studien——texte

Informatik / Herausgegeben von Jürgen Perl

Rainer Hahn / Peter Stock
ELAN-Handbuch
1979, 160 Seiten, kart., DM 19,80
ISBN 3-400-00386-7

Günter Hommel / Joachim Jäckel / Stefan Jähnichen
Karl Kleine / Wilfried Koch / Kees Koster
ELAN-Sprachbeschreibung
1979, 108 Seiten, kart., DM 14,80
ISBN 3-400-00384-0

In Vorbereitung:

Jürgen Ebert
Graphenalgorithmen

Karl Heinz Sturm
Mikroprozessoren

Peter Brucker
Scheduling

ELAN-
Sprachbeschreibung

von

Günter Hommel/Joachim Jäckel
Stefan Jähnichen/Karl Kleine
Wilfried Koch/Kees Koster

Akademische Verlagsgesellschaft
Wiesbaden 1979

Adresse der Autoren:
Stefan Jähnichen
TU Berlin FB 20 Informatik
Institut für Angewandte Informatik
Ernst-Reuter-Platz 7
1000 Berlin 10

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Elan-Sprachbeschreibung / von Günter Hommel . . . –
Wiesbaden: Akademische Verlagsgesellschaft, 1979. –

(Studien-Texte: Informatik)
ISBN 3-400-00384-0

Printed in Germany/Imprimé en Allemagne
Gesamtherstellung: Decker + Wilhelm, Heusenstamm
ISSN 0172/2239
ISBN 3-400-00384-0

DIE ENTWICKLUNG DER PROGRAMMIERSPRACHE ELAN

Die erste provisorische Beschreibung einer geplanten Sprachfamilie mit dem Namen SLAN wurde Ende 1974 von C.H.A. Koster (damals an der Technischen Universitaet Berlin) vorgelegt. Die Idee war die Entwicklung verschiedener aufeinander aufbauender Sprachstufen, deren Einsatzbereich von der Ausbildung in Schulen und Universitaeten bis zur Systemprogrammierung reichen sollte.

Dieser Sprachentwurf wurde durch Erfahrungen beeinflusst, die sowohl aus der Durchfuehrung der Lehrveranstaltung Algorithmen im Informatik-Grundstudium als auch aus Fortbildungskursen mit Lehrern gewonnen wurden. In beiden von sehr verschiedenen Interessenten besuchten Veranstaltungszyklen zeigte es sich, dass die Entwicklung einer neuen Programmiersprache gerechtfertigt ist, um mit deren Hilfe mehr als bisher ueblich systematische Programmentwicklung zu unterstuetzen.

Einfluss auf den ersten Entwurf hatten weiterhin Diskussionen mit R. Hahn vom Hochschulrechenzentrum (HRZ) der Universitaet Bielefeld auf der Basis seines eigenen Sprachentwurfs (LASSO). Es wurde vereinbart, die Ausbildungsstufe von SLAN gemeinsam weiterzuentwickeln und zu implementieren. Fuer diese Implementierung wurden kurze Zeit spaeter in Bielefeld zwei Diplomarbeiten an J. Liedtke und U. Bartling vergeben, aus denen der zur Zeit verfuegbare ELAN Compiler entstand.

Die Sprachfamilie SLAN wurde 1975 in einem Werkstattgespraech ueber Schulsprachen beim Forschungs- und Entwicklungszentrum fuer objektivierte Lehr- und Lernverfahren GmbH (FEoLL) in Paderborn erstmals vorgestellt und in einem vom Bundesministerium fuer Forschung und Technologie (BMFT) gefoerderten Arbeitskreis Schulsprache (ASS) eingebracht. Schon vor der Konstituierung dieses Arbeitskreises konnte P. Heyderhoff vom Informatik-Kolleg der GMD (Gesellschaft fuer Mathematik und Datenverarbeitung) fuer eine Unterstuetzung der SLAN-Entwicklung gewonnen werden. Auch er verzichtete auf die Weiterentwicklung eines eigenen Entwurfs (SOL).

SLAN wurde daraufhin im ASS von den Mitgliedern S. Jaehnichen und W. Koch (TU Berlin) zusammen mit Herrn P. Heyderhoff vertreten. Die Berliner Vertreter entwickelten zusammen mit K. Kleine (ebenfalls TU Berlin) eine erste Sprachbeschreibung der inzwischen in ELAN umbenannten Schulsprache. Nicht zuletzt dadurch konnten auch andere Mitglieder des ASS von den Vorteilen dieser Sprache ueberzeugt werden.

Die Entwicklung und Implementierung von ELAN an der TU Berlin wurde durch die Finanzierung des Forschungsvorhabens 'Schulsprache' durch die Deutsche Forschungsgemeinschaft (DFG) wesentlich vorangetrieben. Im Rahmen dieses Vorhabens arbeiteten A. Bernatzik, G. Hommel und

J. Jaeckel an der Entwicklung und Implementierung von ELAN. Die Foerderung durch die DFG lief Ende 1977 aus, da ELAN aus dem Forschungsstadium herausgewachsen war. Aus diesem Grunde konnten die Compileraktivitäten nicht zu Ende gefuehrt werden. Sie haben aber wichtige Erkenntnisse ueber die Implementierbarkeit einiger Sprachkonstrukte eingebracht.

Noch waehrend der Taetigkeit des ASS konnte von der Universitaet Bielefeld eine Implementierung auf der Rechenanlage SIEMENS 4004 unter BS 1000 vorgelegt werden, die in der Implementierunssprache CDL programmiert ist und wegen der guten Portierungseigenschaften von CDL auch auf IBM 370 unter VM/370-CMS und auf TR 440 zur Verfuegung gestellt werden konnte. Die implementierte Sprache SLAN3B wich allerdings in einigen Punkten von ELAN ab. Inzwischen ist der Compiler auf ELAN umgestellt worden.

Der ASS empfahl in seinem Abschlussbericht, auf die Verwendung von BASIC zu verzichten, und empfahl stattdessen ELAN und PASCAL als Schulsprachen.

ELAN wurde dann in groesserem Umfang bei der Durchfuehrung der Informatik-Grundausbildung an der TU Berlin, in Lehrerfortbildungskursen an der PH Berlin und an der TU Berlin, in Kursen des HRZ der Uni Bielefeld und im Informatik-Kolleg der GMD Bonn eingesetzt. Weiterhin konnten in Bielefeld, Berlin und Bonn Lehrer gewonnen werden, die bereit waren, ELAN im Schul-Unterricht einzusetzen. Alle diese Aktivitaeten zeigten, dass ELAN fuer den Anfaengerunterricht in Informatik hervorragend geeignet ist.

Die Weiterentwicklung von ELAN aufgrund der praktischen Erfahrungen wurde weiterhin an der TU Berlin zusammen mit dem HRZ Bielefeld und der GMD Bonn durchgefuehrt. Ein Ergebnis dieser Zusammenarbeit ist die hier vorliegende Sprachbeschreibung.

ELAN ist inzwischen nicht nur auf den Rechenanlagen Siemens 4004, IBM 370 und TR 440 implementiert, sondern es gelang dem HRZ Bielefeld mit Unterstuetzung der GMD Bonn auch eine Uebertragung auf einen Mikrocomputer Z80. Dabei wurde das auf ELAN zugeschnittene Betriebssystem EUMEL entwickelt. Uebertragungen diese Systems auf Mikrocomputer anderer Hersteller sind geplant.

An der TU Berlin wird zur Zeit im Rahmen des dritten DV-Programms der Bundesregierung und in Zusammenarbeit mit der Firma Nixdorf ein Compiler fuer ELAN mit besserer Struktur und besseren Portierungseigenschaften fertiggestellt. Parallel dazu wird ein auf die Beduerfnisse des Einsatzes von ELAN abgestimmtes Benutzersystem entwickelt. Beide Komponenten sind zur Uebertragung auf beliebige Klein- und Mikrorechner vorgesehen.

INHALTSVERZEICHNIS

Kapitel 1

ENTWURFSKRITERIEN

Die Sprache ELAN (Educational LANguage) ist ein Werkzeug im Informatik-Unterricht fuer Anfaenger. Ein derartiger Unterricht soll sich immer an den Grundsaetzen systematischer Programmierungstechnik orientieren, auch wenn die Vorgehensweise im Einzelnen dem jeweiligen Ausbildungsbereich angepasst ist und daher unterschiedlich sein kann. ELAN unterstuetzt einen solchen Unterricht durch moderne Sprachkonzepte und ist unter anderem im Unterricht des Sekundarbereichs der Schulen sowie im Grundstudium der Fachhochschulen und Universitaeten einsetzbar.

Informatik-Unterricht soll am Anfang immer Algorithmenlehre sein, die anhand moeglichst verschiedenartiger aber verstaendlicher Anwendungsbeispiele aus unterschiedlichen Fachgebieten vermittelt werden kann. Die dazu verwendete Sprache muss die Moeglichkeit bieten, Algorithmen einfach und lesbar zu formulieren. Das entstandene Programm soll in moeglichst natuerlicher Formulierung den Loesungsweg widerspiegeln, was durch die Verwendung frei waehlbarer Bezeichnungen fuer Objekte und Algorithmen unterstuetzt wird. Die Konstruktion sehr unterschiedlich strukturierter Objekte aus dem jeweiligen Anwendungsbereich muss in einfacher Weise moeglich sein. Die Sprache soll moeglichst einfach in ihrer Struktur und damit in Syntax und Semantik leicht lehr- und lernbar sein. Andererseits soll durch die Anwendung weniger, ebenfalls einfacher, Kompositionsregeln fuer Algorithmen und Daten starke Flexibilitaet und damit die Anpassung an viele Anwendungsgebiete erreicht werden. Insoweit bedeutet die Einfachheit der Sprache nicht den Verzicht auf ausgepraegte Problemnaehe.

Unnoetiger Ballast soll ebenso vermieden werden, wie vielleicht wuenschenswerter, aber teurer Komfort, der die Implementierung erschwert. Andererseits ist eine zu starke Eingrenzung der Konzepte, etwa auf ausschliesslich operative Formelschreibweise, zu speziell und sicherlich nicht ohne Schwierigkeiten vermittelbar. Schliesslich muss die Sprache in dem Sinne sicher sein, dass ungewollte semantische Nebeneffekte moeglichst vermieden werden, da sie das Erreichen des Ausbildungszieles erheblich beeintraechtigen.

Die systematische Konstruktion von Algorithmen erfolgt im Unterricht mit Hilfe vorgegebener Datentypen und zugehoeriger Operationen (abstrakte Datentypen), die fuer das jeweilige Anwendungsgebiet typisch sind. Diese vorgegebenen Bausteine koennen zur Formulierung von Problemloesungen vom Programmierer mit den in der Sprache vorhandenen

Steuerkonstrukten und Datenkonstruktoren sinnvoll verknuepft werden. Wir bezeichen dieses Vorgehen als Programmierung im Kleinen im Gegensatz zur Programmierung im Grossen, der Definition von Bausteinen bei hierarchischer Zerlegung eines komplexen Problems [Koster74, Koster76].

Anfaengerunterricht wird ueberwiegend die Beherrschung der Programmierung im Kleinen als Lernziel haben. Als Methode der Konstruktion wird die Technik der schrittweisen Verfeinerung (Top-down-Methode) empfohlen. Sie besteht darin, zunaechst abstrakt formulierte Anweisungen oder abstrakte Datenbeschreibungen schrittweise zu konkretisieren und dazu die vorgegebenen Bausteine (Datentypen und zugehoerige Operationen = abstrakte Datentypen) und Konstruktionsmittel der Sprache zu benutzen.

Bei der Programmierung im Grossen werden bottom-up Bausteine fuer das jeweilige Anwendungsgebiet bereitgestellt.

Es ist wesentlich, dass die Programmiersprache sowohl die Top-down- als auch die Bottom-up-Vorgehensweise unterstuetzt, und dass das vom Lernenden angefertigte Programm die Loesungsmethode und den Loesungsweg widerspiegelt. Insbesondere sollte die Programmiersprache die Gelegenheit bieten, die bei der Entwicklung des Programms vorgenommenen Abstraktionen (Benennungen, Zusammenfassungen) explizit im Programmtext zu belassen. Dies erfordert, dass sowohl die dekomponierenden als auch die eine Abstraktionsebene formenden Bausteine (Algorithmen, Typen und Objekte) benannt werden koennen. Die eingefuehrten Namen gehoeren dabei im Gegensatz zu Kommentaren funktionell zum Programm.

Kapitel 2

DIE METHODE DER SPRACHBESCHREIBUNG

Die Sprache wird durch die Angabe der Syntax der Konstrukte und ihrer Semantik beschrieben. Dem Zweck dieser Beschreibung entsprechend gehoert dazu auch eine pragmatische Argumentation. Zwischen Semantik und Pragmatik wird nicht streng getrennt und zu ihrer Beschreibung kein formaler Kalkuel benutzt. Die Semantik der Sprachkonstrukte wird durch die Definition der Eigenschaften von Objekten, durch Aufzaehlung primitiver Operationen auf den Objekten und die Elaboration, d.h. Ausfuehrung algorithmischer Bausteine der Sprache angegeben.

Die Syntax wird mit Hilfe einer zweistufigen Grammatik (van Wijngaarden-Form [Wijngaarden75, Cleaveland77, Peck74]) definiert, ohne jedoch deren Moeglichkeiten zur vollstaendigen Beschreibung der statischen Semantik zu nutzen. Diese Beschreibungsart hat den Vorteil, eine sehr genaue und doch relativ dichte Definition der Struktur der Sprachkonstrukte zu geben.

Die Sprachkonstrukte werden ausgehend von der globalen Programmstruktur in einer Reihenfolge angegeben, die dem Vorgehen vom Allgemeinen zum Speziellen entspricht. Auf der untersten Stufe der Sprachbeschreibung stehen die Primaerausdruecke.

2.1 BEGRIFFE UND REGELN

Im folgenden wird die syntaktische Notation der zweistufigen Grammatik erlaeutert. Die Grammatik besteht aus einer Menge von Regeln, welche die einzelnen Elemente der Sprache beschreiben. Jede Regel besteht aus einer linken und einer rechten Seite, die durch das Zeichen ":" (zu lesen "ist definiert durch") voneinander getrennt sind. Die Regel wird mit dem Zeichen "." beendet. Die linke Seite gibt an, was beschrieben wird, und die rechte Seite ist die Beschreibung. Auf der rechten Seite koennen auch Bezeichnungen fur Sprachkonstrukte benutzt werden, die durch andere Regeln definiert werden.

Beispiel: Beschreibung der 'choice clause' (IF-Anweisung)

```
choice clause:
  if token, condition, then part,
  else part option, end if token.
```

Die Syntax-Regeln definieren Begiffe. Zu jedem Begriff gibt es genau eine definierende Regel, die den Begriff mit Hilfe anderer Begriffe und Symbole definiert. Fuer Symbole gibt es keine definierenden Regeln. Fuer sie existieren vielmehr Repraesentationen, die je nach der Hardware-Konfiguration, auf der ELAN implementiert ist, verschieden sein koennen. Zum unmittelbaren Codieren eines Programmtextes ist nur die Kenntnis der Repraesentation der Symbole noetig. Alle Beispiele dieser Beschreibung sind nach der in Kapitel 9 angegebenen Standard-Repraesentation codiert. Begriffe und Symbole werden mit kleinen Buchstaben bezeichnet. Dabei sind Symbole gekennzeichnet durch die Endung 'symbol'. Zwischenraeume spielen bei der Bezeichnung von Symbolen oder Begriffen keine Rolle.

Den Symbolen eng verwandt sind die Token (kenntlich an der Endung 'token'). Token sind Folgen von einem oder mehreren Symbolen. Dieser Folge kann wahlweise ein Kommentar vorangehen. Token legen (neben 'denoter's) daher fest, wo im Programmtext ein Kommentar moeglich ist. In den meisten Syntaxregeln kommen keine Symbole sondern Token vor. Der Uebergang von den Token zu den Symbolen wird in Kapitel 9 beschrieben.

Die rechte Seite einer Regel kann aus mehreren durch ";" (zu lesen als "oder") getrennten Alternativen bestehen, die alle zur Definition des links stehenden Begriffs gleichberechtigt verwendet werden koennen. Jede dieser Alternativen heisst eine "direkte Produktion" des links stehenden Begriffs. Eine Alternative besteht aus Gliedern, das sind Bezeichnungen fuer Symbole (bzw. Token) und Begriffe. Die Glieder einer Alternative sind voneinander durch "," (zu lesen als "gefolgt von") getrennt.

Beispiele:

```
declaration:
  object declaration;
  type declaration;
  shorthand declaration.

shorthand declaration:
  let token, shorthand association list.
```

Die rechte Seite der Regel 'declaration' besteht aus drei Alternativen, naemlich

```
object declaration
type declaration
shorthand declaration,
```

fuer die es ihrerseits wieder definierende Regeln gibt. Eine dieser Regeln ist 'shorthand declaration'. Sie besteht aus einer Alternative mit zwei Gliedern.

Die sukzessive Ersetzung aller in einer ausgewaehlten Alternative vorkommenden Begriffe durch genau eine ihrer direkten Produktionen fuehrt ueber eine oder mehrere Stufen zu einer sogenannten "terminalen

Produktion" des zu definierenden Begriffs. Darin kommen nur noch Symbole vor, die nach Ersetzung durch ihre Repraesentationen ein Stueck Programmtext darstellen. Weil dieser "Produktionsprozess" von oben nach unten vor sich geht, heisst diese Grammatik generativ. Sie eignet sich nicht unbedingt zur Beschreibung des umgekehrten Vorgangs, naemlich der Interpretation des Programmtextes und dessen Reduktion auf Begriffe durch einen menschlichen oder maschinellen Prozessor.

In der syntaktischen Beschreibung kommen auch Regeln vor, die nicht zur Ableitung einer terminalen Produktion benutzt werden. Der in einer solchen Regel definierte Begriff erscheint auf keiner rechten Seite einer anderen Regel. Solche Regeln sind mit einem Stern "*" gekennzeichnet. Sie dienen einer zusaetzlichen Begriffsbildung, um im Text ueber bestimmte Sachverhalte reden zu koennen.

Regeln werden abschnittsweise mit Kleinbuchstaben bezeichnet.

2.2 METABEGRIFFE

Die Anzahl der Regeln zur Beschreibung der Sprachkonstrukte ist sehr gross. Ueberdies werden an vielen Stellen einige den Konstrukten zugeordnete Attribute, wie Typ und Access, mit in die Bezeichnungen der entsprechenden Begriffe uebernommen. So gibt es z.B. nicht den Begriff des Identifizierers sondern vielmehr die verschiedenen Begriffe wie

```
int var identifier

int const identifier

real var identifier

real const identifier

        . . .
```

Da es beliebig viele Typen geben kann, ist eine Formulierung von endlich vielen Regeln fuer derartige vom Typ abhaengige Begriffsbezeichner unmoeglich. Syntaxregeln werden deswegen haeufig unter Zuhilfenahme von Abkuerzungen, den sogenannten Metabegriffen, formuliert. Metabegriffe werden mit Hilfe von Grossbuchstaben gebildet. Sie haben ihre eigene, durchaus nicht triviale Syntax. Zur Erklaerung der Metabegriffe gibt es auf einer zweiten syntaktischen Stufe sogenannte Metaregeln.

Die Form der Metaregeln ist aehnlich der der Regeln. Linke (zu definierende) Seite und rechte (definierende) Seite werden durch "::" getrennt, Meta-Alternativen durch ";", den Abschluss bildet ein Punkt. Die Glieder von Alternativen sind entweder Folgen von Kleinbuchstaben oder wieder Metabegriffe. Sie werden im Gegensatz zu den Gliedern einer Alternative einer Regel durch Zwischenraeume voneinander getrennt. Metaregeln produzieren in voellig analoger Weise zu den Regeln als terminale Produktionen Folgen von Kleinbuchstaben.

Aus einer Regel mit Metabegriffen erhaelt man eine Regel ohne Metabegriffe, wenn alle vorkommenden Metabegriffe "konsistent" ersetzt werden. Dies bedeutet, dass ein Metabegriff auf der linken und der rechten Seite einer Regel bei jedem Auftreten durch die gleiche terminale Produktion dieses Metabegriffs ersetzt wird.

An Stellen, wo eine konsistente Ersetzung fuer mehrere Vorkommen eines Metabegriffs in einer Regel verhindert werden soll, sind implizit zusaetzliche Metabegriffe gegeben, welche aus dem urspruenglichen Metabegriff bestehen, gefolgt von einer Ziffer ('DIGIT'). Diese Metabegriffe produzieren den urspruenglichen Metabegriff (z.B. MODE1:: MODE.). Eine Anwendung hierfuer findet sich z.B. in 5.5.3.a:

```
OPERATOR::
  left MODE1 right MODE2 yielding MOID PRIORITY operator;
  MODE1 yielding MOID monadic operator.
```

Nach konsistenter Ersetzung von Metabegriffen koennen Mehrdeutigkeiten auftreten, die mit Hilfe von '<' und '>' eliminiert werden. Ein Beispiel dafuer ist die Regel 2.3.g:

```
<NOTION1> or <NOTION2> :
  NOTION1;
  NOTION2.
```

Die Maechtigkeit der konsistenten Ersetzung kann an der syntaktischen Formulierung des Begriffs 'token' demonstriert werden:

Die Metaregeln

```
NOTION:: ALPHA;
         NOTION ALPHA.

ALPHA::  a; b; c; ... ; z.
```

erlauben die Produktion jeder Folge von Kleinbuchstaben, also auch der Folge 'if'.

Die Regel (mit Abkuerzungen)

```
NOTION token:
  comment sequence option, NOTION frame.
```

liefert also u.a. eine Regel ohne Abkuerzungen

```
if token:
  comment sequence option, if frame.
```

nach konsistenter Ersetzung des Metabegriffs NOTION durch die terminale Produktion 'if'.

Metaregeln werden kapitelweise mit Grossbuchstaben bezeichnet.

2.3 ALLGEMEIN VERWENDETE KONSTRUKTIONEN

Die folgenden Metaregeln verringern die Anzahl der Syntaxregeln und erhoehen (hoffentlich) ihre Lesbarkeit.

A) NOTION::
ALPHA;
ALPHA NOTION.

B) ALPHA::
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.

C) EMPTY::
.

a) NOTION option:
NOTION;
EMPTY.

b) NOTION sequence:
NOTION;
NOTION, NOTION sequence.

c) NOTION list:
NOTION;
NOTION, comma token, NOTION list.

d) NOTION suite:
period token, NOTION;
NOTION suite, period token, NOTION.

e) NOTION prelude:
NOTION, go on token;
NOTION prelude, NOTION, go on token.

f) NOTION pack:
open token, NOTION, close token.

g) <NOTION1> or <NOTION2> :
NOTION1;
NOTION2.

h) <NOTION1> and <NOTION2> :
NOTION1, NOTION2.

Kapitel 3

OBJEKTE

Die Ausfuehrung eines ELAN-Programms bewirkt die Anwendung bestimmter Aktionen auf Objekte. Objekte koennen geaendert und geliefert werden (Datenobjekte) oder angewendet werden (Prozeduren). Jede Operation auf einem Objekt kann nur innerhalb eines dem Objekt durch den Programmtext zugeordneten Bereiches ('range') erfolgen.

