkarnationsnummer der aufgerufenen Prozedur. Handelt es sich um den Aufruf einer deklarierten Prozedur P, so ist die zu übertragende Inkarnationsnummer der dem statischen Vorgänger von P zugeordneten globalen Inkarnationsvariablen zu entnehmen; andernfalls handelt es sich um den Aufruf einer formalen Prozedur Q und die zu übertragende Inkarnationsnummer sowie die anzuspringende Adresse finden sich in den entsprechenden, zum statischen Vorgänger R von Q lokalen Vektoren; der zugehörige Index ist der R zugeordneten globalen Inkarnationsvariablen zu entnehmen.

Auf der Seite der aufgerufenen Prozedur P ist als erstes die zu P lokale Inkarnationsvariable INKVAR um 1 zu erhöhen; dann sind alle Angaben zu den aktuellen Parametern aus PARAM in der schon genannten Weise zu übernehmen. Als nächstes sind die Werte der in AKTINK angegebenen Zeile der zum statischen Vorgänger von P lokalen Matrix INKG in die globalen Inkarnationsvariablen zu übertragen; dann ist der Wert von INKVAR in die Q zugeordnete globale Inkarnationsvariable zu übertragen; Q ist dabei derjenige statische Vorgänger, der einem formalen Parameter von P zuzuordnen wäre. Schließlich sind die Werte der globalen und lokalen Inkarnationsvariablen in die von INKVAR angegebenen Zeilen der (zu P und Q lokalen) Matrizen INKG und INKL zu übertragen. Nun kann P mit sei-

nen eigentlichen Tätigkeiten beginnen. Wird P normal, d. h. durch Erreichen des Rumpfendes, verlassen, so ist noch als letzte Tätigkeit INKVAR um 1 zu erniedrigen. Man kehrt dann an die Aufrufstelle von P zurück; sie liege im bereinigten erweiterten Rumpf einer Prozedur Q. Hier sind nun noch die Werte der globalen Inkarnationsvariablen zu restaurieren mit Hilfe der Werte der zu Q lokalen Größen INKVAR und INKG: Die Werte der von INKVAR angegebenen Zeile von INKG sind in die globalen Inkarnationsvariablen zu übertragen. Erst damit ist die Simulation des Prozeduraufrufs gänzlich abgeschlossen. Diese Restaurierung hat eine besondere Konsequenz: Im Rahmen der Parameterübergabe kommt es bei formalen Parametern, die switch spezifiziert sind und value gelistet werden sollen, im Rumpf der aufgerufenen Prozedur zu n Prozeduraufrufen, wobei n die Anzahl der Komponenten des aktuellen Verteilers ist. Auch nach jedem dieser Prozeduraufrufe (außer dem letzten) ist diese Restaurierung nötig. Deshalb wird für diesen Fall empfohlen, die durch INKVAR (bereits erhöhter Wert) angegebene Zeile von INKG zur vorübergehenden Abspeicherung der (alten) Werte der globalen Inkarnationsvariablen zu benutzen und mit Hilfe dieser Zeile die Restaurierungen durchzuführen, ehe sie mittels AKTINK endgültig definiert wird.

Sprünge werden in Verbindung mit rekursiven Prozeduren auch etwas komplizierter und zwar sowohl an der Ab- wie auch an der Ansprungstelle, wobei der Ansprung einer nichtvorhandenen Verteilerkomponente wieder abzufangen ist. An der Absprungstelle ist unmittelbar vor dem Absprung AKTINK mit einer Inkarnationsnummer zu versehen. Wird eine deklarierte Marke M angesprungen, so ist AKTINK mit dem Wert der dem statischen Vorgänger von M zugeordneten globalen Inkarnationsvariablen zu versehen. Wird ein formaler Parameter N angesprungen, der value gelistet und label spezifiziert ist, so finden sich der Wert für AKTINK und auch die Ansprungadresse in den betreffenden zum statischen Vorgänger P von N lokalen Vektoren, wobei als Index der Wert der P zugeordneten globalen Inkarnationsvariablen zu nehmen ist. Wird ein formaler Parameter angesprungen, der nicht value gelistet und label spezifiziert ist, so liegt ein parameterloser Prozeduraufruf vor, der

formal ist und nach den dafür gültigen Konventionen abzuwickeln ist; er liefert den Wert für AKTINK sowie die anzuspringende Adresse im simulierten Akkumulator als Ergebnis an. Andernfalls wird eine Komponente eines (deklarierten oder formalen) Verteilers angesprungen, was einen entsprechenden Funktionsaufruf zur Folge hat, der den Wert für AKTINK sowie die anzuspringende Adresse im simulierten Akkumulator als Ergebnis anliefert. Für die Ansprungstelle gilt: Sei R der statische Vorgänger der tatsächlich angesprungenen Marke. Dann sind an der Ansprungstelle die Werte der von AKTINK angegebenen Zeilen der zu R lokalen Matrizen INKG und INKL in die entsprechenden globalen und lokalen Inkarnationsvariablen zu übertragen. Erst damit ist der Sprung beendet. Falls man sicher sein kann, daß alle globalen und lokalen Inkarnationsvariablen dabei mit Werten definiert werden, die sie ohnehin schon enthalten, so kann man sich deren Restaurierung sowie die Ermittlung des Werts für AKTINK natürlich schenken und direkt hinter die die Restaurierung realisierenden PAL-Anweisungen springen. Dies ist z. B. der Fall, wenn man die Ansprungstelle ohne Sprung auf natürlichem Weg erreicht oder bei einem inkarnationslokalen Sprung, d. h. wenn man durch den Sprung die jüngste Prozedurinkarnation nicht verläßt (auch nicht vorübergehend). Weiter bemerkt man, daß die prozedurlokalen Matrizen INKL nur in Verbindung mit Sprüngen benötigt werden. Weist ein Prozedurrumpf entweder keine oder nur kommentierende Markierungen und/oder inkarnationslokale Sprünge auf, so braucht man für diese Prozedur keine Matrix INKL. Dies ist ein weiteres schönes Beispiel dafür, wie zusätzliche Information die Simulation vereinfachen kann.

Hiermit sind die wesentlichen Einzelheiten der Simulation von ALGOL 60-Programmen durch PAL-Programme geklärt und wir beenden damit diese Arbeit.