Benutzerhandbuch

Programmierung

Stand: 1.7.87

Autoren: Irmgard Ley
Werner Metterhausen u.a.

Dieser Text wurde mit der EUMEL – Textverarbeitung erstellt und aufbereitet und mit dem AGFA Laserdrucksystem P400 gedruckt.

Gesellschaft für Mathematik und Datenverarbeitung mbH

Bereich/Location Birlinghoven
Schloß Birlinghoven
Postfach 1240
D – 5205 Sankt Augustin 1

Telefon (02241) 14 – 1
Telex 8 89 469 gmd d
Telefax (02241) 14 28 89
Teletex 2627 – 224135 = GMDVV
Bildschirmtext *43900#

TEIL 1 : Einleitung

TEIL 2 : ELAN

TEIL 3 : Der Editor

TEIL 4 : Kommandosprache

TEIL 5 : Programmierung

TEIL 6 : Das Archiv 'std zusatz'

Anhang : ELAN – Syntaxdiagramme

INDEX

TEIL 1 : Einleitung

1.1 Allgemeines über EUMEL

Dieses Buch bietet eine Übersicht über die Standardprozeduren des Betriebssystem EUMEL. Es bietet damit sowohl Hilfestellung für die Benutzung der standardmäßig vorhandenen Kommandos als auch für die Programmierung, also die Erweiterung dieses Kommandovorrats. Es ist jedoch kein Lehrbuch der Programmierung!

In den ersten drei Kapiteln dieses Programmierhandbuches werden einige Grundbegriffe des Systems, die grundlegende Programmiersprache (ELAN) und der EUMEL – Editor erläutert.

Das vierte Kapitel bietet eine Übersicht über diejenigen Prozeduren und Operatoren, die eher der 'Job – Control – Language' zugerechnet werden können, also häufig im Kommandodialog benutzt werden.

Im fünften Teil sind diejenigen Operationen beschrieben, die meistenteils für die Programmierung benutzt werden. (Compiler, Operationen auf den elementaren Datentypen, Dateien, Ein – und Ausgabe usw.).

Diese Trennung ist jedoch recht willkürlich, es ist ja gerade eine der wichtigen Eigenschaften dieses Betriebssystems, daß es keine Trennung zwischen der Kommandosprache des Betriebssystems und Programmmiersprache für das System gibt. Jedes Systemkommando ist Aufruf einer ELAN Prozedur, jede neue Prozedur stellt eine Erweiterung des Kommandovorrats des Systems dar.

Aus Gründen der Übersichtlichkeit der Zusammenstellung ist dieses Buch nicht frei von Vorwärtsverweisen!

1.2 Struktur des Betriebssystems EUMEL

Grundlegend für das Verständnis des Betriebssystems EUMEL ist der Begriff der Task. Eine Task kann als theoretisch unendliche Wiederholung eines Systemprogramms der Form:

'nimm Kommando entgegen'
'verarbeite Kommando'

aufgefaßt werden. Einige Tasks existieren bereits als Grundstock des Systems, weitere werden von Benutzern des Systems erschaffen und dienen als persönliche Arbeitsumgebung für den 'Eigentümer'. Eine Task kann als benutzereigener, unabhängiger Computer im Computer betrachtet werden, denn sie kann Kommandos entgegennehmen und ausführen und Daten verwalten und aufbewahren.

Eine Task kann neu erzeugt werden, an einen Bildschirm gekoppelt werden und beendet werden.

Das Tasksystem ist in einer baumartigen Struktur angeordnet. Außer der Wurzel 'UR' hat jede Task einen Vorgänger ('Vater – Task') und möglicherweise Nachfolger ('Sohn – Tasks').

Task – Organisation

```
SUPERVISOR
    -
  SYSUR
      ARCHIVE
      configurator
      OPERATOR
      shutup

UR
  PUBLIC
      Benutzertask1
      Benutzertask2
          Benutzertask3
        .....
```

Jeder Benutzer arbeitet innerhalb eines EUMEL – Systems, indem er eine Task an sein Terminal koppelt und dort Programme aufruft.

Dateien sind grundsätzlich Eigentum einer Task. Es ist grundlegend für das Verständnis des Betriebssystems EUMEL, die Beziehung zwischen Tasks und Dateien zu erkennen.