3.1 DATENOBJEKTE

Datenobjekte in ELAN haben

- einen Typ
- einen Wert
- einen Speicherplatz
- eine Ausdehnung.

Durch die Elaboration einer Datenobjekt-Vereinbarung wird einem Objekt ein Name und ein Typ zugeordnet; der Name wird mit einem Zugriffsrecht versehen.

Durch die Elaboration von Konstrukten der Sprache koennen Datenobjekte geliefert werden. Man sagt auch kuerzer, dass ein Wert geliefert wird. Der Wert eines Datenobjektes kann abhaengig vom Zugriffsrecht gelesen und/oder durch eine Operation geaendert werden.

3.1.1 Typen

```
A)  TYPE::
      ETYPE;
      row NUMBER of TYPE1;
      structured with FIELDS;
      abstract type.

B)  NUMBER::
      NUMERAL;
      TAG.
```

```
C)  NUMERAL::
      DIGIT;
      NUMERAL DIGIT.

D)  ETYPE::
      int; real; bool; text.

E)  FIELDS::
      FIELD;
      <FIELDS> and <FIELD>.

F)  FIELD::
      TYPE field TAG.

a)  abstract type:
      bold TAG.
```

ELAN unterscheidet folgende Typen von Datenobjekten:

- elementare Typen
- Reihungstypen
- Strukturtypen
- abstrakte Typen.

Die elementaren Typen ('ETYPE') sind fuer ELAN die Datentypen von 'int'-, 'real'-, 'bool'- und 'text'- Objekten. Fuer sie existieren klassische Denotierungen (Darstellungen von Zahlen, Wahrheitswerten und Texten im Programmtext).

Fuer Objekte vom elementaren Typ gibt es die im Anhang (Standardpakete) aufgefuehrten Operationen und Relationen.

Reihungstypen ('row NUMBER of TYPE') beschreiben eine Folge von Datenobjekten gleichen Typs (Komponenten). Die Auswahl der Elemente erfolgt durch Indizierung (Subskription). Die Numerierung der Elemente eines Objektes vom Reihungstyp beginnt bei 1 und endet bei 'NUMBER'.

Strukturtypen ('structured with FIELDS') beschreiben die Zusammenfassung von Datenobjekten nicht notwendig gleichen Typs (Komponenten). Die Auswahl von Komponenten erfolgt durch Selektion mittels eines Selektors.

Der Wert eines Objekts vom Reihungs- oder Strukturtyp ist die Folge der Werte der Komponenten in der durch den Typ festgelegten Reihenfolge.

Abstrakte Typen ('abstract type') werden in ELAN an der Stelle ihrer Benutzung wie die elementaren Typen behandelt. Abstrakte Typen sind vom Benutzer in Paketen durch entsprechende Typ-Vereinbarungen und

Herausreichen aus dem definierenden Paket formulierbar. Ausserdem sind einige wichtige abstrakte Typen in ELAN standardmaessig vorgegeben (COMPLEX, FILE, VECTOR, MATRIX).

Bei der Benutzung ausserhalb des definierenden Paketes ist die Realisierung eines Datenobjektes abstrakten Typs, d.h seine Konstruktion mit Hilfe von Datenobjekten anderer Typen und damit die Realisierung seines Wertes unbekannt. Alle Operationen mit Objekten abstrakten Typs einschliesslich ihrer Denotierung und einschlieslich des Zugriffs auf Komponenten der Objektfeinstruktur (vgl. 5.2.9) sind ausserhalb des definierten Paketes nur mit Hilfe von Typ-Bezeichnung, exportierten Prozeduren, Operatoren und Konstantenbezeichnern formulierbar.

3.1.2 Zuweisung

Die Zuweisung von Werten eines Typs an Datenobjekte desselben Typs ist eine Basisoperation. Sie wird bei der Beschreibung der Semantik von Assignation und Initialisierung benoetigt.

Der Zuweisungs-Operator (':=') ist in ELAN fuer Objekte elementaren Typs vorgegeben.

Fuer Objekte abstrakten Typs kann mit Hilfe einer Operator-Vereinbarung eine Zuweisung definiert werden. Fuer die meisten der in ELAN vordefinierten abstrakten Datentypen sind Zuweisungsoperatoren definiert.

Ist fuer Objekte vom Reihungs- oder Strukturtyp eine Zuweisung fuer jeden ihrer Komponententypen definiert, so ist damit auch eine Zuweisung fuer diese Objekte durch die Zuweisung der einzelnen Komponenten gegeben.

Die Zulaessigkeit der Zuweisung wird syntaktisch geregelt durch das Accessattribut (5.1) der entsprechenden Konstrukte.

3.2 PROZEDUREN

Prozeduren sind Objekte zur Formulierung von Teilalgorithmen. Ihre Anwendungen sind der Aufruf und die Verwendung als aktueller Parameter im Aufruf einer anderen Prozedur.

Prozeduren arbeiten auf Objekten. Zwischen ihnen koennen Objekte als Parameter ausgetauscht werden. Sie koennen als Ergebnis ihrer Elaboration ein Datenobjekt liefern.

Kapitel 4

PROGRAMMSTRUKTUR

Ein ELAN-Programm besteht aus einer Folge von Paketen, von denen das textuell letzte ('main packet') eine einfachere syntaktische Struktur hat als die uebrigen Pakete ('proper packets'). Einfache Programme bestehen nur aus einem 'main packet'.

'proper packets' dienen der Abstraktion von Daten und Algorithmen. Sie enthalten eine Folge von Vereinbarungen und anderen Konstrukten. Ein 'proper packet' stellt allen nachfolgenden Paketen diejenigen Prozeduren, Operatoren, Datentypen und Konstanten als abstrakte Bausteine zur Verfuegung, deren Bezeichner in der Schnittstelle ('interface') des Pakets aufgefuehrt werden.

Pakete sind die Hilfsmittel der Programmierung im Grossen. Sie erlauben es, dem Benutzer die Feinstruktur von Algorithmen und Daten zu verbergen.

Standardmaessig vorgegeben sind Pakete fuer die Datentypen COMPLEX, FILE, VECTOR und MATRIX (Typdefinitionen sowie die zugehoerigen Konstanten und Zugriffsoperationen, vgl. Anhang A).

a) program:
   proper packet prelude option, main packet.

b) main packet:
   packet body;
   routine body.

c) proper packet:
   packet token, TAG token, interface,
   packet body, endpacket token, TAG token.

d) interface:
   defines token, indicator list, colon token.

e) packet body:
   <unit> or <declaration>;
   packet body, go on token, <unit> or <declaration>.

Die Ausfuehrung eines ELAN-Programms besteht in der einmaligen Ausfuehrung aller durch das 'main packet' direkt oder indirekt applizierten Pakete einschliesslich des 'main packet's. Die Reihenfolge, in der die Pakete ausgefuehrt werden, ist in dieser Beschreibung nicht definiert. Es wird nur garantiert, dass ein Paket ausgefuehrt wird, bevor das erste Objekt aus diesem Paket von aussen angesprochen wird.

Die Ausfuehrung eines Paketes besteht in der Ausfuehrung aller in ihm enthaltenen Konstrukte in der Reihenfolge ihres Auftretens.

Kapitel 5

KONSTRUKTE

Ein Konstrukt ist eine Folge von Symbolen, die durch die Syntaxregeln der Sprache erzeugt werden kann. Dieses Kapitel enthaelt die Beschreibung aller Konstrukte, soweit sie nicht bereits in Kapitel 4 behandelt wurden.

5.1 TYP UND ACCESSATTRIBUT

Konstrukten, die ein Objekt liefern, wird ein Typ und ein Accessattribut zugeordnet.

```
A)  ACCESS::
      const;
      var;
      proc PARAMETY.

B)  PARAMETY::
      PARAMETERS;
      EMPTY.

C)  PARAMETERS::
      PARAMETER;
      <PARAMETER> and <PARAMETERS>.

D)  PARAMETER::
      TYPE ACCESS;
      proc PARAMETY.
```

Der Typ eines Konstrukts ist der Typ des Objektes, welches bei der Elaboration des Konstrukts geliefert wird.

Das Accessattribut legt die Zugriffsrechte fuer ein Objekt fest:

- Accessattribut 'const': Setzen des Wertes nur im Initialisierungsteil einer Konstanten-Vereinbarung und beliebiges Lesen.

- Accessattibut 'var': Wahlweises Setzen des Wertes im Initialisierungsteil einer Variablen-Vereinbarung, beliebiges Lesen und beliebiges Aendern durch Zuweisung.

- Accessattribut 'proc PARAMETY': Ausfuehren der Routine einer Prozedur.

5.2 VEREINBARUNGEN

Vereinbarungen dienen der Festlegung von Bezeichnungen (Namen oder Indikatoren) fuer Objekte, Typen und Operatoren.

Durch Datenobjekt- und Prozedurvereinbarungen wird der Bezeichnung ein Accessattribut zugeordnet.

Durch eine Datenobjektvereinbarung wird einem oder mehreren Objekten ein Typ zugeordnet.

Bei Prozedur- und Operatorvereinbarungen wird eine Routine (routine body) angegeben, die bei applizierendem Auftreten der Bezeichnung im Prozedur-Aufruf oder einer Formel ausgefuehrt wird, falls die Applikation der Bezeichnung die Vereinbarung identifiziert (vgl. Kap 7).

a) declaration:
   data object declaration;
   procedure declaration;
   operator declaration;
   type declaration;
   shorthand declaration.

Beispiele:

1) INT VAR zaehler

2) INT PROC abs (INT CONST val):
   IF val >= 0
      THEN val
      ELSE -val
   ENDIF
ENDPROC abs

3) COMPLEX OP + (COMPLEX CONST z1,z2):
   COMPLEX :(z1.re + z2.re, z1.im + z2.im)
END OP +

4) TYPE DATUM = STRUCT (INT jahr, monat, tag)

5) LET max = 100

5.2.1 Bezeichnungen

Bezeichnungen koennen sein:

- Namen fuer Datenobjekte und Prozeduren
  ('TAG token')
- Namen fuer Refinements
  ('TAG token')
- Namen fuer Selektoren von Strukturtypen
  ('TAG token')
- Namen fuer Typ-Bezeichnungen
  ('bold TAG token')
- Namen fuer Abkuerzungs-Bezeichnungen
  ('TAG token' oder 'bold TAG token')
- Namen fuer Operator-Bezeichnungen
  ('OPERATOR token' oder 'bold TAG token')

In ELAN sind einige Operator-Bezeichnungen und einige Typ-Bezeichnungen vorgegeben (z.B. fuer die elementaren Typen). Alle anderen Bezeichner sind frei waehlbar.

a) indicator:
   TAG token;
   bold TAG token;
   OPERATOR token.

'bold TAG token' werden in der gleichen Weise (z.B. durch Grosschreibung) im Programmtext repraesentiert wie reservierte Schluesselworte (vgl. 9.1). Operator-Bezeichnungen werden entweder durch Sonderzeichen oder 'bold TAG token' repraesentiert.

5.2.2 Gueltigkeitsbereiche

a) *range:
   routine body;
   packet body;
   extended range.

Der Gueltigkeitsbereich einer Vereinbarung, d.h. der Bereich, in dem eine Bezeichnung identifiziert (vgl. Kapitel 7, Identifizierung) werden kann, ist der kleinste die Vereinbarung umfassende 'range'.

Der Gueltigkeitsbereich einer Spezifikation fuer die formalen Parameter einer Prozedur ist die entsprechende Routine.

5.2.3 Erweiterung von Gueltigkeitsbereichen

Der Paketgueltigkeitsbereich von Typ-, Prozedur- und Operator-Vereinbarungen sowie Datenobjekt-Vereinbarungen mit dem Accessattribut 'const' kann ueber ein Paket hinaus durch Aufnahme der entsprechenden Bezeichnung in die Schnittstellenbeschreibung (DEFINES-Liste) des Paketes erweitert werden. Diese Bezeichnungen sind dann auch in den nachfolgenden Paketen sichtbar ('extended range').

Das Auftreten eines Konstantenbezeichners in der Schnittstellenbeschreibung eines Pakets setzt nicht das gleichzeitige Auftreten der entsprechenden Typbezeichnung dieser Konstanten in derselben Schnittstelle voraus.

5.2.4 Einschraenkung des Gueltigkeitsbereiches (Namensueberdeckung)

Wird eine Bezeichnung auf Paketebene deklariert oder benutzt, so darf die gleiche Bezeichnung in Prozedur- oder Operator-Vereinbarungen desselben Pakets nicht als formaler Parameter spezifiziert, nicht deklariert und nicht als Refinement-Name verwendet werden.

Bezeichnungen aus dem 'extended range', die im Paket nicht benutzt werden, koennen im Paket neu definiert werden (Namensueberdeckung).

Weitere Moeglichkeiten zur Namensueberdeckung, wie aus anderen Sprachen bekannt (Blockstruktur), gibt es in ELAN nicht.

Gleiche Namen fuer Objekte in parallelen 'range's (z.B. in zwei verschiedenen Prozeduren) sind keine Ueberdeckungen und somit erlaubt.

5.2.5 Deklarations-Verbote

Um die Eindeutigkeit der Identifizierung sicherzustellen, sind Vereinbarungen und Spezifikationen gleicher Bezeichnungen im selben 'range' nur in bestimmten Faellen fuer generische Objekte zulaessig (siehe 7.4, Zulaessigkeit von Vereinbarungen gleicher Bezeichnungen).

5.2.6 Datenobjekt-Vereinbarungen

a) data object declaration:
   variable declaration;
   constant declaration.

b) variable declaration:
TYPE declarer, variable token,
TYPE variable association list.

c) TYPE variable association:
TAG token, TYPE initialization part option.

d) TYPE initialization part:
initial token, TYPE const primary.

e) constant declaration:
TYPE declarer, constant token,
TYPE constant association list.

f) TYPE constant association:
TAG token, TYPE initialization part.

Bei der Elaboration einer Datenobjektvereinbarung wird der Typ eines Datenobjekts festgelegt und dem Objekt ein Name zugeordnet. Der Name erhaelt das Zugriffsrecht ('ACCESS') 'var' oder 'const'.

Bei der Elaboration einer Konstanten-Vereinbarung werden die Werte der vereinbarten Datenobjekte durch Zuweisung der von den Initialisierungsteilen gelieferten Werte gesetzt. Bei der Elaboration einer Variablen-Vereinbarung koennen die Datenobjekte optional durch Zuweisung auf den vom Initialisierungsteil gelieferten Wert gesetzt werden. Fehlt diese Initialisierung, so ist der Wert des Datenobjekts undefiniert.

Die Semantik der Initialisierung wird aus der zugehoerigen 'assignation' abgeleitet (vgl. 3.1.2, 5.5.5).

Beispiele:

1) INT VAR zaehler

2) REAL CONST pi :: 3.1415926, e :: 2.71828

3) TEXT VAR meldung :: "lernt mit elan, esst mehr obst!"

4) STRUCT (TEXT name, INT personalnummer,
INT steuerklasse, REAL gehalt) VAR personalbogen

5) ROW max COMPLEX VAR spannungstabelle

5.2.7 Prozedur-Vereinbarungen

a) procedure declaration:
   void procedure declaration;
   TYPE procedure declaration.

b) void procedure declaration:
   proc token, TAG token,
   PARAMETERS specification pack option, colon token,
   routine body, endproc token, TAG token.

c) TYPE procedure declaration:
   TYPE result, proc token, TAG token,
   PARAMETERS specification pack option, colon token,
   TYPE const routine body, endproc token, TAG token.

d) TYPE result:
   TYPE declarer.

e) <PARAMETER> and <PARAMETERS> specification:
   PARAMETER segment, comma token,
   PARAMETERS specification.

f) PARAMETER specification:
   PARAMETER segment.

g) proc PARAMETY segment:
   proc PARAMETY declarer, formal parameter list.

h) TYPE ACCESS segment:
   TYPE declarer, ACCESS declarer, formal parameter list.

g) formal parameter:
   TAG token.

h) routine body:
   skeleton, refinement suite option.

j) TYPE const routine body:
   TYPE const skeleton, refinement suite option.

k) skeleton:
   <section> or <shorthand declaration>;
   skeleton, go on token,
     <section> or <shorthand declaration>.

l) TYPE const skeleton:
   TYPE const section;
   skeleton, go on token, TYPE const section.

Die Elaboration einer Prozedurvereinbarung ist leer; sie ist insbesondere nicht die Ausfuehrung der Routine ('routine body').

Beispiele:

```
1) PROC fehler (TEXT CONST meldung):
      line (2);
      put ("+++ Fehler:"); put (meldung);
      line;
      stop
   ENDPROC fehler

2) INT PROC zeichen anzahl (TEXT CONST wort, zeichen):
      INT VAR anzahl :: 0, zaehler;
      FOR zaehler FROM 1 UPTO LENGTH wort
      REPEAT
         IF (wort SUB zaehler) = zeichen
            THEN anzahl INCR 1
         ENDIF;
         zaehler INCR 1
      ENDREPEAT;
      anzahl
   ENDPROC  zeichen anzahl

3) REAL PROC trapezregel (REAL PROC (REAL CONST) function,
                          REAL CONST from, to):
      (function (to) + function (from)) * (to-from) / 2.0
   END PROC trapezregel;
      ...
   REAL CONST x :: trapezregel (sin, 0.0, pi/2.0)
```

5.2.8 Operator-Vereinbarungen

A) ADIC :: DYADIC; TYPE ACCESS.

B) DYADIC :: left TYPE1 ACCESS1 right TYPE2 ACCESS2.

a) TYPE operator declaration:
   TYPE result, op token,
   ADIC yielding TYPE const PRIORITY operator token,
   ADIC operand specification pack, colon token,
   TYPE const routine body, endop token,
   ADIC yielding TYPE const PRIORITY operator token.

b) operator declaration:
   op token, ADIC PRIORITY operator token,
   ADIC operand specification pack, colon token,
   routine body, endop token, ADIC PRIORITY operator token.

c) left TYPE1 ACCESS1 right TYPE2 ACCESS2 operand specification:
   TYPE1 ACCESS1 and TYPE2 ACCESS2 specification.

d) left TYPE ACCESS right TYPE ACCESS operand specification:
TYPE ACCESS specification.

e) TYPE ACCESS operand specification:
TYPE ACCESS specification.

Die Elaboration einer Operator-Vereinbarung ist leer. Die mit Hilfe einer Operator-Vereinbarung vereinbarten Operatoren haben fest vorgegebene Prioritaeten (vgl. dazu 5.5.3). Die Repraesentation von Operatoren durch Sonderzeichen unterliegt einigen Restriktionen (vgl. dazu 9.2.5).

Beispiele:

```
1) INT OP ABS (INT CONST val):
      IF val >= 0
         THEN val
         ELSE -val
      ENDIF
   END OP ABS

2) REAL OP ** (REAL CONST a,b):
      IF a < 0.0
         THEN fehler ("REAL ** REAL: Basis negativ"); 0.0
         ELSE exp ( b*ln(a) )
      ENDIF
   END OP  **
```

5.2.9 Vereinbarungen zur Datenabstraktion

Als Abstraktionsmechanismen fuer Daten existieren in ELAN die Typ- und die Abkuerzungs-Vereinbarung.

Die Abkuerzungs-Vereinbarung ermoeglicht die Einfuehrung von Bezeichnungen ('TAG token' bzw. 'bold TAG token') fuer Denotationen und Typen in einem Paket oder Prozedur- bzw. Operatorskelett. Sie dient lediglich der Abkuerzung. Eine weitergehende Abstraktion (Verbergen von Feinstruktur) ist mit ihr nicht moeglich, und die eingefuehrten Bezeichnungen duerfen nicht in die Schnittstelle eines Pakets aufgenommen werden. Mit Hilfe der Typbezeichnung einer Abkuerzungsvereinbarung vereinbarte Datenobjekte haben den Typ des Deklarierers der Abkuerzungsvereinbarung.

Die Typ-Vereinbarung dagegen dient der weitergehenden Abstraktion. Durch sie wird ein neuer Typ bezeichnet. Der Deklarierer der Typvereinbarung heisst die Feinstruktur des neuen Typs. Mittels der Bezeichnung einer Typvereinbarung deklarierte Datenobjekte haben nicht den Typ der Feinstruktur. Durch Aufnahme der Bezeichnung einer Typvereinbarung in die Schnittstelle des die Typvereinbarung enthaltenen Paketes kann der neu definierte Typ den folgenden Paketen des Programmes als abstrakter

Typ zur Verfuegung gestellt werden. Die Feinstruktur eines abstrakten Typs ist ausserhalb des definierenden Paketes nicht sichtbar.

Innerhalb des definierenden Paketes ist die Feinstruktur eines durch eine Typ-Vereinbarung eingefuehrten neuen Typs sichtbar. Auf Objekte dieses Typs koennen nur innerhalb des definierenden Paketes die Konstrukte Selektion oder Subskription (vgl. 5.5.1.2, 5.5.1.3) angewendet werden, um die entsprechend der Feinstruktur realisierten Objekt-Komponenten auszuwaehlen (bei Reihungs- oder Struktur-Deklarierern). Durch Anwendung des Konkretisierers (vgl. 5.5.4.3) auf Objekte des neuen Typs innerhalb des definierenden Paketes wird das Objekt als vom Typ der Feinstruktur umbetrachtet. Mit Hilfe des Konstruktors (vgl. 5.5.4.2) koennen Objekte des abstrakten Typs denotiert werden.

Der Deklarierer einer Typvereinbarung darf die Typbezeichnung der Typvereinbarung nicht enthalten (direkte Typrekursion). Auch jede andere Konstruktion eines Deklarierers, die zu Typrekursion fuehrt, ist nicht zulassig.

a) shorthand declaration:
   let token, shorthand association list.

b) shorthand association:
   TAG token, is defined as token, ETYPE const denoter;
   bold TAG token, is defined as token, TYPE declarer.

c) type declaration:
   type token, type association list.

d) type association:
   bold TAG, is defined as token, TYPE declarer.

Typ-Vereinbarungen sind nur im Paketrumpf erlaubt, waehrend Abkuerzungs-Vereinbarungen sowohl im Paketrumpf als auch im Skelett von Routinen (in Prozedur- und Operatorvereinbarungen) auftreten koennen.

Die Elaboration einer Typ- oder Abkuerzungsvereinbarung ist leer.

Beispiele:

1) Abkuerzungsvereinbarung:

```
LET n = 50, m = 100, ZAEHLER = INT,
    r = 1.5, t = "abcdef";
 ...

LET REIHE = ROW n ZAEHLER;
 ...

REIHE VAR pseudodynamisches feld
```

2) Typvereinbarung:

```
TYPE PERSON = STRUCT (TEXT name, vorname,
                      INT alter, SEX geschlecht);
TYPE SEX = INT;
 ...

SEX CONST maennlich :: SEX:(0); weiblich :: SEX:(1);
 ...

PERSON VAR mensch :: PERSON: ("mueller", "lieschen",
                              17, weiblich)
```

5.2.10 Deklarierer

Deklarierer beschreiben Typen und Accessattribute.

a) row NUMBER of TYPE declarer:
   row token, NUMBER token, TYPE declarer.

b) structured with FIELDS declarer:
   struct token, open token, FIELDS declarator,
   close token.

c) FIELDS declarator:
   TYPE consistent FIELDS segment.

d) <FIELDS1> and <FIELDS2> declarator:
   TYPE consistent FIELDS1 segment, comma token,
   FIELDS2 declarator.

e) TYPE consistent FIELDS and TYPE field TAG segment:
   TYPE consistent FIELDS segment, comma token,
   TAG token.

f) TYPE consistent TYPE field TAG segment:
   TYPE declarer, TAG token.

g) abstract type declarer:
   bold TAG token.

h) const declarer:
   constant token.

i) var declarer:
   variable token.

j) proc PARAMETY declarer:
   proc token,
     virtual PARAMETERS object declarer part pack option.

k) virtual <PARAMETER> and <PARAMETERS> object declarer part:
virtual PARAMETER object declarer part, comma token,
virtual PARAMETERS object declarer part.

l) virtual TYPE ACCESS object declarer part:
TYPE declarer, ACCESS declarer.

5.3 REFINEMENTS

Mit dem Refinement bietet ELAN ein Sprachkonstrukt zur Unterstuetzung der Methode der schrittweisen Verfeinerung. Refinements unterstuetzen die Loesungsfindung und dokumentieren die Entwicklungsschritte.

Refinements dienen der Programmstrukturierung im Kleinen. Ihr Zweck ist das Benennen von Teilalgorithmen, die fuer die Problemloesung benoetigt werden. Dabei wird ihre Realisierung an der Stelle der Benutzung des Refinements offengelassen und erst spaeter als Teilproblem geloest. Die sukzessive Verwendung von Refinements ueber mehrere Stufen fuehrt zu einer lokalen Baumstruktur des Programms.

Refinements koennen nur innerhalb einer Routine benutzt werden. Ihr Gueltigkeitsbereich ist die umgebende Prozedur. Refinements definieren keine eigenen Datenraeume. Die rekursive Benutzung von Refinements ist nicht zugelassen. Fuer Teilalgorithmen, die auf Paket-Daten arbeiten, oder mehrfach in unterschiedlichem Kontext angewendet werden, oder deren Ablauf durch Parameter gesteuert wird, steht das Prozedurkonzept zur Verfuegung.

Refinements stehen nach der 'section' einer Routine und werden von ihr und untereinander durch Punkte getrennt ('suite').

Da ein 'main packet' im einfachsten Fall nur aus einer Routine besteht, koennen die Refinements als Werkzeug zur Programmierung im Kleinen eingefuehrt werden, ohne dass das Prozedurkonzept bekannt ist.

a) refinement:
TAG token, colon token, TYPE ACCESS section;
TAG token, colon token, section.

Beispiel:

```
{ Kaninchen vermehren sich nach dem Vorbild
  der Fibonacci-Zahlen }

INT VAR junge paare :: 1,
        fruchtbare paare :: 0,
        baby paare,
        monat :: 0;
REPEAT
   kaninchen werden geboren;
   junge werden reif;
   babies werden junge;
   monat INCR 1;
   bestand ausdrucken
UNTIL monat = 12
ENDREPEAT.

kaninchen werden geboren:
   baby paare := fruchtbare paare.

junge werden reif:
   fruchtbare paare := fruchtbare paare + junge paare.

babies werden junge:
   junge paare := baby paare.

bestand ausdrucken:
   put("Nach"); put(monat); put("Monaten gibt es");
   put(junge paare + fruchtbare paare);
   put("Kaninchenpaare."); line
```

5.4 STEUERKONSTRUKTE

Steuerkonstrukte dienen der Ablaufsteuerung eines Programms. ELAN kennt folgende Steuerkonstrukte:

- die Aneinanderreihung von Programmeinheiten,
- die Abfrage und die Abfragekette,
- die Auswahl,
- die Wiederholung,
- den Terminator.

5.4.1 Programmeinheiten

Die Bestandteile einer Programmeinheit ('section') werden ihrer textlichen Reihenfolge entsprechend nacheinander ausgefuehrt. Das Ergebnis der 'section' ist das der zuletzt ausgefuehrten 'unit'. In einer 'section' koennen Datenobjekt-Vereinbarungen und 'units' in beliebiger Reihenfolge auftreten.

a) section:
<unit> or <data object declaration>;
section, go on token, <unit> or
<data object declaration>.

b) TYPE ACCESS section:
TYPE ACCESS unit;
section, go on token, TYPE ACCESS unit.

c) TYPE ACCESS unit:
TYPE ACCESS primary;
TYPE ACCESS choice clause;
TYPE ACCESS multiple choice clause.

d) unit:
primary;
choice clause;
multiple choice clause;
repetition;
terminator.

5.4.2 Abfragen und Abfrageketten

Die 'choice clause' ist das Ausdrucksmittel zur Beschreibung der bedingten Ausfuehrung von Konstrukten.

a) choice clause:
if token, condition, then part,
else part option, end if token.

b) TYPE ACCESS choice clause:
if token, condition, TYPE ACCESS then part,
TYPE ACCESS else part, end if token.

c) then part:
then token, section.

d) TYPE ACCESS then part:
then token, TYPE ACCESS section.

e) else part:
```
else token, section;
elif token, condition, then part,
else part option.
```

f) TYPE ACCESS else part:
```
else token, TYPE ACCESS section;
elif token, condition, TYPE ACCESS then part,
TYPE ACCESS else part.
```

g) condition:
```
bool const primary.
```

Eine 'choice clause' kann nur ein Datenobjekt liefern, wenn der 'else part' vorhanden ist. Dieses Objekt hat bei jeder moeglichen Elaboration dieser 'choice clause' denselben Typ.

Das Accessattribut einer 'TYPE ACCESS choice clause', die ein Datenobjekt liefert, wird bestimmt durch Balancierung zwischen den Accessattributen der konstituierenden 'sections' (vgl. 8.2, Balancierung).

Beispiele:

1)
```
IF a < 0 THEN a := -a ENDIF
```

2)
```
IF x > 0
   THEN positive summe INCR x;
        positive anzahl INCR 1
ELIF x < 0
   THEN negative summe DECR x;
        negative anzahl INCR 1
ELSE nullanzahl INCR 1
ENDIF
```

3)
```
IF saldo > 0.0
   THEN put ("HABEN"); put (saldo)
   ELSE put ("SOLL "); put (-saldo);
        line; put ("BITTE KONTO AUFFUELLEN")
ENDIF
```

5.4.3 Auswahl

Im Gegensatz zur Abfrage mit Wahrheitswerten ('bool') wird bei der Auswahl eine ganze Zahl benutzt ('int').

a) multiple choice clause:
```
select token, case selection, of token,
case part sequence, otherwise part option,
end select token.
```

b) TYPE ACCESS multiple choice clause:
select token, case selection, of token,
TYPE ACCESS case part sequence, TYPE ACCESS otherwise part,
end select token.

c) case selection:
int const primary.

d) case part:
case prefix part, section.

e) TYPE ACCESS case part:
case prefix part, TYPE ACCESS section.

f) case prefix part:
case token, NUMBER list, colon token.

h) otherwise part:
otherwise token, section.

h) TYPE ACCESS otherwise part:
otherwise token, TYPE ACCESS section.

Wie bei der 'TYPE ACCESS choice clause' muessen bei der 'TYPE ACCESS multiple choice clause' zur Bestimmung der Accessattribute alle 'TYPE ACCESS case parts' und der 'TYPE ACCESS otherwise part' balanciert und eventuell Anpassungen vorgenommen werden (vgl. 8.2, Balancierung).

Beispiel:

```
INT VAR monat;
LET januar = 1,
    februar = 2,
    ... .
    dezember = 12;
    ...
INT VAR tage;
tage := anzahl der tage im monat;
    ...
anzahl der tage im monat:
   SELECT monat OF
      CASE februar:
         IF schaltjahr
            THEN 29
            ELSE 28
         ENDIF
      CASE april, juni, september, november: 30
      OTHERWISE 31
   ENDSELECT
```

5.4.4 Wiederholungen

Die Wiederholung dient zur mehrfachen Ausfuehrung eines Programmstueckes. Die Anzahl der Durchlaeufe kann sowohl durch Zaehlen mit einer Integer-Variablen als auch durch Abbruchbedingungen festgelegt werden.

a) repetition:
   for part option, while part option, repeat token,
   repetition part option, until part option,
   end repeat token.

b) for part:
   for token, int var identifier, from token,
   from part, direction, to part.

c) from part:
   int const primary.

d) direction:
   upto token;
   downto token.

e) to part:
   int const primary.

f) while part:
   while token, condition.

g) repetition part:
   section.

h) until part:
   until token, condition.

In einer Zaehlschleife (Benutzung des 'for part') laeuft die Integer-Variable vom Startwert ('from part') in Schritten von 1 oder -1 je nach 'direction' bis zum Erreichen des Endwertes ('to part').

Nach Verlassen einer Zaehlschleife ist der Wert der Laufvariablen undefiniert, es sei denn, die Zaehlschleife wurde vorzeitig durch Elaboration eines 'while part' oder 'until part' oder durch Elaboration eines Terminators verlassen.

Abbruchbedingungen koennen entweder mit WHILE- oder UNTIL-Konstruktionen vor bzw. nach dem zu wiederholenden Programmteil ('repetition part') oder innerhalb desselben mit Hilfe eines Terminators (5.4.4) formuliert werden.

Die Elaboration einer Wiederholung besteht aus der einmaligen Elaboration des 'from part' und des 'to part' (in dieser Reihenfolge)

und der folgenden, durch 'for part', 'while part' und 'until part' kontrollierten mehrfachen Elaboration des 'repetition part'. Dabei werden die durch den 'for part' kontrollierte Zuweisung an die Laufvariable und der 'while part' (in dieser Reihenfolge) vor jeder Wiederholung des 'repetition part' und der 'until part' jeweils nach der Wiederholung des 'repetition part' elaboriert.

Die Elaboration der Wiederholung liefert kein Objekt.

Beispiele:

1) { Preisberechnung fuer ein Telegramm. Das Telegramm besteht aus einer sechsstelligen Kennzahl, gefolgt von Worten einschliesslich dem Trennsymbol 'stop'. Zwei 'stop'-Symbole markieren das Telegrammende. }

```
LET zaehlwortpreis = 0.60,
    zaehlwortlaenge =  12;
lies kennzahl;
initialisiere preis;
initialisiere fenster;
WHILE NOT telegrammende erreicht
REPEAT
   verarbeite ein wort;
   verschiebe fenster
ENDREPEAT;
drucke den preis.

lies kennzahl:
   INT VAR kennzahl;
   get (kennzahl);
   IF kennzahl < 100 000 OR kennzahl >= 1 000 000
      THEN put ("Falsche Kennzahl: "+text(kennzahl));
           stop
   ENDIF.

initialisiere preis:
   REAL VAR telegrammpreis :: 0.0.

initialisiere fenster:
   TEXT VAR dieses wort, naechstes wort;
   get (dieses wort); get (naechstes wort).

telegrammende erreicht:
   dieses wort = "stop" AND naechstes wort = "stop".

verarbeite ein wort:
   telegrammpreis INCR wortpreis.

wortpreis:
   real (LENGTH dieses wort DIV zaehlwortlaenge + 1)
   * zaehlwortpreis.
```

```
verschiebe fenster:
   dieses wort := naechstes wort;
   get (naechstes wort).

drucke den preis:
   put ("Telegramm kostet:");
   put (telegrammpreis)
```

2) { Nullsetzen eines Vektors }

```
LET size = 100;
ROW size INT VAR vector;
INT VAR index;
FOR index FROM 1 UPTO size
REPEAT
   vector [index] := 0
ENDREPEAT
   ...
```

5.4.5 Terminatoren

Mit Hilfe eines Terminators (LEAVE) kann die Elaboration einer Prozedur, eines Operators oder eines Refinements beendet werden. Dabei wird gegebenenfalls ein Ergebniswert entsprechend dem Resultattyp der zu terminierenden Prozedur bzw. des zu terminierenden Operators oder entsprechend dem geforderten Ergebnistyp der Refinementanwendung angegeben.

a) terminator:
   leave token, terminated clause, with part option.

b) terminated clause:
   TAG token;
   OPERATOR token.

c) with part:
   with token, terminator result.

d) terminator result:
   TYPE ACCESS primary.

Der Terminator muss textuell nicht in der 'section' des Refinements stehen, dessen Elaboration er beendet. Wird ein Refinement R ein- oder mehrfach angewendet und enthaelt es einen Terminator fuer ein anderes Refinement Q, so darf R nur innerhalb von Q (oder anderer innerhalb von Q angewendeter Refinements) angewendet werden.

Soll ein Terminator die Elaboration eines Prozedur- oder Operatoraufrufs beenden, so kann er an jeder Stelle innerhalb des Rumpfes der aufgerufenen Operation stehen.

Die Elaboration eines Terminators liefert kein Objekt.

Beispiel:

```
INT PROC suche tabellenindex (TEXT CONST wort):

   INT VAR index :: tabellenanfang;
   WHILE noch tabellenelemente
   REPEAT
      untersuche element
   ENDREPEAT;
   0.

noch tabellenelemente:
   index <= tabellenende.

untersuche element:
   IF wort gefunden
      THEN LEAVE suche tabellenindex WITH index
      ELSE index INCR 1
   ENDIF.

wort gefunden:
   tabelle [index] = wort

ENDPROC suche tabellenindex
```

5.5 PRIMAERAUSDRUECKE

Primaerausdruecke sind die algorithmischen Grundbausteine der Sprache. Die Elaboration eines Primaerausdrucks kann ein Objekt liefern.

a)
```
primary:
  primary pack;
  call;
  refinement application;
  PRIORITY formula.
```

b)
```
TYPE ACCESS primary:
  TYPE ACCESS primary pack;
  TYPE ACCESS identifier;
  TYPE ACCESS subscription;
  TYPE ACCESS selection;
  TYPE ACCESS call;
  TYPE ACCESS refinement application;
  TYPE ACCESS PRIORITY formula;
  TYPE ACCESS interpreter;
  TYPE ACCESS denoter.
```

5.5.1 Grund-Algorithmen

Die Grund-Algorithmen sind der Zugriff auf einen Identifizierer, die Subskription, die Selektion und der Prozeduraufruf.

5.5.1.1 Bezeichner

a) TYPE ACCESS identifier:
   TAG token.

b) proc PARAMETY identifier:
   TAG token.

Ein Bezeichner ('identifier') regelt den Zugriff auf das Objekt, dem er durch eine Vereinbarung zugeordnet ist. Bezeichner sind ausschliesslich im Gueltigkeitsbereich der zugehoerigen Vereinbarung ('range', vgl. 5.2.2) verwendbar. Welche Vereinbarung ein Bezeichner "identifiziert", wird in Kapitel 7 (Identifizierung) beschrieben.

Die Elaboration eines Bezeichners liefert das mit der Vereinbarung verbundene Objekt. Der Bezeichner hat Typ und Accessattribut der Vereinbarung.

Bezeichner mit Accessattribut 'proc PARAMETY' koennen entweder in einem 'call' oder als aktuelle Parameter eines Prozeduraufrufs auftreten. 'proc PARAMETY identifier' haben keinen Typ; sie identifizieren Prozeduren, deren Aufruf keinen Wert liefert.

5.5.1.2 Subskription

Die Subskription ermoeglicht den Zugriff auf Elemente von Reihen.

a) TYPE ACCESS subscription:
   row NUMBER of TYPE ACCESS primary,
   sub token, index, bus token.

b) index:
   int const primary.

Die Elaboration einer Subskription beginnt mit der Elaboration des konstituierenden Primaerausdrucks und des Index in einer nicht festgelegten Reihenfolge. Der Primaerausdruck muss ein Datenobjekt vom Reihungstyp liefern. Die Subskription liefert diejenige Komponente, deren Nummer gleich dem Wert des Index ist und hat deren Typ. Das Accessattribut der Subskription ist das des konstituierenden Primaerausdrucks.

Beispiel:

```
ROW 12 TEXT monate;
     ...
monate [1] := "Januar"
```

5.5.1.3 Selektion

Die Selektion ermoeglicht den Zugriff auf die Komponenten ('fields') von Datenobjekten vom Strukturtyp.

a) TYPE ACCESS selection:
   structured with FIELDS primay,
   selection token, selector.

b) selector:
   TAG token.

Die Elaboration einer Selektion besteht aus der Elaboration des konstituierenden Primaerausdrucks, der ein Datenobjekt vom Strukturtyp liefern muss. Der Selektor identifiziert genau eine Komponente dieses Strukturtyps. Die Selektion liefert diese Komponente und hat deren Typ. Das Accessattribut der Selektion ist das des konstituierenden Primaerausdrucks.

Beispiel:

```
STRUCT (TEXT name, INT alter) VAR person;
    ...
person.name := "Meier"
```

5.5.1.4 Prozedur-Aufruf

a) TYPE const call:
   TYPE proc PARAMETY identifier,
   actual parameter list pack option.

b) call:
   proc PARAMETY identifier,
   actual parameter list pack option.

c) actual parameter:
   TYPE ACCESS primary;
   proc PARAMETY identifier.

Der konstituierende Identifizierer eines Prozedur-Aufrufs identifiziert zusammen mit den Typen der Parameter genau eine Prozedurvereinbarung

(vgl. 7). Der Aufruf einer Prozedur wird in folgenden Schritten elaboriert: (Falls die Prozedur keine Parameter hat, entfallen die Schritte 1 und 2.)

1. Alle aktuellen Parameter werden in nicht definierter Reihenfolge elaboriert. Die Elaboration eines aktuellen Parameters liefert ein Objekt mit dem Accessattribut der entsprechenden Parameterspezifikation der Prozedurvereinbarung. Das Accessattribut des auf der Position eines aktuellen Parameters stehenden 'primary' kann unter Umstaenden vom Accessattribut der Parameterspezifikation verschieden sein. Zulaessige Anpassungen werden in Kapitel 7 und 8 beschrieben.

2. Das bei der Elaboration eines aktuellen Parameters gelieferte Objekt wird fuer die Dauer der Elaboration der Routine (zusaetzlich) durch den entsprechenden formalen Parameter bezeichnet, und der Zugriff darauf durch das Accessattribut der Parameterspezifikation festgelegt.

3. Die Routine der Prozedurvereinbarung wird elaboriert. Diese Elaboration besteht aus der Elaboration des Skeletts ('skeleton') der Routine.

Die Art der Prozedur-Vereinbarung ('TYPE procedure' oder 'procedure') legt fest, ob die Routine einen Wert liefert oder nicht. Der Wert der Routine ist entweder der Wert des konstituierenden 'skeleton' oder der Wert eines seine Elaboration beendenden Terminators.

Der Typ des Aufrufs einer 'TYPE procedure' ist derjenige des 'result' der Vereinbarung, das Accessattribut immer 'const'. Der Wert eines Prozedur-Aufrufs ist der Wert, den die Elaboration der korrespondierenden Routine liefert.

Prozeduren koennen rekursiv aufgerufen werden.

5.5.1.5 Zuweisung

Die Zuweisung ist in ELAN durch eine dyadische Formel (vgl. 5.5.3), die 'assignation', formulierbar.

a)
```
*assignation:
  prio 1 formula.
```

5.5.2 Refinement-Anwendung

a) refinement application:
TAG token.

b) TYPE ACCESS refinement application:
TAG token.

Jede Refinement-Anwendung identifiziert genau ein Refinement aus derselben Routine.

Die Elaboration einer Refinement-Anwendung ist die Elaboration der section des identifizierten Refinements.

Typ und Accessrecht der Refinement-Anwendung ergeben sich aus dem TYPE ACCESS der konstituierenden 'section' des Refinements (vgl. 5.3).

5.5.3 Formeln

A) MODE::
TYPE ACCESS.

B) MOID::
TYPE ACCESS;
EMPTY.

C) OPERATOR::
left MODE1 right MODE2 yielding MOID PRIORITY operator;
MODE1 yielding MOID monadic operator.

D) PRIORITY::
prio i;
PRIORITY i.

a) MOID PRIORITY formula:
MODE1 PRIORITY operand,
left MODE1 right MODE2
yielding MOID PRIORITY operator token,
MODE2 PRIORITY i operand;
MOID monadic operand.

b) MODE PRIORITY operand:
MODE PRIORITY formula;
MODE PRIORITY i operand.

c) MOID monadic operand:
MODE1 yielding MOID monadic operator token,
MODE1 monadic operand;
MOID primary.

d) left MODE1 right MODE2 yielding EMPTY prio i operator:
   becomes.

e) left MODE1 right MODE2 yielding MOID prio ii operator:
   bold TAG;
   {other special}.
   (vgl. 9.2.5, Symbole fuer Operatoren)

f) left MODE1 right MODE2 yielding MODE prio iii operator:
   or.

g) left MODE1 right MODE2 yielding MODE prio iii i operator:
   and.

h) left MODE1 right MODE2 yielding MODE prio iii ii operator:
   equal; notequal;
   less; lessequal;
   greater; greaterequal.

i) left MODE1 right MODE2 yielding MODE prio iii iii operator:
   plus; minus.

j) left MODE1 right MODE2 yielding MODE prio iii iii i operator:
   asterix; divide; intdiv; modulo.

k) left MODE1 right MODE2 yielding MODE prio iii iii ii operator:
   obelix.

l) prio iii iii iii:
   monadic.

m) MODE1 yielding MOID monadic operator:
   not; plus; minus; bold TAG;
   {other special}.
   (vgl. 9.2.5, Symbole fuer Operatoren)

Jedes Operatorzeichen hat genau eine Prioritaet und identifiziert zusammen mit den Typen der Operanden (bzw. dem Typ des Operanden) genau eine Operator-Vereinbarung.

Ein Operatorzeichen hat stets die gleiche, vorgegebene und nicht aenderbare Prioritaet. Es kann aufgrund des Prinzips der Generizitaet (vgl. Kapitel 7) als Bezeichnung fuer mehrere Operatoren dienen.

Es gibt neun Prioritaeten (in der Syntax durch eine Folge von Kleinbuchstaben 'i' beschrieben). Die niedrigste Prioritaet haben die Zuweisungsoperatoren (mit dem Zeichen 'becomes' ":=", vgl. Abschnitt 3.1.2).

Der Operator einer dyadischen Formel ist stets der der niedrigsten Prioritaet. Seine Operanden koennen wiederum Formeln sein, wobei der

linke Operand die gleiche Prioritaet wie der Operator hat. Daraus ergibt sich die gewohnte Links-nach-Rechts-Abarbeitung bei der Aufeinanderfolge dyadischer Operatoren gleicher Prioritaet. In einer monadischen Formel mit mehreren Operatoren ist entsprechend der am weitesten links stehende Operator der Operator der Formel.

Die Elaboration einer Formel beginnt mit der Elaboration der Operanden in nicht definierter Reihenfolge. Die Beziehung zwischen formalen Parametern der Operatorvereinbarung und entsprechenden Operanden der identifizierenden Formel werden nach demselben Mechanismus geregelt wie beim Prozeduraufruf. Nach Elaboration der Operanden wird die Routine der Operatorvereinbarung elaboriert (vgl. 5.5.1.4).

Der Typ einer Formel wird durch die Operator-Vereinbarung festgelegt, das Accessattribut ist 'const'. Das von der Formel gelieferte Datenobjekt ist das nach Elaboration der Routine der Operator-Vereinbarung gelieferte Objekt.

Analog zu Prozeduren koennen auch Operatoren rekursiv benutzt werden.

5.5.4 Typ-Interpretierer

a) TYPE ACCESS interpreter:
   TYPE ACCESS row display;
   TYPE ACCESS constructor;
   TYPE ACCESS concretizer.

Typ-Interpretierer dienen zur Komposition von Objekten und zur Abstraktionsbildung bzw. umgekehrt zum Zugriff auf die Feinstruktur von Objekten abstrakten Typs.

5.5.4.1 Display

a) row NUMBER of TYPE const row display:
   sub token, TYPE component list, bus token.

b) TYPE component:
   TYPE const primary.

Die Elaboration eines 'row display' besteht in dem Aufbau eines (neuen) Datenobjekts vom Reihungstyp. (Die konstituierenden Primaerausdruecke werden in nicht festgelegter Reihenfolge elaboriert, wobei die gelieferten Werte alle vom gleichen Typ sein muessen. Sie werden den entsprechenden Komponenten des neuen Objektes zugewiesen. Alle konstituierenden Primaerausdruecke haben das Accessattribut 'const'.

Das 'row display' liefert das neue Datenobjekt vom Reihungstyp. Sein Accessattribut ist 'const'.

Beispiel:

```
ROW 5 INT CONST tabulator :: [1, 10, 16, 35, 71]
```

Der Zweck des 'row display' ist die bequeme Denotierung von Reihungen.

5.5.4.2 Konstruktor

a) TYPE const constructor:
   TYPE declarer, constructor open token,
   constituent list, close token.

b) constituent:
   TYPE const primary.

Konstruktoren dienen der Denotierung von Datenobjekten beliebigen Typs. Sie sind nur verwendbar, wenn die Feinstruktur (Typ, Anzahl und Anordnung der direkten Komponenten) bekannt ist.

Konstruktoren werden benoetigt, wenn ein Objekt abstrakten Typs innerhalb des diesen Typ definierenden Pakets denotiert werden soll.

Die Elemente der 'constituent list' des Konstruktors entsprechen in der formulierten Reihenfolge den durch den (konkreten) Deklarierer definierten Feldkomponenten in der dort gegebenen Reihenfolge.

Die Elaboration eines Konstruktors besteht im Aufbau eines (neuen) Datenobjektes vom Typ des konstituierenden Deklarierers. Die konstituierenden Primaerausdrucke werden in hier nicht festgelegter Reihenfolge elaboriert, wobei die gelieferten Werte vom Typ der entsprechenden Komponente des konstituierenden Deklarierers oder seine Feinstruktur sein muessen. Ist nur ein konstituierender Primaerausdruck vorhanden, so muss er entweder vom Typ des konstituierenden Deklarierers oder vom Typ seiner Feinstruktur sein. Alle konstituierenden Primaerausdruecke haben das Accessattribut 'const'.

Der Konstruktor liefert das neue Datenobjekt vom angegebenen Typ und hat das Accessattribut 'const'.

Beispiele:

```
1) LET KARTEIKARTE = STRUCT(TEXT name, REAL gehalt);
     ...

   KARTEIKARTE VAR karte :: KARTEIKARTE: ("meier", 3720.20)
```

2) PACKET complexrechnung
DEFINES ... . COMPL, complexnull, complexi, ... :
...

```
TYPE COMPL = STRUCT(REAL reell, imag);
COMPL CONST complexnull :: COMPL:(0.0, 0.0),
           complexi    :: COMPL:(0.0, 1.0);
```

...

ENDPACKET complexrechnung

Ausserhalb dieses Pakets ist eine direkte Denotierung fuer COMPL-Objekte mit Hilfe des Konstruktors nicht moeglich. Ist eine Denotierung ausserhalb dieses Paketes erwuenscht, so muss das Paket auch eine entsprechende Denotierungsprozedur zur Verfuegung stellen.

```
3) LET PERSON = STRUCT (TEXT name,
                        ELTERN eltern,
                        REAL groesse),
     ELTERN = STRUCT (TEXT vater, mutter);
   ...

PERSON:("mueller", ELTERN:("otto", "ida"), 1.83)
```

5.5.4.3 Konkretisierer

a) TYPE1 ACCESS concretizer:
concr token, TYPE2 ACCESS primary pack.

Der Konkretisierer ermoeglicht den Zugriff auf die Feinstruktur von Objekten abstrakten Typs in dem sie definierenden Paket. Mit der Typ-Vereinbarung (5.2.6) wird von der Feinstruktur, der Realisierung eines entsprechend vereinbarten Objekts, abstrahiert. Um in demselben Paket die notwendigen Prozeduren zur Manipulation von Objekten dieses Typs zu schreiben, wird aber auch Zugriff auf deren Feinstruktur benoetigt. Dabei gelten die folgenden Regeln:

- Ein Datenobjekt abstrakten Typs behaelt (auch bei Verwendung als Parameter in Prozedur- (5.5.1.4) oder Operator-Aufruf (Formel, 5.5.3)) stets diesen abstrakten Typ. Dies gilt auch innerhalb des definierenden Pakets.

- Der Konkretisierer erzeugt, angewendet auf ein Datenobjekt abstrakten Typs ein Datenobjekt mit dem Typ der Feinstruktur des abstrakten Typs. Dies ist ein einfaches typmaessiges Umbetrachten des Objekts ohne Aenderung seiner anderen Eigenschaften. Die Anwendung eines Konkretisierers fuehrt jeweils um genau eine Abstraktionsstufe auf die Realisierung des abstrakten Typs zurueck. Damit kann bei Verwendung des

Konkretisierers die jeweils gewuenschte Identifizierung von Prozeduren oder Operatoren sichergestellt werden.

- Ein Typuebergang vom abstrakten zum konkreten Typ erfolgt automatisch bei Anwendung der Grundoperationen Subskription und Selektion auf Objekte abstrakten Typs im definierenden Paket (implizite Konkretisierung).

Die Elaboration eines Konkretisierers besteht aus der Elaboration des Primaerausdruckes.

Der Typ des Konkretisierers ist die Feinstruktur des Typs des konstituierenden Primaerausdrucks. Das Accessattribut ist das des Primaerausdrucks.

Beispiel:

```
TYPE DATUM = INT;
{ gepackte Darstellung eines Datums }
  ...
DATUM PROC datum (INT CONST jahr, monat, tag):
   {Beispiel einer Denotierungsprozedur}
   DATUM: (jahr * 10000 + monat * 100 + tag)
ENDPROC datum;
  ...
PROC put (DATUM CONST d):
   put( CONCR(d) )
ENDPROC put
```

Der Aufruf von

```
put (d)
```

anstelle von

```
put(CONCR(d))
```

ist zwar syntaktisch korrekt, leistet jedoch nicht die beabsichtigte Ausgabe von d. Da d ein DATUM ist, wuerde das put der Routine dieselbe Prozedur put identifizieren, die gerade vereinbart wird, und somit bei Ausfuehrung in endlose Rekursion geraten.

Kapitel 6

DENOTIERUNGEN

Denotierungen dienen der Darstellung von Werten elementaren Typs im Programmtext.

a) ETYPE const denoter:
   comment sequence option, ETYPE denotation;
   comment sequence option, TAG token.

b) int denotation:
   numeral.

c) real denotation:
   numeral, period symbol, numeral, exponent option.

d) exponent:
   exponent symbol, sign option, numeral.

e) sign:
   plus symbol;
   minus symbol.

f) numeral:
   digit CYPHER symbol;
   numeral, digit CYPHER symbol.

g) bool denotation:
   true symbol;
   false symbol.

h) text denotation:
   quote symbol, text item sequence option, quote symbol.

i) text item:
   {Any character symbol, but not a quote symbol};
   quote image.

j) quote image:
   quote symbol, quote symbol.

Der Typ einer Denotierung ist implizit durch ihre Repraesentation gegeben. Das Accessattribut ist immer 'const'.

Die Elaboration einer Denotierung liefert den dargestellten Wert.

Kapitel 7

IDENTIFIZIERUNG

Bezeichnungen in einem ELAN-Programm stehen entweder in definierender oder in applizierender Position. Definierendes Auftreten von Bezeichnungen erfolgt in einer Vereinbarung, einer Parameter-Spezifikation oder einem Refinement. Bezeichnungen an anderen Stellen im Programmtext stehen in applizierender Position.

Jede Bezeichnung in applizierender Position identifiziert eine zugehoerige Bezeichnung in definierender Position.

7.1 EINFACHE IDENTIFIZIERUNG

Jede in applizierender Position auftretende Bezeichnung eines Datenobjekts, eines Selektors, einer Abkuerzung oder eines abstrakten Typs identifiziert genau eine definierende Bezeichnung und damit eine Vereinbarung.

Jede Refinement-Anwendung in einer Routine identifiziert genau ein Refinement aus derselben Routine.

7.2 GENERISCHE IDENTIFIZIERUNG

Gleiche Bezeichnungen von Prozeduren oder Operatoren in applizierender Position koennen - je nach Kontext - verschiedene Prozeduren bzw. Operatoren identifizieren. Diese Moeglichkeit wird im mathematischen Kalkuel benutzt, z.B. bei der Definition algebraischer Infix-Verknuepfungsoperatoren fuer sehr verschiedene Objekte (Addition fuer ganze Zahlen, reelle Zahlen, Matrizen, etc.). In ELAN wird dieses Prinzip der Generizitaet auch auf Prozeduren ausgedehnt.

Ein Operator bzw. ein Prozedurbezeichner identifiziert eine entsprechende (moeglicherweise nicht in ELAN formulierte) Vereinbarung, wenn nicht nur die definierende Bezeichnung die gleiche ist, sondern auch die Typen aller einander entsprechenden formalen und aktuellen Parameter (bzw. Operanden) gleich sind. Durch Anpassungsoperationen wird sichergestellt, dass einige fuer den Benutzer komfortable Angleichungen des Accessattributs zwischen aktuellen und formalen Parametern vorgenommen werden koennen (siehe Kapitel 8).

Generische Objekte duerfen nicht als aktuelle Parameter auftreten.

Die eindeutige Identifizierung generischer Objekte wird gewaehrleistet durch Deklarationsverbote (7.4) und Namensueberdeckung (5.2.4).

7.3 IDENTIFIZIERUNG UND ANPASSUNGEN

Der Identifizierungsvorgang fuer Prozeduren und Operatoren ist also nur bei gleichzeitiger Betrachtung moeglicher Anpassungen des Accessattributs der aktuellen an die entsprechenden formalen Parameter definierbar.

Folgende Regeln beschreiben den Identifizierungsvorgang:

1. Die Prozedur- bzw. Operatorbezeichnungen des Aufrufs und der identifizierten Prozedur bzw. des identifizierten Operators sind gleich.

2. Die Anzahl der aktuellen Parameter bzw. Operanden ist gleich der Anzahl der formalen Parameter.

3. Fuer jeden Parameter gilt, dass der Typ des aktuellen Parameters bzw. Operanden gleich dem Typ des formalen Parameters ist und das Accessattribut uebereinstimmt oder an das des formalen Parameters anpassbar ist. Moegliche Anpassungen sind 'consting' und 'deproceduring' (vgl. 8.1, Anpassungsoperationen).

7.4 ZULAESSIGKEIT VON VEREINBARUNGEN GLEICHER BEZEICHNUNGEN

Mehrere Vereinbarungen, Spezifikationen oder Refinements mit gleicher Bezeichnung im selben Gueltigkeitsbereich sind nur fuer Prozeduren bzw. Operatoren zulaessig, wenn diese sich in der Parameteranzahl und/oder durch die Typen der Parameter unterscheiden.

Dadurch wird fuer jede Applikation einer Bezeichnung die Identifizierung eindeutig.

Dabei sind die beschriebenen Einschraenkungen des Gueltigkeitsbereiches (vgl. 5.2.4) zu beachten.

Kapitel 8

ANPASSUNGEN

An allen Stellen dieser Sprachbeschreibung, an denen vom Accessattribut eines Konstruktes gesprochen wird, ist das Accessattribut nach Anwendung entsprechender Anpassungsoperationen gemeint. Dies gilt insbesondere auch fuer die Syntaxregeln.

8.1 ANPASSUNGSOPERATIONEN

Zur Aenderung des Accessattributs eines Konstrukts gibt es zwei Anpassungsoperationen:

- der Uebergang vom Accessattribut 'var' nach 'const' (consting),
- der Uebergang vom Accessattribut 'proc PARAMETY' nach 'const' (deproceduring), wobei die entsprechende Routine ausgefuehrt wird.

8.2 BALANCIERUNG

Bei Balancierung wird das eindeutig feststellbare Ziel-Accessattribut eines Konstruktes aus den Ausgangs-Accessattributen von einem oder mehreren Konstrukten bestimmt. So wird zum Beispiel das Accessattribut einer Abfragekette durch die Accessattribute der 'then part's und des 'else part' festgelegt. Sind die Accessattribute der 'then part's und des 'else part' alle 'var', so ist das Accessattribut der Abfragekette ebenfalls 'var'. Ist eines dieser Accessattribute 'const', so ist das Accessattribut der Abfragekette 'const' und ueberall dort, wo das Ausgangs-Accessattribut 'var' ist, wird "consting" durchgefuehrt.

Balancierung ist in folgenden Faellen erforderlich:

- zwischen der 'section' eines Refinements, das einen Wert liefern soll und dem 'terminator result' eines Terminators, mit dem dieses Refinement beendet werden kann,
- zwischen den 'then part's und dem 'else part' einer Abfrage bzw. Abfragekette,
- zwischen den 'case part's und dem 'otherwise part' einer Auswahl.

Kapitel 9

REPRAESENTATIONEN

Aus einer terminalen Produktion des Begriffes 'program' entsteht wie in Abschnitt 2.1 geschildert nach Ersetzung aller Symbole (terminale Produktionen, die auf 'symbol' enden) durch die in Abschnitt 9.2 gegebenen Hardware - Repraesentationen ein ELAN-Programm.

9.1 TOKEN UND SYMBOLE

Beim Uebergang von den Token zu den Symbolen wird das Einfuegen von Kommentaren geregelt. Kommentare koennen ueberall im Programmtext auftreten, ausser in reservierten Symbolen, Bezeichnern und Denotierungen.

Bei der Konzipierung der Hardware-Repraesentationen wurde vom ISO-7-Bit(ASCII) Zeichensatz mit Gross/Kleinschreibung ausgegangen. Die Darstellung der 'RESERVED symbols' und 'bold TAGs' erfolgt durch Grossbuchstaben und der 'TAGs' durch Kleinbuchstaben. Dies ist die Standard-Repraesentation fuer ELAN. An Installationen, an denen Gross/Kleinschreibung nicht zur Verfuegung steht, koennen 'RESERVED symbols' und 'bold TAGs' auch durch Hochkommata als Begrenzer dargestellt werden.

```
A) TAG::
     LETTER;
     TAG LETTER;
     TAG DIGIT.

B) LETTER::
     letter ALPHA.

C) DIGIT::
     digit CYPHER.

D) CYPHER::
     zero; one; two; three; four;
     five; six; seven; eight; nine.
```

```
E)  RESERVED::
      ETYPE; type; struct; row; let; var; const;
      packet; endpacket; defines; proc; endproc;
      if; then; else; elif; endif;
      select; of; case; otherwise; endselect;
      repeat; while; until; endrepeat;
      for; from; upto; downto;
      leave; with; concr;
      op; endop;
      OPERATOR.


a)  number:
      NUMBER token;
      digit zero sequence token.

b)  NOTION token:
      comment sequence option, NOTION frame.

c)  comment:
      comment open symbol, comment item sequence option,
      comment close symbol.

d)  comment item:
      comment;
      NOTION symbol
      {but not a 'comment open symbol'
       or a 'comment close symbol'}

e)  RESERVED frame:
      RESERVED symbol.

f)  LETTER frame:
      LETTER symbol.

g)  DIGIT frame:
      DIGIT symbol.

h)  TAG LETTER frame:
      TAG frame, LETTER symbol.

i)  TAG DIGIT frame:
      TAG frame, DIGIT symbol.

j)  bold TAG LETTER frame:
      bold TAG frame, bold LETTER symbol.

k)  NUMBER DIGIT frame:
      NUMBER frame, DIGIT symbol.
```

9.2 LISTE DER REPRAESENTATIONEN

Im folgenden werden die Repraesentationen durch Auflistung gegeben. Alternative Repraesentationen fuer ein Symbol werden voneinander durch Kommata getrennt.

9.2.1 Symbole fuer Bezeichner

Symbol	Repraesentation(en)
letter a symbol	a
letter b symbol	b
letter c symbol	c
letter d symbol	d
letter e symbol	e
letter f symbol	f
letter g symbol	g
letter h symbol	h
letter i symbol	i
letter j symbol	j
letter k symbol	k
letter l symbol	l
letter m symbol	m
letter n symbol	n
letter o symbol	o
letter p symbol	p
letter q symbol	q
letter r symbol	r
letter s symbol	s
letter t symbol	t
letter u symbol	u
letter v symbol	v
letter w symbol	w
letter x symbol	x
letter y symbol	y
letter z symbol	z
bold letter a symbol	A
bold letter b symbol	B
bold letter c symbol	C
bold letter d symbol	D
bold letter e symbol	E
bold letter f symbol	F
bold letter g symbol	G
bold letter h symbol	H
bold letter i symbol	I
bold letter j symbol	J
bold letter k symbol	K
bold letter l symbol	L
bold letter m symbol	M
bold letter n symbol	N
bold letter o symbol	O

bold letter p symbol	P
bold letter q symbol	Q
bold letter r symbol	R
bold letter s symbol	S
bold letter t symbol	T
bold letter u symbol	U
bold letter v symbol	V
bold letter w symbol	W
bold letter x symbol	X
bold letter y symbol	Y
bold letter z symbol	Z
digit zero symbol	0
digit one symbol	1
digit two symbol	2
digit three symbol	3
digit four symbol	4
digit five symbol	5
digit six symbol	6
digit seven symbol	7
digit eight symbol	8
digit nine symbol	9

9.2.2 Syntaktische Symbole

Symbol	Repraesentation(en)
bus symbol	], /)
case symbol	CASE
close symbol	)
colon symbol	:
comma symbol	,
concr symbol	CONCR
constant symbol	CONST
constructor open symbol	:(
defines symbol	DEFINES
downto symbol	DOWNTO
elif symbol	ELIF
else symbol	ELSE
end op symbol	ENDOP, END OP
endif symbol	ENDIF, END IF, FI
endpacket symbol	ENDPACKET, END PACKET
endproc symbol	ENDPROC, END PROC, ENDPROCEDURE, END PROCEDURE
endrepeat symbol	ENDREPEAT, END REPEAT, ENDREP, END REP, PER
endselect symbol	ENDSELECT, END SELECT
for symbol	FOR
from symbol	FROM
go on symbol	;
if symbol	IF
initial symbol	::

is defined as symbol	=
leave symbol	LEAVE
let symbol	LET
of symbol	OF
op symbol	OP, OPERATOR
open symbol	(
otherwise symbol	OTHERWISE
packet symbol	PACKET
period symbol	.
proc symbol	PROC, PROCEDURE
repeat symbol	REPEAT, REP
row symbol	ROW
select symbol	SELECT
selection symbol	.
struct symbol	STRUCT
sub symbol	[, (/
then symbol	THEN
type symbol	TYPE
until symbol	UNTIL
upto symbol	UPTO
variable symbol	VAR
while symbol	WHILE
with symbol	WITH

9.2.3 Symbole fuer Standard-Typen

Symbol	Repraesentation(en)
int symbol	INT
real symbol	REAL
bool symbol	BOOL
text symbol	TEXT

9.2.4 Symbole fuer Denotationen

Symbol	Repraesentation(en)
exponent symbol	e
true symbol	TRUE
false symbol	FALSE
quote symbol	"

9.2.5 Symbole fuer Operatoren

Symbol	Repraesentation(en)
becomes symbol	:=
or symbol	OR

and symbol	AND, &
equal symbol	=
notequal symbol	<>, ¬=, /=
less symbol	<
lessequal symbol	<=
greater symbol	>
greaterequal symbol	>=
plus symbol	+
minus symbol	-
asterix symbol	*
divide symbol	/
intdiv symbol	DIV, %, //
modulo symbol	MOD
obelix symbol	**
not symbol	¬, NOT

Als weitere Operatoren koennen andere in der jeweiligen Hardware-Konfiguration verfuegbare einzelne Sonderzeichen mit Hilfe von Operatorvereinbarungen eingefuehrt werden.

Ausser den aufgelisteten Folgen von Sonderzeichen koennen keine weiteren Folgen als Operatoren definiert werden.

9.2.6 Sonstige Symbole

Symbol	Repraesentation(en)
comment open symbol	{, #(, (*
comment close symbol	},)#, *)

Anhang A

Standard-Pakete der Programmiersprache E L A N

A.1 UEBERBLICK

In diesem Anhang werden die in jeder ELAN-Implementierung standardmaessig vorgegebenen Operationen auf Standard-Typen und entsprechende Konstanten dieser Typen definiert.

Er umfasst

- die elementaren Typen der Sprache ELAN
- standardmaessig vorgegebene abstrakte Typen
- mathematische Konstanten und Standardfunktionen
- Ein/Ausgabe
- diverse Konstanten und Informationsprozeduren

Eine spezielle Implementierung kann den definierten Standard durch weitere ergaenzen (z.B. durch Prozeduren, die die aktuelle Uhrzeit oder das Datum liefern). Es muss jedoch betont werden, dass dies lokale (d.h. nur fuer diese Implementierung gueltige) Erweiterungen der Sprache sind, und somit nicht Gegenstand dieser Definition der Standard-Pakete von ELAN.

Die Definition der Standardpakete enthaelt Angaben ueber die dem Benutzer zur Verfuegung stehenden Typen, Datenobjekte, Prozeduren und Operatoren.

Fuer Typen und Datenobjekte sind Deklarationen angegeben. Fuer Prozeduren und Operatoren sind nur die Deklarationskoepfe aufgefuehrt. Die Semantik wird in Kommentaren erlaeutert, da die Routinen nicht in ELAN formulierbar zu sein brauchen.

A.2 ELEMENTARE TYPEN

Die elementaren Typen INT, REAL, BOOL und TEXT benoetigen in einem ELAN-Compiler eine Spezialbehandlung. Diese ist notwendig wegen ihrer syntaktischen Sonderstellung in gewissen Positionen (z.B. wird der Wert eines BOOL-Objekts fuer die Abfrage benoetigt) und ihrer Denotierbarkeit im Programmtext. Ausserdem ermoeglicht diese Sonderbehandlung eine effiziente Programmausfuehrung.

A.2.1 Der Datentyp I N T

Der Datentyp INT (Integer) ermoeglicht die Benutzung einer Teilmenge der ganzen Zahlen.

Konstanten

Der groesste Wert eines INT-Objektes wird nicht durch die Sprache definiert, sondern durch die jeweilige Implementierung festgelegt.

```
INT CONST  maxint     {groesster Integer-Wert der Implementierung}
```

Assignation

```
OP  :=   (INT VAR ziel, INT CONST quelle)
                      {Zuweisung von quelle an ziel}
```

Vergleiche

```
BOOL OP = (INT CONST a, b)
                      {a gleich b}
BOOL OP <> (INT CONST a, b)
                      {a ungleich b}
BOOL OP < (INT CONST a, b)
                      {a kleiner b}
BOOL OP <= (INT CONST a, b)
                      {a kleiner oder gleich b}
BOOL OP > (INT CONST a, b)
                      {a groesser b}
BOOL OP >= (INT CONST a, b)
                      {a groesser oder gleich b}
```

Arithmetik

```
INT OP + (INT CONST a, b)
                      {Addition von a und b}
INT OP - (INT CONST a, b)
                      {Subtraktion von a und b}
```

```
INT OP * (INT CONST a, b)
                    {Multiplikation von a mit b}
INT OP DIV (INT CONST a, b)
                    {Ganzzahlige Division a durch b}
INT OP MOD (INT CONST a, b)
                    {Rest aus a DIV b}
INT OP ** (INT CONST a, b)
                    {Exponentiation a hoch b; b >= 0}
INT OP + (INT CONST i)
                    {monadisches +}
INT OP - (INT CONST i)
                    {monadisches -}
```

Arithmetik mit Zuweisung

```
OP INCR (INT VAR a, INT CONST b)
                    {a := a + b}
OP DECR (INT VAR a, INT CONST b)
                    {a := a - b}
```

Betrag und Vorzeichen

```
INT OP SIGN (INT CONST i)
INT PROC sign (INT CONST i)
INT OP ABS (INT CONST i)
INT PROC abs (INT CONST i)
```

A.2.2 Der Datentyp R E A L

Der Datentyp REAL ermoeglicht die Benutzung einer Teilmenge der rationalen Zahlen.

Konstanten

Die Genauigkeit und maximale Groesse von REAL-Werten ist nicht durch die Sprache ELAN definiert. Auch diese Konstanten sind Kenngroessen jeder Implementierung.

```
REAL CONST maxreal      {groesster REAL-Wert der Implementierung}

           smallreal    {kleinster REAL-Wert,
                         sodass 1.0 + smallreal > 1.0
                         und 1.0 - smallreal < 1.0 ist}
```

Assignation

```
OP := (REAL VAR ziel, REAL CONST quelle)
                    {Zuweisung von quelle an ziel}
```

Vergleiche

```
BOOL OP = (REAL CONST a, b)
                    {a gleich b}
BOOL OP <> (REAL CONST a, b)
                    {a ungleich b}
BOOL OP < (REAL CONST a, b)
                    {a kleiner b}
BOOL OP <= (REAL CONST a, b)
                    {a kleiner oder gleich b}
BOOL OP > (REAL CONST a, b)
                    {a groesser b}
BOOL OP >= (REAL CONST a, b)
                    {a groesser oder gleich b}
```

Arithmetik

```
REAL OP + (REAL CONST a, b)
                    {Addition von a und b}
REAL OP - (REAL CONST a, b)
                    {Subtraktion von a und b}
REAL OP * (REAL CONST a, b)
                    {Multiplikation von a mit b}
REAL OP / (REAL CONST a, b)
                    {Division a durch b}
REAL OP MOD (REAL CONST a, b)
                   {a modulo b}
REAL OP ** (REAL CONST a, INT CONST b)
                    {Exponentiation a hoch b; b >= 0}
REAL OP + (REAL CONST r)
                    {monadisches +}
REAL OP - (REAL CONST r)
                    {monadisches -}
```

Arithmetik mit Zuweisung

```
OP INCR (REAL VAR a, REAL CONST b)
                    {a := a + b}
OP DECR (REAL VAR a, REAL CONST b)
                    {a := a - b}
```

Betrag und Vorzeichen

```
INT OP SIGN (REAL CONST x)
INT PROC sign (REAL CONST x)
REAL OP ABS (REAL CONST x)
REAL PROC abs (REAL CONST x)
```

Ueberfuehrung von INT-Werten nach REAL und umgekehrt

```
REAL PROC real (INT CONST i)

INT PROC trunc (REAL CONST a)
                        {Abschneiden der Dezimalstellen}
                        {Es gilt:
                         trunc(-a) = -trunc(a)}
INT PROC round (REAL CONST a)
                        {Runden}
                        {Es gilt:
                         round(-a) = -round(a)}
```

A.2.3 Der Datentyp B O O L

Konstanten

```
BOOL CONST true :: TRUE
BOOL CONST false :: FALSE
```

Assignation

```
OP := (BOOL VAR ziel, BOOL CONST quelle)
                    {Zuweisung von quelle an ziel}
```

Boolesche Verknuepfungen

```
BOOL OP NOT (BOOL CONST a)
                    {Negation von a}
BOOL OP AND (BOOL CONST a, b)
                    {Logisches und}
BOOL OP OR (BOOL CONST a, b)
                    {Logisches oder}
BOOL OP XOR (BOOL CONST a, b)
                    {Logisches exklusives oder}
```

A.2.4 Der Datentyp T E X T

Konstanten

```
TEXT CONST niltext = "",
           blank   = " ",
           quote   = """";
```

Assignation

```
OP := (TEXT VAR ziel, TEXT CONST quelle)
                    {Zuweisung von quelle an ziel}
```

Vergleiche

```
BOOL OP = (TEXT CONST a, b)
                    {a gleich b}
BOOL OP <> (TEXT CONST a, b)
                    {a ungleich b}
BOOL OP < (TEXT CONST a, b)
                    {a vor b}
BOOL OP <= (TEXT CONST a, b)
                    {a vor oder gleich b}
BOOL OP > (TEXT CONST a, b)
                    {a nach b}
BOOL OP >= (TEXT CONST a, b)
                    {a nach oder gleich b}
```

Die Texte werden nach den ueblichen lexikografischen Regeln verglichen, wobei die Relationen zwischen einzelnen Zeichen durch die interne Codierung des benutzten Rechners oder des Compilers festgelegt wird. Rechnerunabhaengig ist jedoch die Ordnung von Buchstaben a..z und A..Z und die Ordnung der Ziffern 0..9 festgelegt, wobei sich keine anderen Zeichen in den jeweiligen Reihen befinden sollen. Beim Vergleich von Texten unterschiedlicher Laenge, wobei der eine Text den anderen als Anfang enthaelt, gilt:

```
a + x > a   {a,x: TEXT; LENGTH x > 0}
```

Operationen auf Texten

```
INT OP LENGTH (TEXT CONST t)
INT PROC length (TEXT CONST t)
                    {Laenge des Textes in Zeichen}

TEXT OP + (TEXT CONST a, b)
                    {Verkettung der beiden Texte}

OP CAT (TEXT VAR a, TEXT CONST b)
                    {a := a + b}

TEXT OP * (INT CONST i, TEXT CONST t)
                    {i-1 faches + mit sich selbst;
                     i >= 0}

TEXT PROC compress (TEXT CONST t)
                    {t ohne Leerzeichen am Anfang und Ende}
```

```
TEXT PROC text (TEXT CONST t, INT CONST l)
                    {Erzeugung eines Textes aus t
                     mit der Laenge l, l >= 0;
                     fuer l > LENGTH t  Auffuellen mit Leerzeichen,
                     fuer l < LENGTH t  Abschneiden}

TEXT PROC text (TEXT CONST t, INT CONST l, p)
                    {Erzeugen eines Textes aus t mit der
                     Laenge l beginnend bei der Position
                     p in t;
                     l >= 0, l <= p <= LENGTH t;
                     fuer l > (LENGTH t) - p + 1
                                      Auffuellen mit Leerzeichen,
                     fuer l < (LENGTH t) - p + 1
                                      Abschneiden};

TEXT PROC subtext (TEXT CONST t, INT CONST von)
                    {Teiltext von t, beginnend bei der
                     Position von;
                     l <= von <= LENGTH t}

TEXT PROC subtext (TEXT CONST t, INT CONST von, bis)
                    {Teiltext von t, beginnend bei der
                     Position von, endend bei der
                     Position bis;
                     l <= von <= LENGTH t,
                     von <= bis <= LENGTH t};

TEXT OP SUB (TEXT CONST t, INT CONST p)
                    {hat die Semantik von
                     subtext (t, p, p)}

PROC replace (TEXT VAR t, INT CONST p, TEXT CONST new)
                    {Teiltextersetzung von t durch new
                     an den Positionen p bis
                     p + LENGTH new - 1;
                     l <= p <= LENGTH t;
                     die Laenge von t bleibt unveraendert,
                     d.h. wenn
                     LENGTH new > (LENGTH t) - p + 1
                     wird abgeschnitten}

PROC change (TEXT VAR t, TEXT CONST old, new)
                    {Teiltextersetzung von t;
                     der Teiltext old wird an der Stelle
                     seines ersten Auftretens durch den
                     Text new ersetzt;
                     tritt der Text old in t nicht auf,
                     so erfolgt eine Meldung;
                     die Laenge von t kann durch die
                     Prozedur change geaendert werden}
```

```
INT PROC pos (TEXT CONST t1, t2)
                    {das erste Auftreten von t2 steht in t1 ab der
                     gelieferten Position;
                     falls t2 nicht in t1 enthalten ist
                     wird 0 geliefert;
                     ist t2 der niltext, so wird 0
                     geliefert}
```

Die Zaehlung der Textpositionen beginnt immer bei 1.

```
INT PROC pos (TEXT CONST t1, t2, INT CONST anfangsposition)
              {Analog der obigen Prozedur pos, die Suche
               beginnt jedoch erst an der Anfangsposition.}
```

A.3 STANDARD ABSTRAKTE TYPEN

Ausser den in Abschnitt A.2 behandelten ELAN-Standardtypen gibt es einige Standardpakete fuer abstrakte Typen der numerischen Mathematik:

- Komplexe Zahlen (Datentyp COMPLEX)
- Vektoren und Matrizen von REAL-Zahlen (Datentypen VECTOR und MATRIX)

Vektoren und Matrizen haben zur Laufzeit des Programms festlegbare Groessen ("dynamische arrays").

A.3.1 Der Datentyp C O M P L E X

```
TYPE COMPLEX = STRUCT (REAL re, im);
```

Konstanten

```
COMPLEX CONST complex null = COMPLEX: (0.0, 0.0),
              complex eins = COMPLEX: (1.0, 0.0),
              complex i    = COMPLEX: (0.0, 1.0);
```

Denotierungsprozeduren

```
COMPLEX PROC complex (REAL CONST r, i)
        {Komplexe Zahl in kartesischen Koordinaten}
COMPLEX PROC complexpolar (REAL CONST abs, phi)
        {Komplexe Zahl in Polarkoordinaten}
```

Selektionsprozeduren

```
REAL PROC realpart (COMPLEX CONST z)
                            {Realteil von z}
REAL PROC imagpart (COMPLEX CONST z)
                            {Imaginaerteil von z}
REAL PROC abs (COMPLEX CONST z)
REAL OP ABS (COMPLEX CONST z)
                            {Betrag von z}
REAL PROC phi (COMPLEX CONST z)
                            {Winkel von z
                             (Polardarstellung)}
```

Assignation

```
OP := (COMPLEX VAR ziel, COMPLEX CONST quelle)
                     {Zuweisung von quelle an ziel}
```

Vergleiche

```
BOOL OP = (COMPLEX CONST a, b)
                     {a gleich b}
BOOL OP <> (COMPLEX CONST a, b)
                     {a ungleich b}
```

Arithmetik

```
COMPLEX OP CONJ (COMPLEX CONST z)
                        {konjugiert komplexe zu z}
COMPLEX OP + (COMPLEX CONST a, b)
                        {Addition von a und b}
COMPLEX OP - (COMPLEX CONST a, b)
                        {Subtraktion von a und b}
COMPLEX OP * (COMPLEX CONST a, b)
                        {Multiplikation von a mit b}
COMPLEX OP / (COMPLEX CONST a, b)
                        {Division a durch b}
COMPLEX OP + (COMPLEX CONST a)
                        {monadisches +}
COMPLEX OP - (COMPLEX CONST a)
                        {monadisches -}
```

Ein/Ausgabe (Dialog-E/A, vgl. A.5.1)

```
PROC get (COMPLEX VAR c)
                  {Einlesen eines COMPLEX-Wertes von der
                   Standardeingabe}
PROC put (COMPLEX CONST c)
                  {Ausgabe eines COMPLEX-Wertes auf die
                   Standardausgabe}
```

A.3.2 Der Datentyp V E C T O R

Ein Vector ist eine Reihung von REAL's mit zur Laufzeit des Programms festgelegter Laenge. Der Zugriff auf die Elemente erfolgt durch Indizierung (mit dem SUB-Operator). Die Numerierung der Elemente beginnt stets bei 1. VECTORen bieten die Moeglichkeit, Reihungen aufzubauen, ohne im Programmtext die Anzahl der Elemente der Reihung, wie bei z.B. ROW 100 REAL notwendig, schon festzulegen.

```
TYPE VECTOR = {Realisierung wird hier nicht angegeben};
```

Denotierungsprozeduren

```
VECTOR PROC vector (INT CONST lng, REAL CONST value)
                    {Erzeugen eines Vectors der Laenge lng,
                     dessen Elemente alle den Wert value
                     haben}
VECTOR PROC vector (INT CONST lng):
                     vector (lng, 0.0) ENDPROC vector;
```

Selektionsroutinen

```
REAL OP SUB (VECTOR CONST v, INT CONST i)
                    {i-tes Element von v}
INT OP LENGTH (VECTOR CONST v)
INT PROC length (VECTOR CONST v)
                    {Laenge von v}
```

Assignation

```
OP := (VECTOR VAR ziel, VECTOR CONST quelle)
                {Zuweisung von quelle an ziel}
PROC replace (VECTOR VAR v, INT CONST i, REAL CONST r)
                {Ersetzung des i-ten Elementes von v
                 durch r}
```

Vergleiche

```
BOOL OP = (VECTOR CONST a, b)
                {a gleich b}
BOOL OP <> (VECTOR CONST a, b)
                {a ungleich b}
```

Arithmetik

```
VECTOR OP + (VECTOR CONST a, b)
                {Addition von a und b}
VECTOR OP - (VECTOR CONST a, b)
                {Subtraktion von a und b}
REAL OP * (VECTOR CONST a, b)
                {skalare Multiplikation von a und b}
VECTOR OP * (VECTOR CONST v, REAL CONST r)
                {Multiplikation von v mit dem Skalar r}
VECTOR OP * (REAL CONST r, VECTOR CONST v)
                {Multiplikation von v mit dem Skalar r}
VECTOR OP / (VECTOR CONST v, REAL CONST r)
                {Multiplikation von v mit dem Skalar 1/r}
VECTOR OP + (VECTOR CONST v)
               {monadisches +}
VECTOR OP - (VECTOR CONST v)
               {monadisches -}
```

Ein/Ausgabe (Dialog-E/A, vgl. A.5.1)

```
PROC get (VECTOR VAR v, INT CONST laenge)
                {Einlesen eines VECTORSs angegebener
                 Laenge von Standardeingabe}
PROC put (VECTOR CONST v)
                {Ausgabe eines VECTORs auf
                 Standardausgabe}
```

A.3.3 Der Datentyp M A T R I X

Der Datentyp MATRIX realisiert eine Matrix von REAL-Koeffizienten mit zur Laufzeit dynamisch festlegbarer Groesse (rows, columns). Fuer den Zugriff auf einzelne Elemente, Zeilen und Spalten steht der SUB-Operator, bzw. row- und column-Prozeduren zur Verfuegung. Die Numerierung von Zeilen und Spalten beginnt stets bei 1.

```
TYPE MATRIX = {Realisierung wird hier nicht angegeben};
```

Denotierungsprozeduren

```
MATRIX PROC matrix (INT CONST rows, columns, REAL CONST value)
                    {Erzeugen einer Matrix der Groesse
                     rows*columns, deren Elemente alle
                     den Wert value haben}

MATRIX PROC matrix (INT CONST rows, columns):
             matrix (rows, columns, 0.0) ENDPROC matrix;
```

```
MATRIX PROC idn (INT CONST rows, columns)
                        {Einheitsmatrix rows*columns}
```

Selektionsroutinen

```
VECTOR PROC row (MATRIX CONST m, INT CONST i)
                        {i-te Zeile von m}
VECTOR PROC column (MATRIX CONST m, INT CONST i)
                        {i-te Spalte von m}
REAL OP SUB (MATRIX CONST m, ROW 2 INT indexpair)
                        {Durch indexpair bestimmtes Element
                         von m}
INT OP ROWLENGTH (MATRIX CONST m)
                        {Anzahl der Elemente in einer Reihe}
INT OP COLUMNLENGTH (MATRIX CONST m)
                        {Anzahl der Elemente in einer Spalte}
```

Zuweisungsroutinen

```
OP := (MATRIX VAR ziel, MATRIX CONST quelle)
                    {Zuweisung von quelle an ziel}

PROC replace row (MATRIX VAR m, INT CONST rowindex,
                          VECTOR CONST rowvalue)
                           {Ersetzung der durch rowindex
                            bestimmten Zeile von m durch
                            rowvalue}
PROC replace column (MATRIX VAR m, INT CONST columnindex,
                          VECTOR CONST columnvalue)
                           {Ersetzung der durch columnindex
                            bestimmten Spalte von m durch
                            columnvalue}
PROC replace element (MATRIX VAR m, INT CONST rowindex,
                          columnindex, REAL CONST value)
                           {Ersetzung des durch rowindex und
                            columnindex bestimmten Elementes
                            von m durch value}
```

Vergleiche

```
BOOL OP = (MATRIX CONST a, b)
                        {a gleich b}
BOOL OP <> (MATRIX CONST a, b)
                        {a ungleich b}
```

Arithmetik

```
MATRIX OP + (MATRIX CONST a, b)
                        {Addition von a und b}
```

```
MATRIX OP - (MATRIX CONST a, b)
                    {Subtraktion von a und b}
MATRIX OP * (MATRIX CONST a, b)
                    {Multiplikation von a und b}
MATRIX OP * (MATRIX CONST m, REAL CONST r)
                    {Multiplikation m mit Skalar r}
MATRIX OP * (REAL CONST r, MATRIX CONST m)
                    {Multiplikation m mir Skalar r}
MATRIX OP INV (MATRIX CONST m)
                    {Inverse von m}
REAL OP DET (MATRIX CONST m)
                    {Determinante von m}
MATRIX OP TRANSP (MATRIX CONST m)
                    {Transponierte von m}
PROC transp (MATRIX VAR m)
                    {Transponieren m in situ}
MATRIX OP + (MATRIX CONST m)
                   {monadisches +}
MATRIX OP - (MATRIX CONST m)
                   {monadisches -}
```

Ein/Ausgabe (Dialog-E/A, vgl. A.5.1)

```
PROC get (MATRIX VAR m, INT CONST rows, columns)
                 {Einlesen einer MATRIX der Groesse
                  rows*columns von Standardeingabe}
PROC put (MATRIX CONST m)
                 {Ausgabe einer MATRIX auf Standard-
                  ausgabe}
```

A.4 MATHEMATISCHE ROUTINEN UND KONSTANTEN

Quadratwurzel

```
REAL PROC sqrt (REAL CONST x)
```

Trigonometrie

```
REAL CONST pi = 3.14159265358979323846264 3{...};

REAL PROC sin (REAL CONST x)
REAL PROC cos (REAL CONST x)
REAL PROC tan (REAL CONST x)
REAL PROC arctan (REAL CONST x)
```

Exponential- und Logarithmusfunktion

```
REAL CONST e = 2.71828182845904523536028 7{...};
```

```
REAL PROC exp (REAL CONST x)
REAL PROC ln (REAL CONST x)
```

Quasi-Zufallszahlengenerator

```
REAL PROC random   {Gleichverteilung zwischen 0.0 und 1.0}

INT PROC random (INT CONST min, max)
                    {min <= max; der Wertebereich
                     liegt zwischen einschliesslich
                     min und max.
                     Diese Prozedur wird intern auf die
                     REAL PROC random zurueckgefuehrt}

PROC initialize random (REAL CONST x)
                   {0 <= x <= 1
                    (sonst keine Initialisierung);
                    macht random reproduzierbar}
```

A.5 EIN/AUSGABE

A.5.1 Dialog-Ein/Ausgabe

Fuer den einfachsten Fall von Ein/Ausgabe und zur Unterstuetzung einer Dialogumgebung gibt es einige einfache Prozeduren, die den Benutzer davon entlasten, seine Zeilen selbst aufzubauen bzw. zu analysieren, sowie fuer korrekte Synchronisation von Terminal-Ein/Ausgabe zu sorgen.

A.5.1.1 Zeilenverwaltung

Zur Zeilenverwaltung existiert ein Zeilenpuffer, der durch die Prozeduren put / get / line / page verwaltet wird.

Bei der Aufeinanderfolge von put und get (bzw. get und put) wird vor der eigentlich auszufuehrenden Aktion der Zeilenpuffer ausgegeben (bzw. der Zeilenpuffer reinitialisiert). Ausserdem wird ein impliziter Zeilenwechsel bei der Ausgabe dann eingefuegt, wenn ein Wert nicht mehr auf die Zeile passt.

Ein expliziter Zeilenwechsel kann durch den Aufruf von

```
PROC line (INT CONST zeilenzahl)
PROC line
```

veranlasst werden. Seitenwechsel (A.5.5) erfolgt durch den Aufruf von

```
PROC page
```

Ein Aufruf von line oder page bewirkt die Ausgabe des aktuellen Zeilenpuffers, falls die letzte Ein/Ausgabe-Operation ein put war, anderenfalls wird der Zeilenpuffer reinitialisiert. Danach werden bei

- line(n) n Leerzeilen ausgegeben,

- page in der Ausgabe eine neue Seite begonnen.

A.5.1.2 Die Ausgabeprozedur p u t

Bei numerischen Werten (INT, REAL) wird der konvertierte Wert in den Zeilenpuffer geschrieben, sofern er dort vollstaendig Platz findet. Noetigenfalls wird die Ausgabe auf einer neuen Zeile fortgesetzt.

Texte werden, soweit moeglich, in den Zeilenpuffer gesetzt. Konnte ein Text nicht vollstaendig in den Zeilenpuffer gebracht werden, so wird der Rest des Textes auf einer neuen Zeile ausgegeben.

Die Ausgabe wird vom bisherigen Zeileninhalt durch ein Leerzeichen getrennt, ausgenommen am Zeilenanfang.

```
PROC put (INT CONST i)
PROC put (REAL CONST r)
PROC put (TEXT CONST t)
```

A.5.1.3 Die Eingabeprozedur g e t

Fuer Objekte vom Typ INT, REAL und TEXT existieren die Eingabeprozeduren:

```
PROC get (INT VAR i)
PROC get (REAL VAR r)
PROC get (TEXT VAR t)
TEXT PROC get
```

Die Objekte werden jeweils bis zum naechsten Leerzeichen bzw. Zeilenende aus dem Zeilenpuffer geholt. Fuehrende Leerzeichen werden ueberlesen. Noetigenfalls werden implizit neue Eingabezeilen in den Eingabepuffer geholt (A.5.6.1).

Fuer das Einlesen von Textobjekten stehen zwei weitere Prozeduren zur Verfuegung:

```
PROC get (TEXT VAR t, TEXT CONST delimiter)
PROC get (TEXT VAR t, INT CONST maxlength)
```

Fuer beide Prozeduren gilt, dass Leerzeichen am Zeilenende der realen Benutzereingabe nicht uebertragen werden.

Die erste Prozedur liest von der aktuellen Position bis zum Anfang des Vergleichstextes (delimiter), wobei der Vergleichstext nicht zum Ergebnis gehoert. Steht der Vergleichstext an aktueller Position, so ist das Ergebnis niltext. Die neue aktuelle Position befindet sich nach jeder Leseoperation direkt hinter dem Vergleichstext. Ein impliziter Zeilenwechsel wird nicht vorgenommen.

Die zweite Prozedur liest einen Text vorgegebener maximaler Laenge (maxlength), jedoch nur bis zum Zeilenende. Ein impliziter Zeilenwechsel wird auch hier nicht vorgenommen.

Fuer beide Prozeduren folgt aus dem oben gesagten, dass die Eingabe einer Leerzeile als Ergebnis niltext liefert, da Leerzeichen am Zeilenende nicht uebertragen werden.

A.5.2 Dateien

Ein/Ausgabe in ELAN ist unabhaengig von irgendwelchen Eigenschaften von Geraeten, Spoolsystemen etc. Stattdessen gibt es Dateien ('files'), welche die notwendige Verwaltung in Kooperation mit dem jeweiligen Betriebssystem erledigen. Nach ihrem Verwendungszweck werden zwei Dateitypen unterschieden:

```
TYPE FILE    = {sequentieller Zugriff}
               {Feinstruktur wird hier nicht gegeben};

TYPE DIRFILE = {direkter Zugriff (random access)}
               {Feinstruktur wird hier nicht gegeben};
```

A.5.3 Verwaltung der Dateien

Die Assoziierung einer Datei mit einer externen (Betriebssystem-) Kennzeichnung geschieht z.B. bei ihrer Vereinbarung durch die Prozeduren

```
FILE PROC sequential file
             (TRANSPUTDIRECTION CONST direction,
              TEXT CONST identification);

DIRFILE PROC direct file
             (TRANSPUTDIRECTION CONST direction,
              TEXT CONST identification);
```

Die Kennzeichnung 'identification' kann der Name einer vom Betriebssystem verwalteten Datei oder auch eine Geraetebezeichnung sein. Sie ist implementierungsabhaengig.

Durch die Assoziierung mit der externen Kennzeichnung wird eine Datei auch eroeffnet (es gibt keine besondere open-Prozedur). Der Erfolg dieser Operation kann durch die Informationsprozeduren open und new (A.5.7.1) abgefragt werden.

Eine Datei kann immer nur auf eine der drei folgenden Weisen (direction) betrieben werden:

```
TRANSPUTDIRECTION CONST  input, output, update;
```

Dabei ist fuer Dateien vom Typ FILE nur input oder output, fuer Dateien vom Typ DIRFILE nur input (Lesen) oder update (Lesen und Schreiben) erlaubt. Ein Wechsel des Betriebsmodus ist nur durch explizites Schliessen (close) und Wiedereroeffnen einer Datei moeglich.

Das Abschliessen einer Datei geschieht explizit durch die Prozeduren.

```
PROC close (FILE CONST f)
PROC close (DIRFILE CONST f)
```

Allerdings werden auch nicht vom Benutzer abgeschlossene Dateien bei Verlassen des Gueltigkeitsbereichs der Deklaration implizit abgeschlossen.

Dateien koennen geloescht werden:

```
PROC erase (FILE CONST f)
PROC erase (DIRFILE CONST f)
```

A.5.4 Strukturierung und Zugriffsauswahl

Die Ein/Ausgabe erfolgt in ELAN prinzipiell zeilenweise (Record-E/A von TEXTen). Die Prozeduren getline und putline beziehen sich bei sequentieller Ein/Ausgabe (A.5.5) stets auf den naechsten Satz, bei direkt Ein/Ausgabe (A.5.6) auf den angegebenen Schluessel. Schluessel sind TEXTe fester Laenge. Sequentielle Dateien koennen eine zusaetzliche Seitenstrukturierung haben.

A.5.5 Sequentielle Ein/Ausgabe

Bei sequentieller Ein/Ausgabe wird jeweils genau eine Zeile uebertragen.

```
PROC putline (FILE CONST f, TEXT CONST line)
PROC getline (FILE CONST f, TEXT VAR line)
```

Ist eine Ausgabezeile laenger als maxlinelength(f) (A.5.7.3), so wird sie nur entsprechend verkuerzt ausgegeben, und der Rest geht verloren. Beim Lesen nach Erreichen von end-of-file wird an den Parameter 'line' der Wert niltext zugewiesen (vgl. Test auf end-of-file: A.5.7.2).

Fuer sequentielle Dateien, die eine zusaetzliche Seitenstrukturierung haben, gibt es die Prozedur (unwirksam bei anderen):

```
PROC page (FILE CONST f)
```

Sie setzt die Datei auf den Anfang der naechsten Seite.

Sequentielle Dateien koennen auf ihre Anfangsposition zurueckgesetzt werden durch

```
PROC reset (FILE CONST f)
```

Es kann Dateien geben, bei denen diese Operation keine Wirkung hat(z.B. Ausgabe auf den Schnelldrucker).

A.5.5.1 Sequentielle Ein/Ausgabe mit Zeilenpuffer

Die folgenden Prozeduren koennen analog zur Dialog-E/A benutzt werden:

```
PROC line (FILE CONST f)

PROC line (FILE CONST f, INT CONST zeilenzahl)
       {nur bei TRANSPUTDIRECTION OUTPUT}

PROC page (FILE CONST f)
       {nur bei TRANSPUTDIRECTION OUTPUT}

PROC put (FILE CONST f, INT CONST i)
PROC put (FILE CONST f, REAL CONST r)
PROC put (FILE CONST f, TEXT CONST t)

PROC get (FILE CONST f, INT VAR i)
PROC get (FILE CONST f, REAL VAR r)
PROC get (FILE CONST f, TEXT VAR t)
TEXT PROC get (FILE CONST f)

PROC get (FILE CONST f, TEXT VAR t,
          TEXT CONST delimiter)
PROC get (FILE CONST f, TEXT VAR t,
          INT CONST maxlength)
```

Alle diese Prozeduren werden auf putline und getline zurueckgefuehrt. Der Zeilenpuffer ist jeweils der Datei zugeordnet.

A.5.6 Direkte Ein/Ausgabe

Direkte Ein/Ausgabe wird ueber einen Satzschluessel (ein Textobjekt fester Laenge) gesteuert, der die jeweilige Position in der Datei angibt.

```
PROC putline (DIRFILE CONST f, TEXT CONST key, line)
PROC getline (DIRFILE CONST f, TEXT CONST key,
                                TEXT VAR line)
```

Die maximale Laenge des Schluessel-Textes ist implementierungsabhaengig. Wird ein zu kurzer Text als Schluessel angegeben, so wird er rechts mit Leerzeichen auf die notwendige Laenge aufgefuellt. Wird ein zu langer Text angegeben, so wird er entsprechend rechts abgeschnitten.

Der Aufruf von getline fuer einen nicht vorhandenen Satz liefert niltext fuer line; entsprechend loescht putline mit niltext den angegebenen Satz.

Ist eine Ausgabezeile laenger als maxlinelength(f) (A.5.7.3), so wird sie entsprechend verkuerzt ausgegeben.

A.5.7 Informationsprozeduren

A.5.7.1 Zustandsabfragen

```
BOOL PROC opened (FILE CONST f)
BOOL PROC opened (DIRFILE CONST f)
                    {Datei wurde korrekt eroeffnet}

BOOL PROC new (FILE CONST f)
BOOL PROC new (DIRFILE CONST f)
                    {Datei wurde beim Eroeffnen neu
                     eingerichtet}
```

A.5.7.2 Tests auf Ueberschreitung des Dateiendes

Mit den beiden folgenden Prozeduren kann der Benutzer pruefen, ob bei der letzten Ein/Ausgabe-Operation das Dateiende ueberschritten wurde. Dies ist erst nach der Operation moeglich, weil fast alle Betriebssysteme keine Tests wie end-of-file-ahead kennen. Es muss also immer erst ein echter Versuch unternommen werden.

```
BOOL PROC eof (FILE CONST f)
BOOL PROC eof                    {Dialog-E/A (A.5.1)}
```

A.5.7.3 Tests auf Ueberschreiten des Seitenendes

Analog zu eof kann bei Dateien mit Seitenstruktur das Ueberschreiten des Seitenendes mit folgenden Prozeduren festgestellt werden:

```
BOOL PROC eop (FILE CONST f)
BOOL PROC eop                    {Dialog-E/A (A.5.1)}
```

A.5.7.4 Maximale Ausdehnung einer Datei

Da es auch Zeilen , einer limitierten Laenge (z.B. Druckzeilen) gibt, kann der Benutzer die Zeilenlaenge mittels

```
INT PROC maxlinelength (FILE CONST f)
INT PROC maxlinelength (DIRFILE CONST f)
INT PROC maxlinelength              {Dialog-E/A (A.5.1)}
```

feststellen. Man beachte, dass es sich dabei stets um logische Zeilenlaengen handelt. Entsprechend liefert

```
INT PROC maxpagelength (FILE CONST f)
INT PROC maxpagelength              {Dialog-E/A (A.5.1)}
```

die Anzahl der Zeilen, die maximal auf eine Seite passen.

A.6 KONVERTIERUNGEN

Durch Konvertierungsprozeduren ist es moeglich, von einem Datenobjekt eine Repraesentation als TEXT zu erhalten, bzw. in einem TEXT eine Repraesentation fuer einen Wert zu finden, (und ggf. umzuwandeln), z.B. aus einem INT-Objekt ein TEXT-Objekt mit der Darstellung des Wertes.

A.6.1 Umwandlungen in einen Text

A.6.1.1 Standardformat moeglichst kurzer Laenge

```
TEXT PROC text (INT CONST i)
TEXT PROC text (REAL CONST r)
TEXT PROC text (COMPLEX CONST c)
```

A.6.1.2 Standardformat fest vorgegebener Laenge

(Die Eintragung erfolgt rechtsbuendig fuer numerische Werte, fuer Texte jedoch linksbuendig.)

```
TEXT PROC text (INT CONST i, INT CONST laenge)
TEXT PROC text (REAL CONST r, INT CONST laenge)
TEXT PROC text (REAL CONST r, INT CONST laenge,nachkomma)
```

A.6.2 Erkennung numerischer Werte

```
INT PROC int (TEXT CONST t)
INT PROC int (TEXT CONST t, INT VAR p)

REAL PROC real (TEXT CONST t)
REAL PROC real (TEXT CONST t, INT VAR p)

COMPLEX PROC complex (TEXT CONST t)
COMPLEX PROC complex (TEXT CONST t, INT VAR p)
```

Im Text t wird versucht, eine Repraesentation des entsprechenden Datentyps zu finden und sie zu konvertieren. Dies beginnt entweder mit dem ersten Zeichen (erste Form), oder an der Stelle p, wobei p beim Erkennen auf das erste Zeichen im Text hinter dem erkannten Datenwert gesetzt wird.

Ein Konvertierungsfehler (keine Repraesentation eines Datenwertes an der untersuchten Stelle im Text, oder auch Overflow beim Akkumulieren einer zu grossen Repraesentation) kann erfragt werden durch:

```
BOOL PROC last conversion ok
```

Die Konvertierungsroutinen verursachen keinen Programmabbruch.

A.7 PROGRAMMABBRUCH

Ein Programmabbruch kann durch den Aufruf der Prozeduren

```
PROC stop
PROC errorstop
```

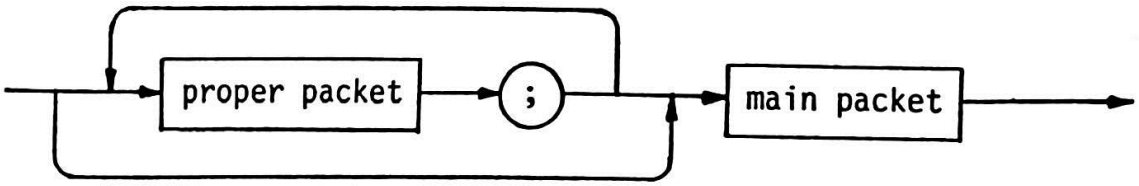
erreicht werden.

Beim Aufruf von errorstop wird zusaetzlich eine implementierungsabhaengige Postmortembehandlung gestartet.

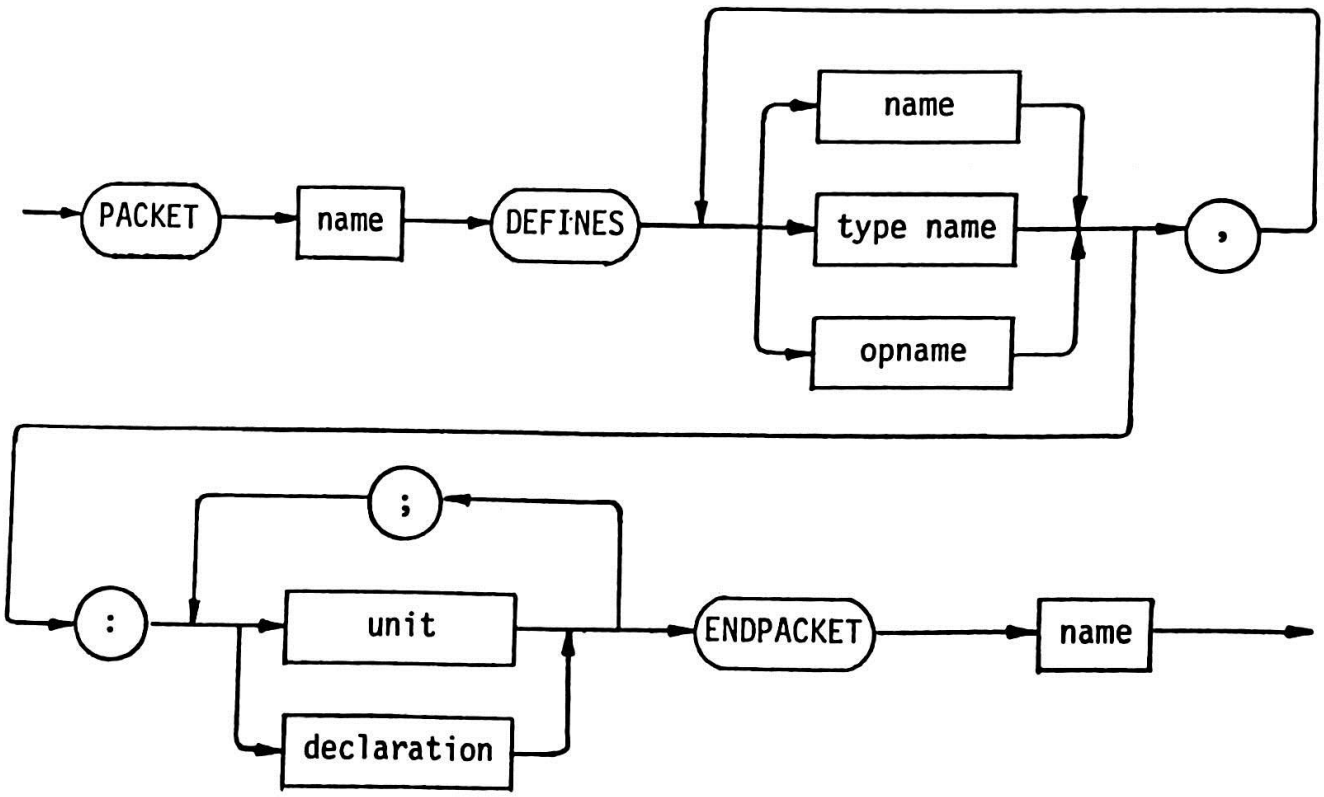
Anhang B

Syntax-Diagramme

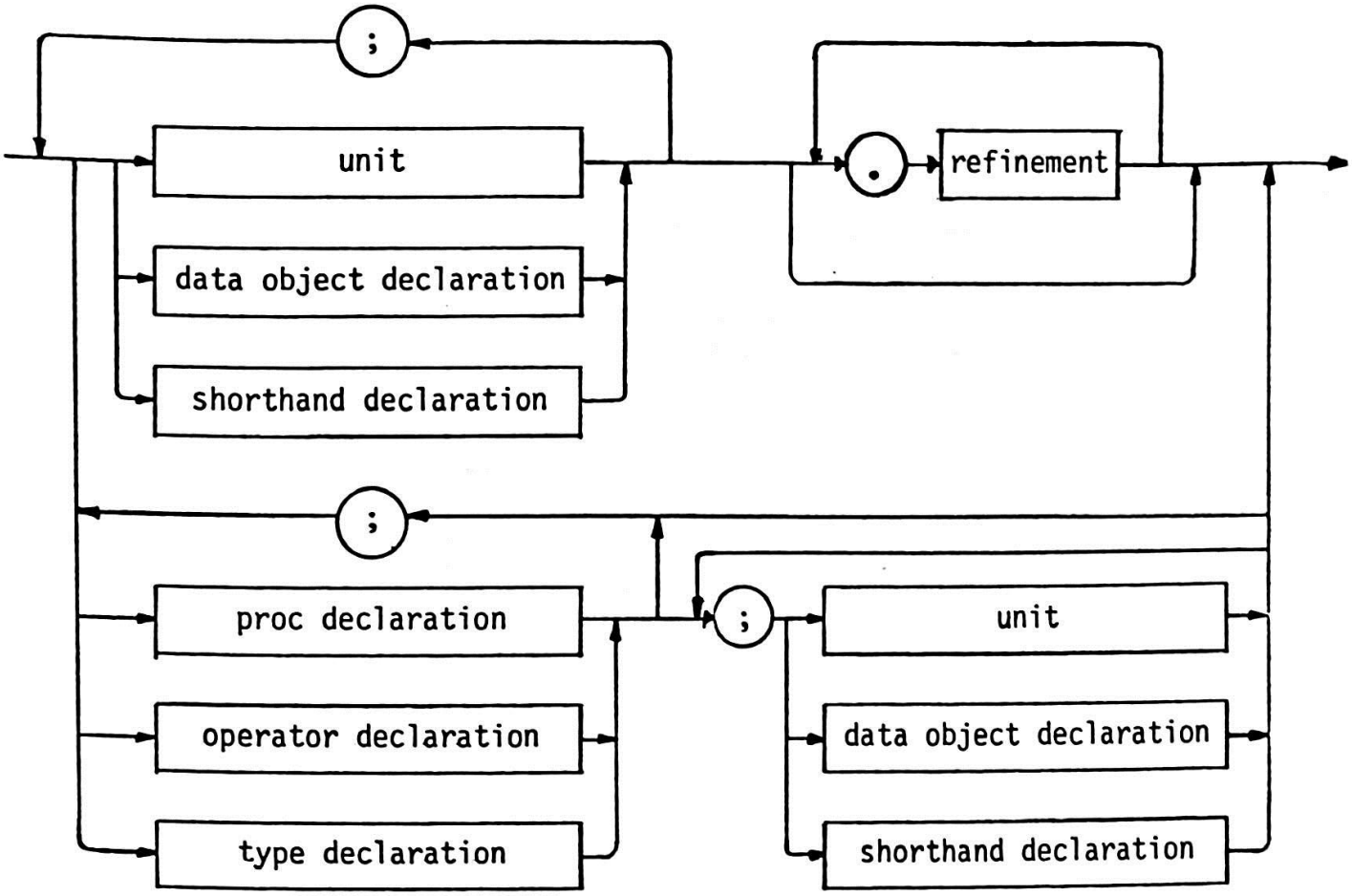
program



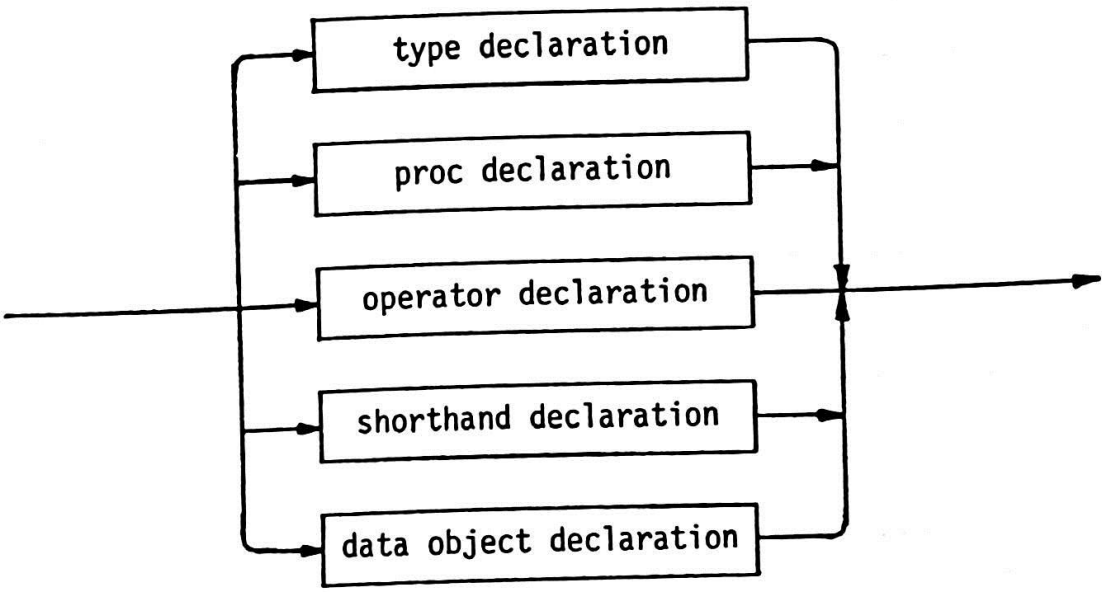
proper packet



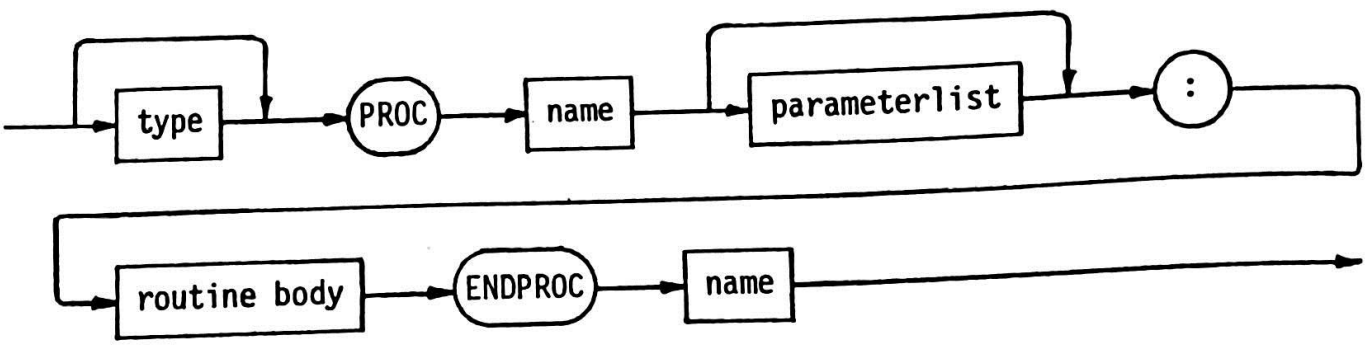
main packet


;
unit
data object declaration
shorthand declaration
.
refinement
;
proc declaration
operator declaration
type declaration
;
unit
data object declaration
shorthand declaration

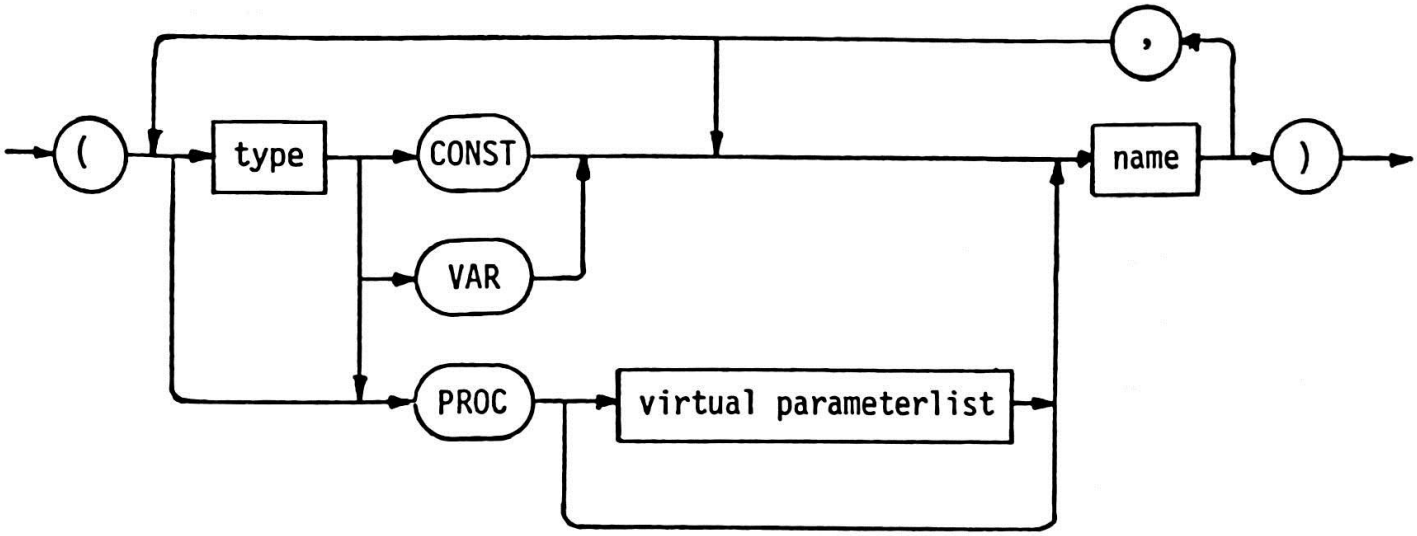
declaration



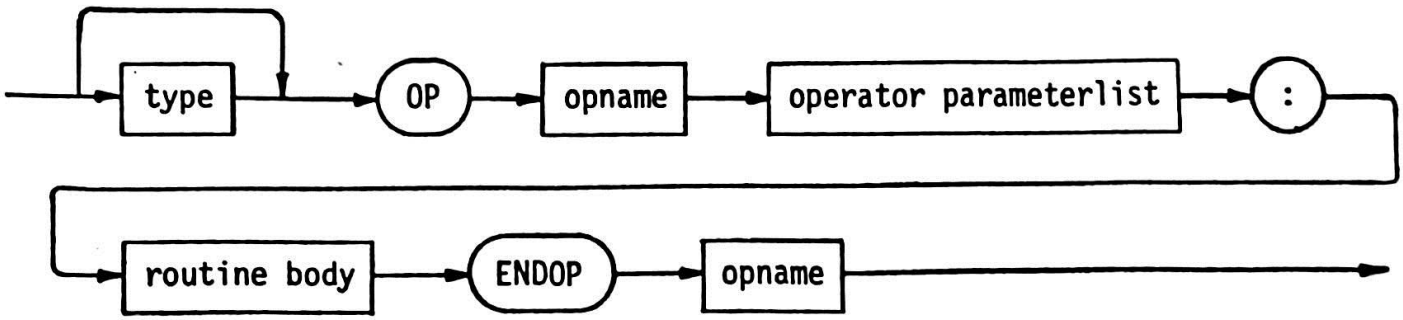
proc declaration



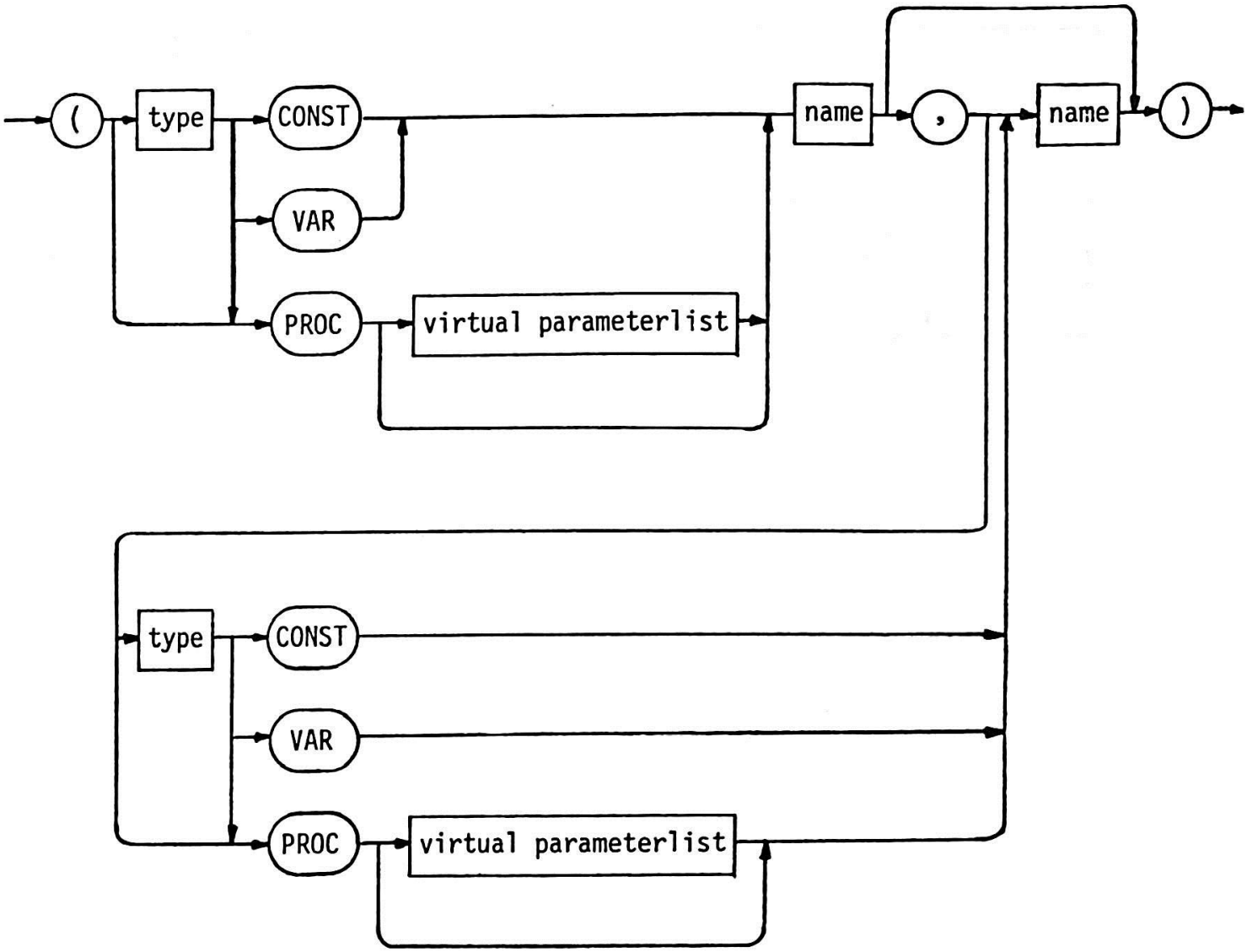
parameter list



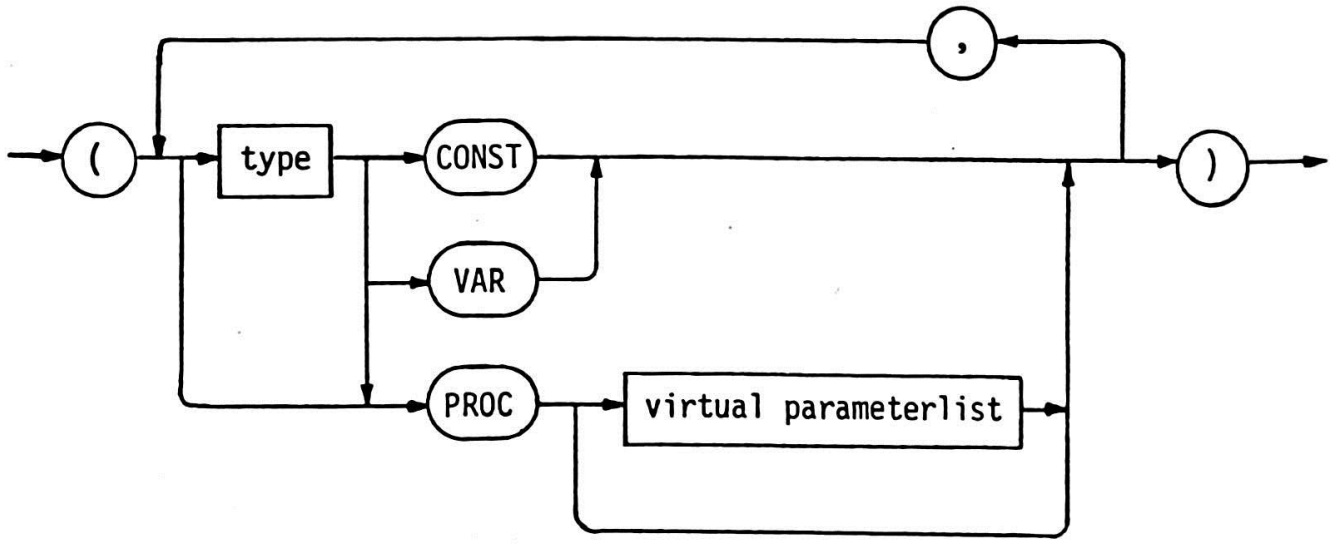
operator declaration



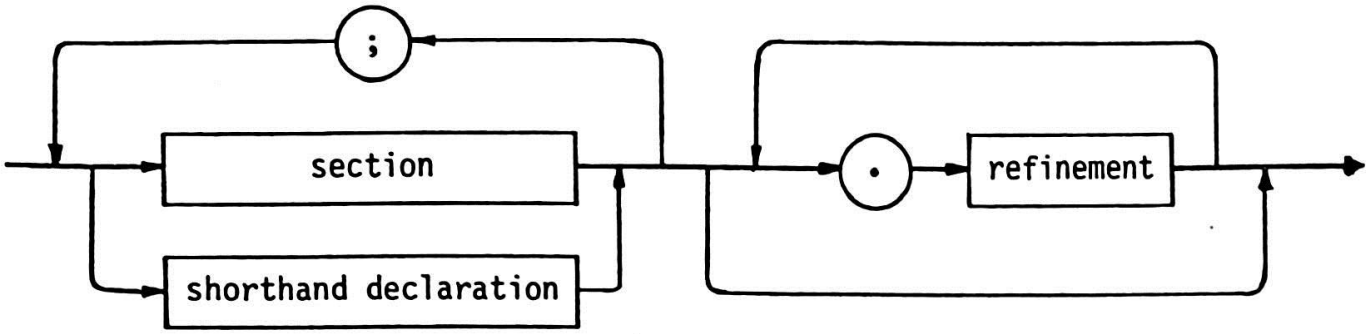
operator parameter list



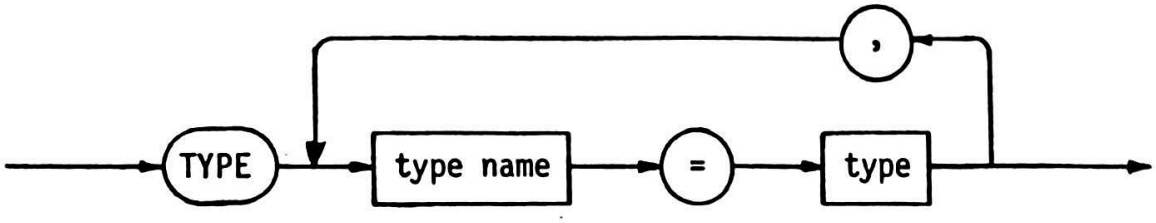
virtual parameterlist



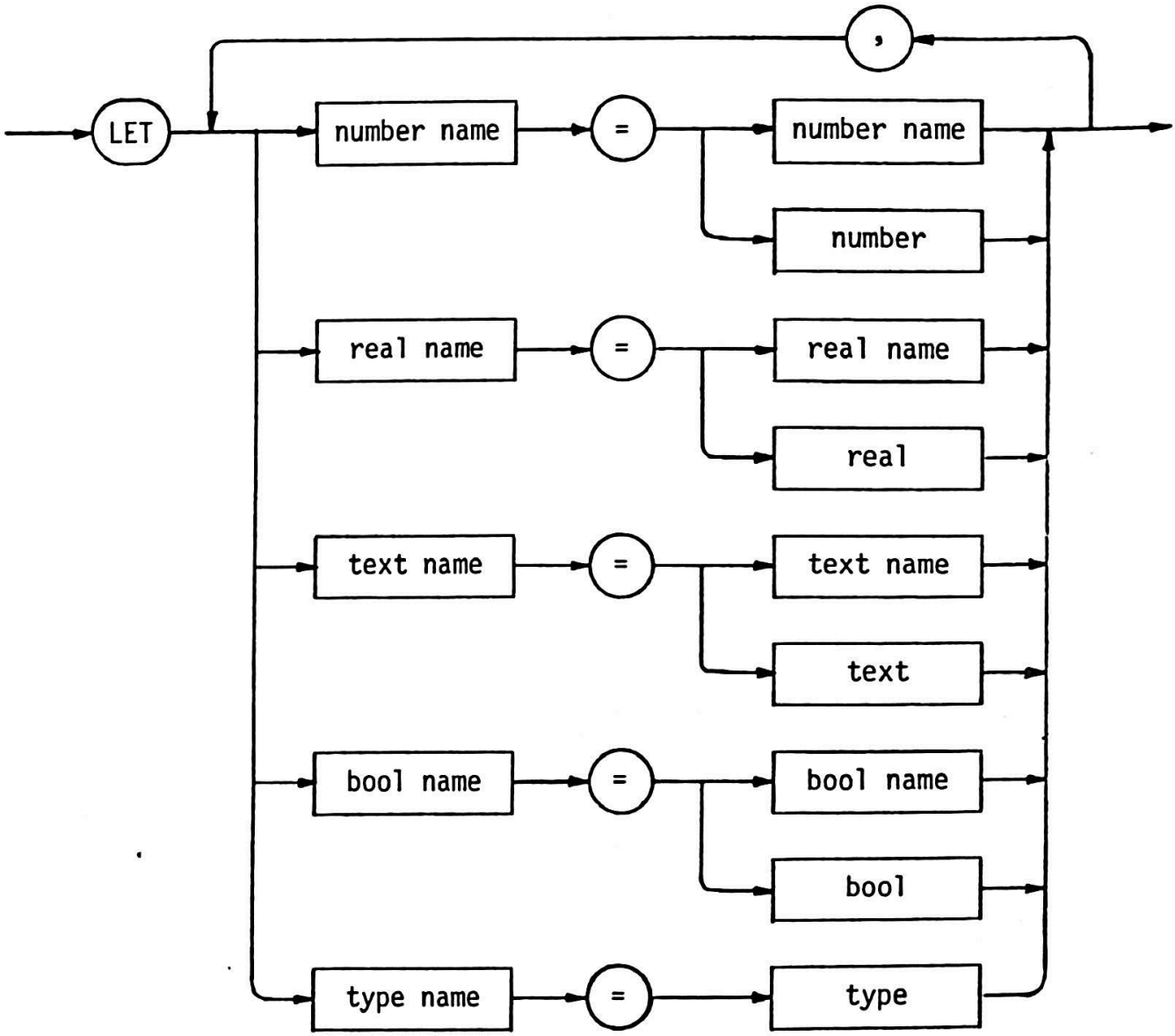
routine body



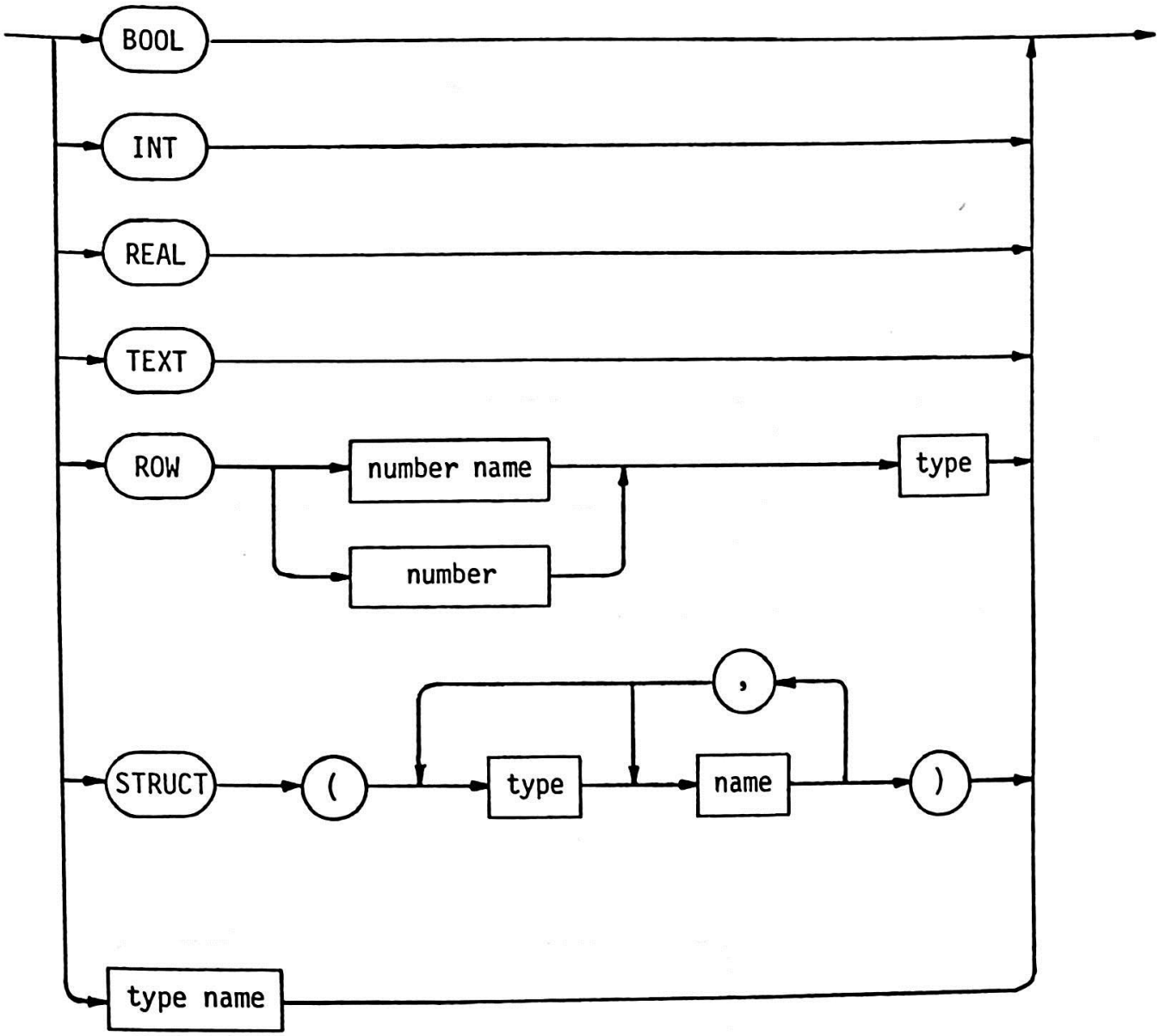
type declaration



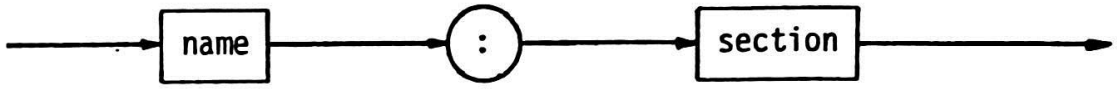
shorthand declaration



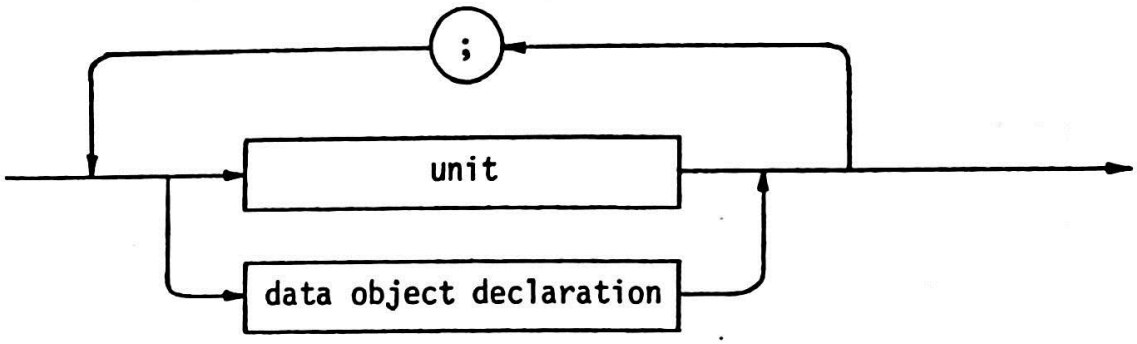
type


BOOL
INT
REAL
TEXT
ROW
number name
number
type
STRUCT
(
,
type
name
)
type name

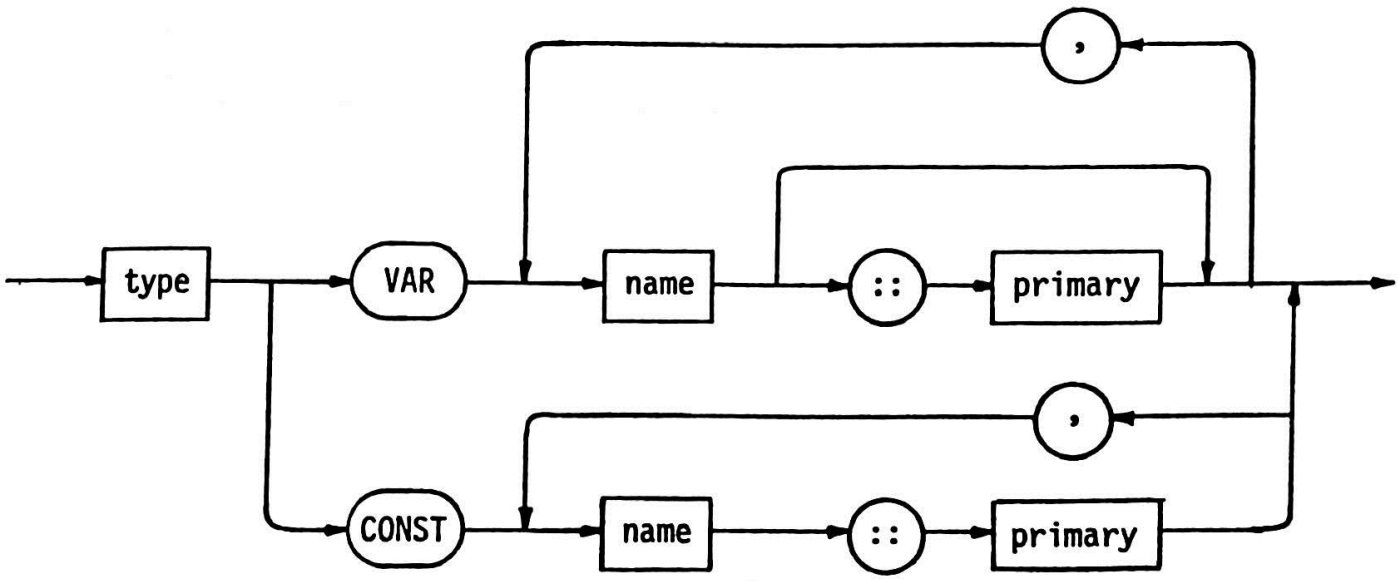
refinement



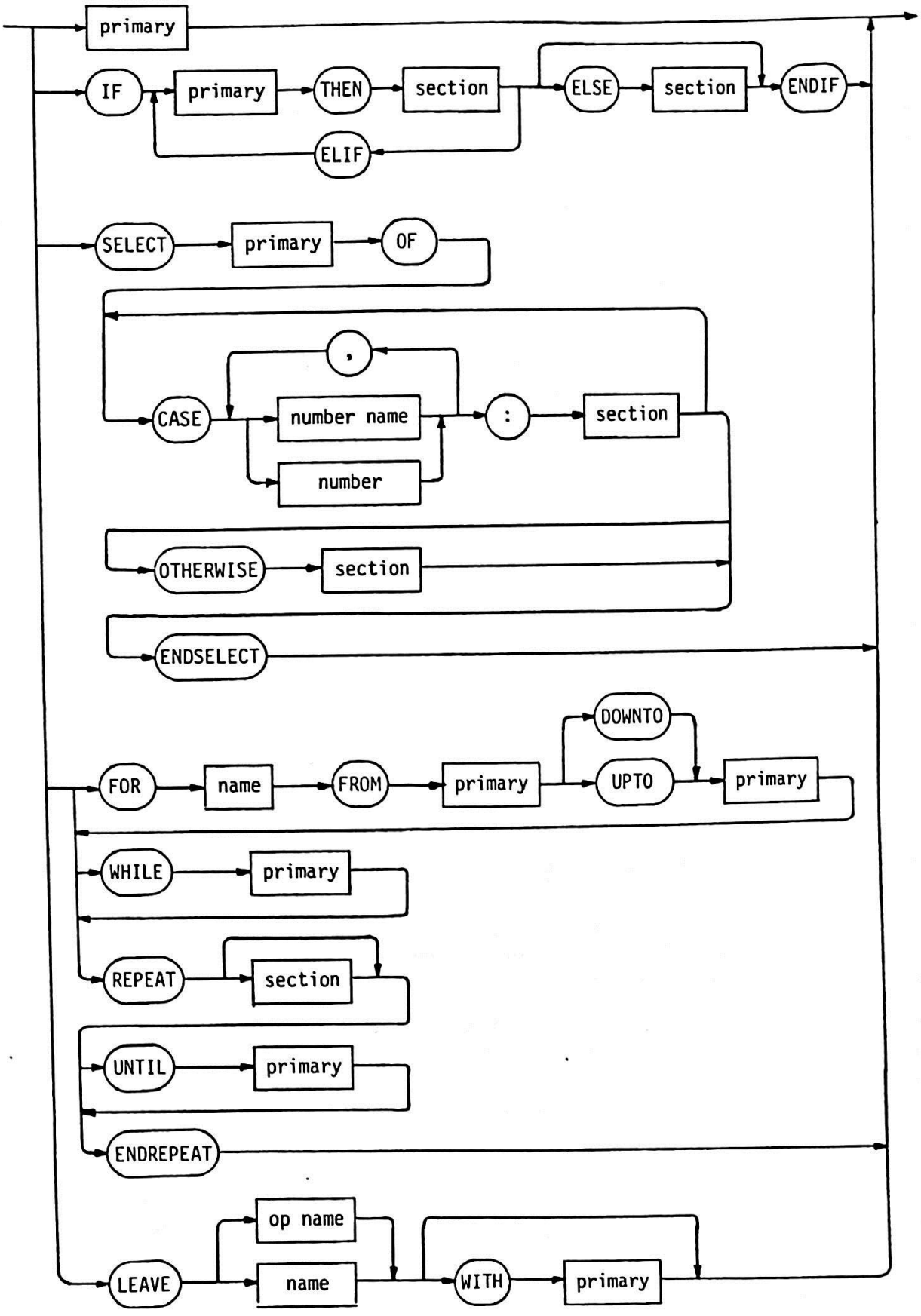
section



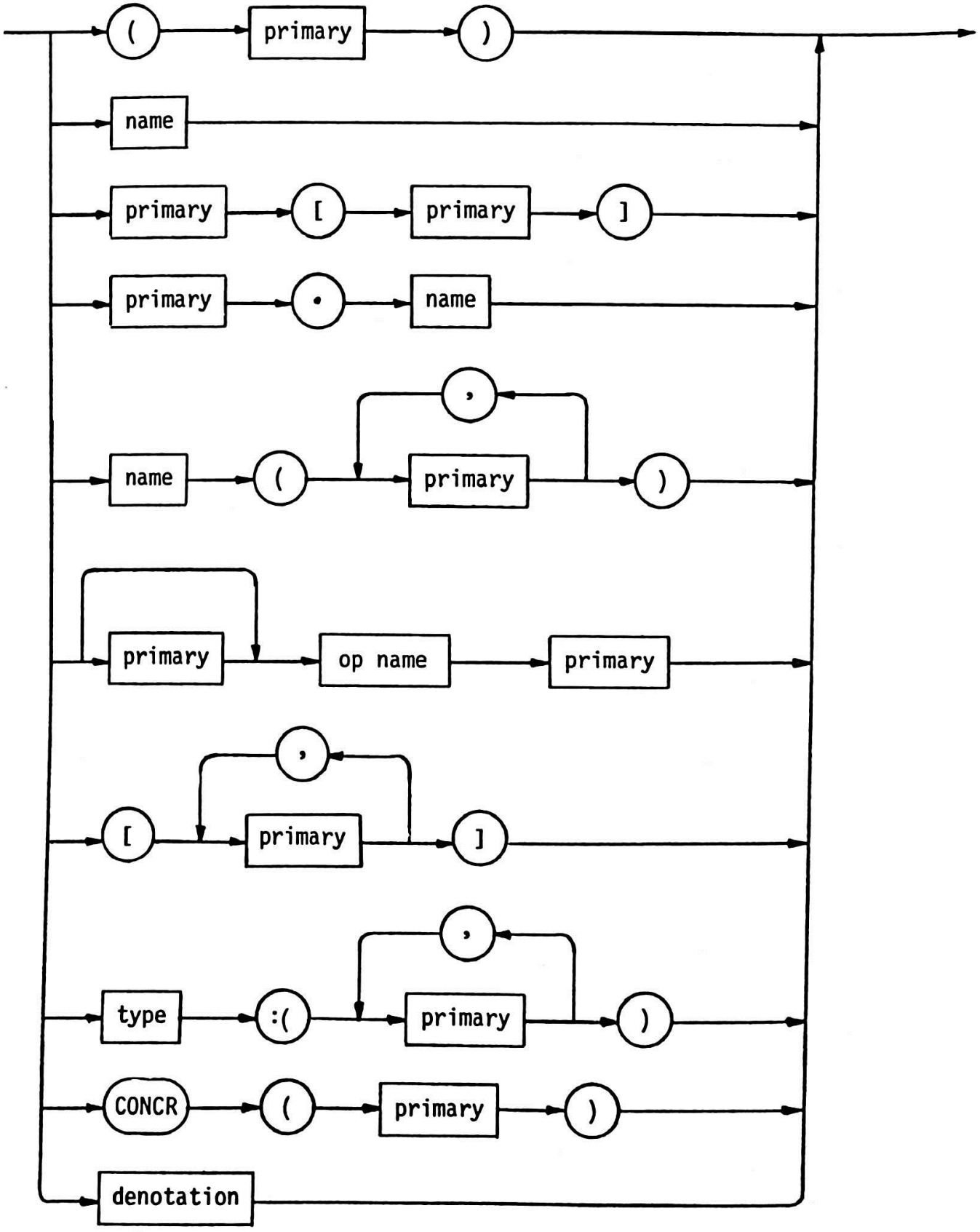
data object declaration



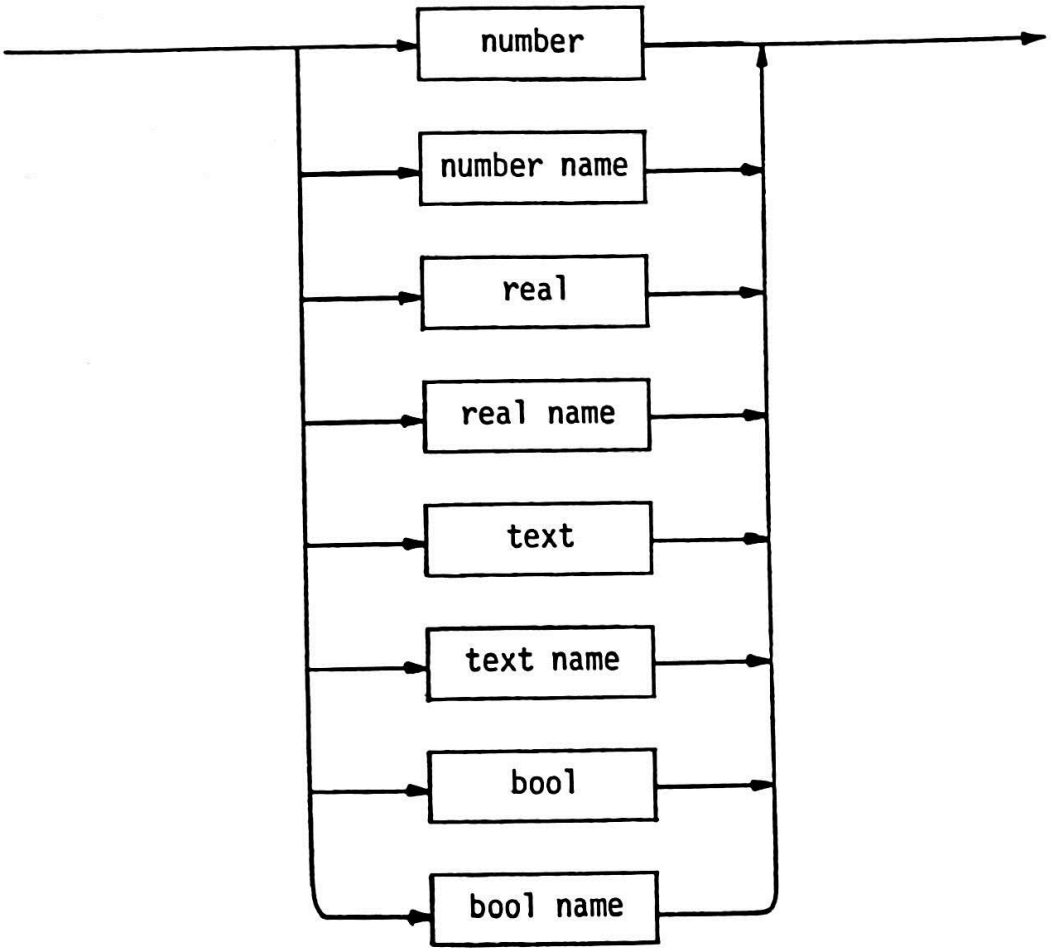
unit


primary
IF
primary
THEN
section
ELSE
section
ENDIF
ELIF
SELECT
primary
OF
,
CASE
number name
:
section
number
OTHERWISE
section
ENDSELECT
DOWNTO
FOR
name
FROM
primary
UPTO
primary
WHILE
primary
REPEAT
section
UNTIL
primary
ENDREPEAT
op name
LEAVE
name
WITH
primary

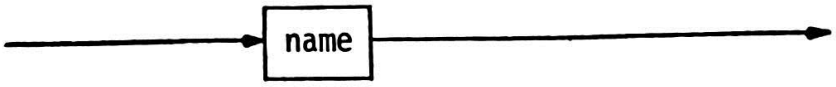
primary


(
primary
)
name
primary
[
primary
]
primary
.
name
,
name
(
primary
)
primary
op name
primary
,
[
primary
]
,
type
:(
primary
)
CONCR
(
primary
)
denotation

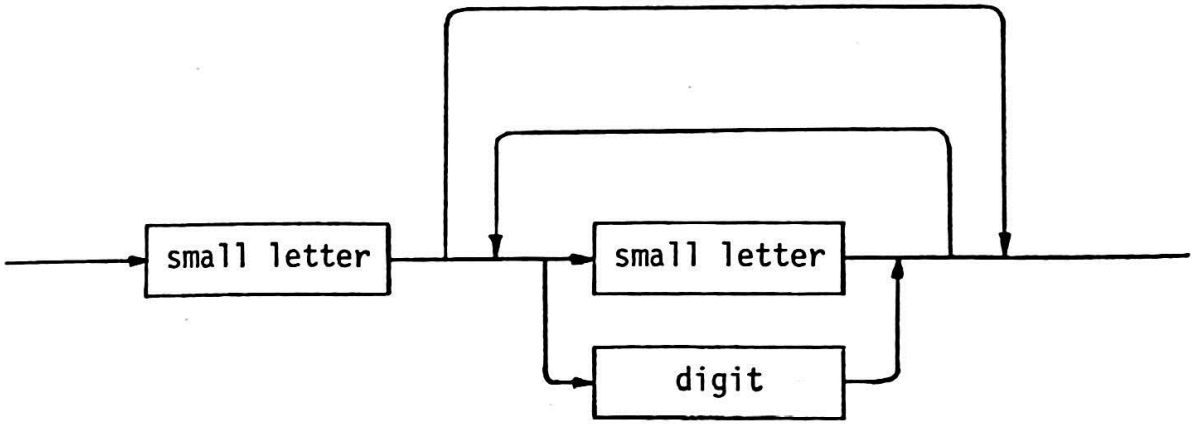
denotation



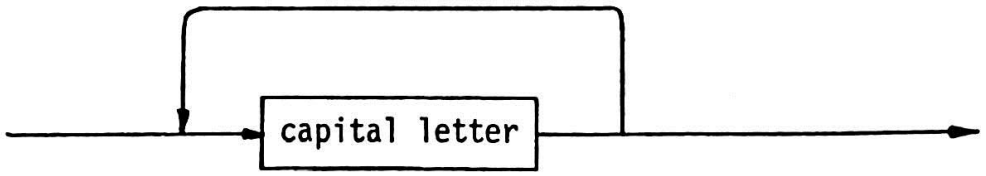
number name, real name, text name, bool name



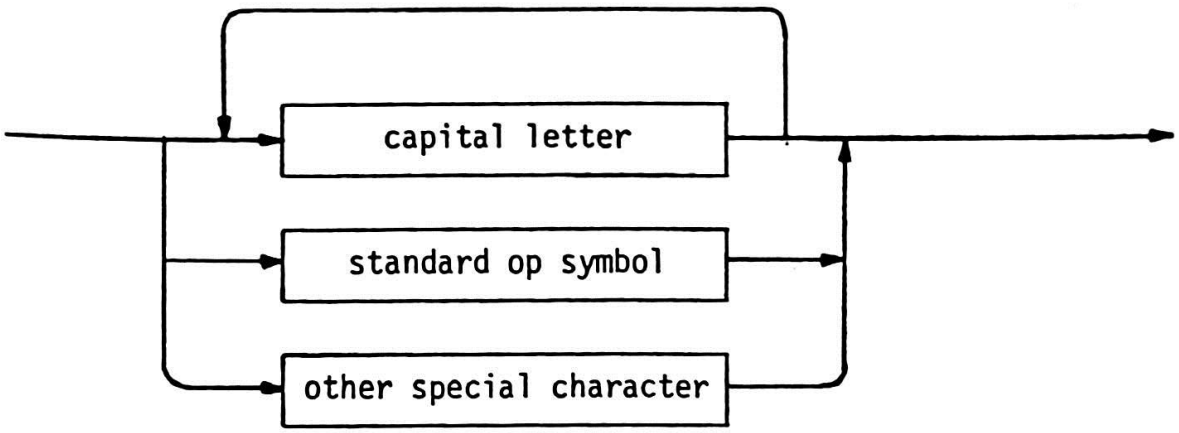
name



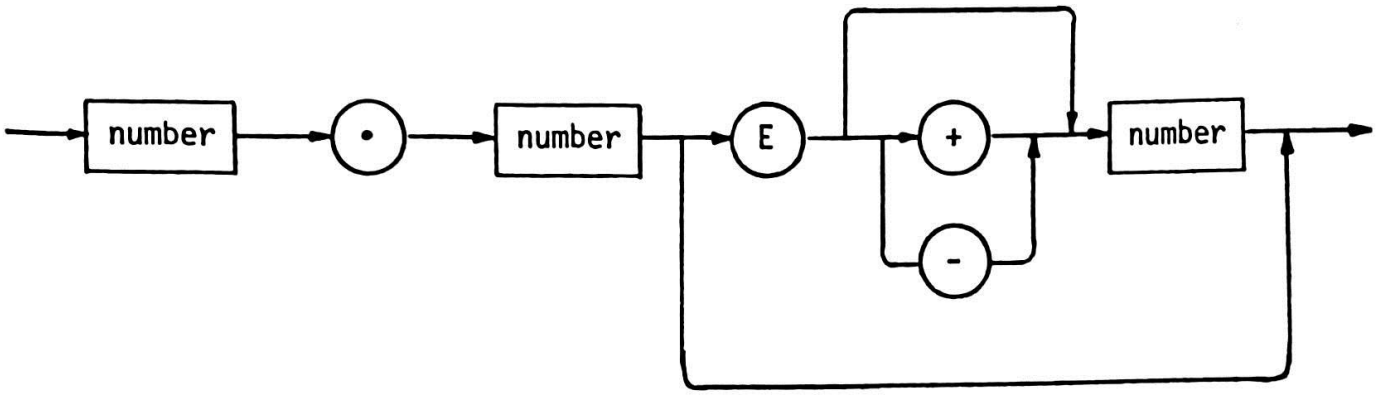
type name



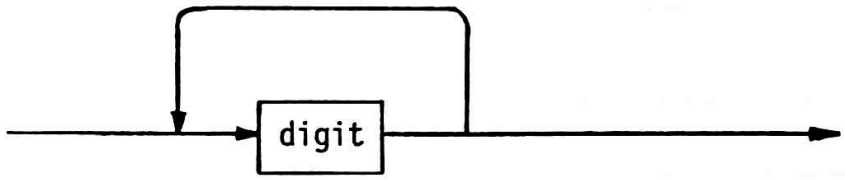
op name


capital letter
standard op symbol
other special character

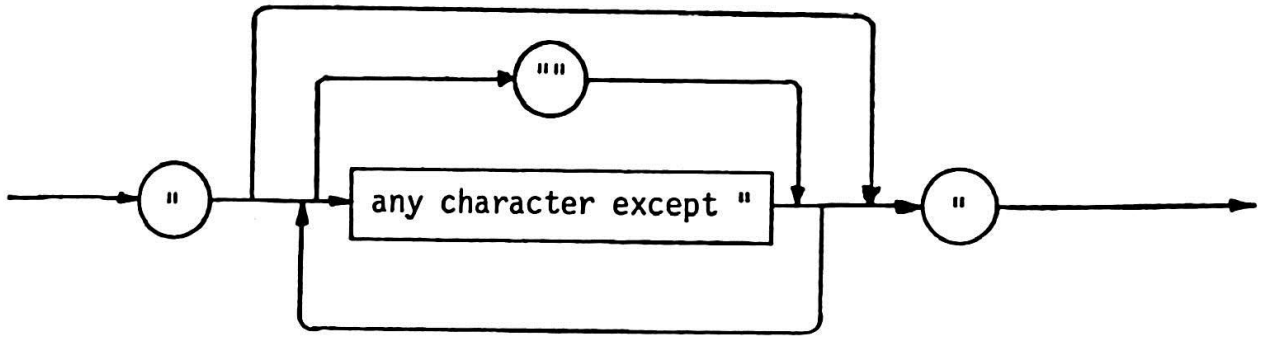
real


number
.
number
E
+
-
number

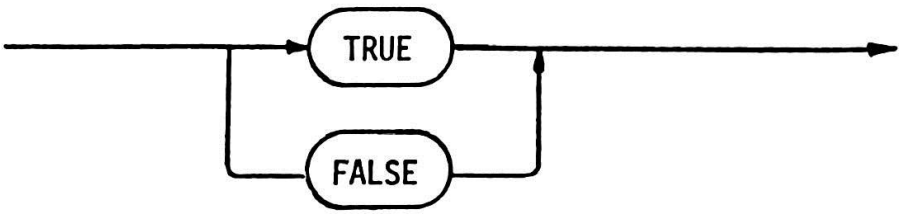
number



text



bool



Literaturverzeichnis

[Wijngaarden75] A. v.Wijngaarden (Ed.): Revised Report on the Algorithmic Language ALGOL68. Acta Informatica, vol. 5 # 1-3 (1975), (auch als Reprint, Springer Verlag 1976).

[Cleaveland77] J.C. Cleaveland, R.C. Uzgalis: Grammars for Programming Languages. American Elsevier, New York, N.Y. 1977.

[Peck74] J.E.L. Peck: Two-Level Grammars in Action. in Information Processing 74, Proceedings of IFIP-Congress 1974, Stockholm, North Holland Publ. Company 1975.

[Koster74] C.H.A. Koster: Beating the Global. IFIP WG 2.4, Working paper, La Grande Motte, Mai 1974.

[Koster76] C.H.A. Koster: Visibility and Types. ACM Conference on Data: Abstration, Definition and Structure, Salt Lake City, Maerz 1976, SIGPLAN Notices, special issue.

[Hommel76] G. Hommel, S. Jaehnichen, W. Koch: SLAN - Eine erweiterbare Sprache zur Unterstuetzung der strukturierten und modularen Programmierung. 4. GI-Fachtagung Programmiersprachen, Erlangen, Springer Verlag, Maerz 1976.

[Hommel77] G. Hommel, C.H.A. Koster: Systematisches Programmieren mit ELAN. TU Berlin, Fachbereich 20 (Informatik), Bericht 77-11, 1977.

[Bittner79] M. Bittner, J. Jaeckel, S. Jaehnichen: ELAN-Beispiel-sammlung. TU Berlin, Fachbereich 20 (Informatik), Forschungsgruppe Softwaretechnik, 1979.

[Heyderhoff76] P. Heyderhoff: Einfuehrung in ELAN. GMD Informatik-Kolleg, Bonn, Maerz 1976.

[Bartling79] U. Bartling,R. Hahn,J. Liedtke: ELAN-Benutzerhandbuch. Hochschulrechenzentrum der Universitaet Bielefeld, 1979.

INDEX I: Syntaxregeln

INDEX II: Standard-Bezeichner

INDEX III: Stichwortverzeichnis