Eine Task ist ein Prozeß, der gegebenenfalls Dateien besitzt. Dateien können nur in einer Task existieren. Um eine Datei einer anderen Task zur Verfügung zu stellen, wird eine Kopie der Datei an die andere Task geschickt, die sendende Task ist danach Eigentümer des 'Originals', die empfangende Task Eigentümer der 'Kopie'.

Soll eine Hierarchie von Dateien aufgebaut werden, so ist sie über eine Hierarchie von Tasks zu realisieren, da in einer Task alle Dateien gleichberechtigt sind.

Bis zu dieser Stelle war stets von Dateien die Rede. Dateien sind jedoch ein Spezialfall der grundlegenderen Struktur des Datenraumes.

Ein Datenraum ist ein allgemeiner Datenbehälter. Ein Datenraum kann beliebige Daten aufnehmen und erlaubt direkten Zugriff auf diese Daten. Die Struktur der Daten im Datenraum unterscheidet sich nicht von der Struktur der Programmdaten. Der 'innere Datentyp' eines Datenraums wird vom Programmierer festgelegt.

Vorgeprägt vom System gibt es Textdateien, jeder andere Datentyp muß vom Programmierer geprägt werden, um so Dateien erzeugen zu können, die Objekte eben dieses neuen Typs enthalten.

1.3 Eigenschaften des Betriebssystems

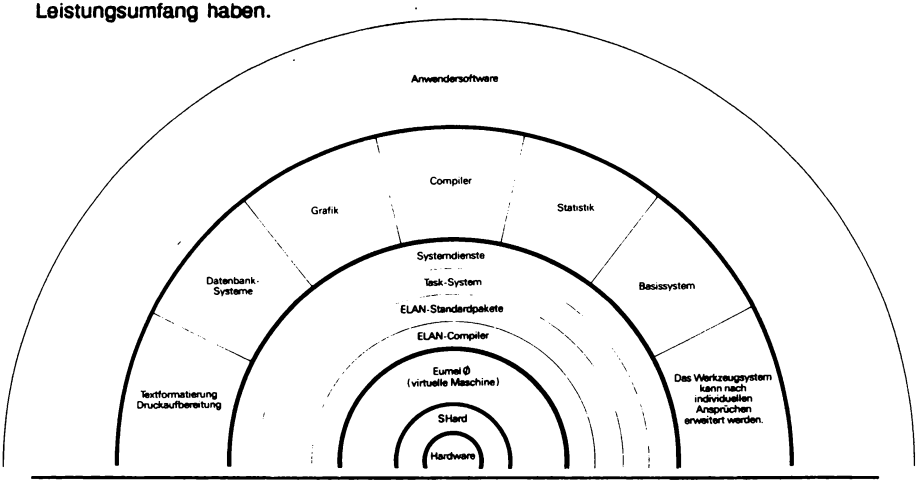
Der erste Entwurf des Mikroprozessor – Betriebssystems EUMEL (Extendable multi User Microprozessor ELAN system) entstand 1979 mit dem Anspruch, auf Mikrocomputern den Anwendern Hilfsmittel und Unterstützungen zu bieten, wie sie sonst nur auf Großrechnern zur Verfügung gestellt werden.

Aspekte, die EUMEL von anderen Betriebssystemen für Mikrocomputer unterscheiden, sind:

- Hardwareunabhängigkeit
- Multitaskingkonzept
- Multiuserbetrieb
- Erweiterbarkeit
- virtuelle Speicherverwaltung
- Datensicherheit

Das EUMEL – Schichtenmodell

Die Hardwareunabhängigkeit des Betriebssystems EUMEL begründet sich in seinem Aufbau aus Schichten (sogenannten virtuellen Maschinen), die einen klar definierten Leistungsumfang haben.



Jede Schicht erwartet und erhält von ihren Nachbarn wohldefinierte Eingaben und gibt wohldefinierte Ausgaben weiter. Änderungen in einer Schicht müssen also in den angrenzenden Schichten beachtet werden, aber nicht in allen Teilen des Systems.

Um EUMEL auf Rechner mit einem neuen Prozessortyp zu portieren, wird zunächst eine auf die Eigenheiten des Prozessors abgestimmte EUMEL0 – Maschine entworfen und eine Hardwareanpassung (SHard : Software/Hardware – Interface) für einen Rechner mit diesem Prozessor hergestellt. Alle höheren Schichten des Systems bleiben unberührt. Weitere mit diesem Prozessortyp ausgestattete Rechner können mit EUMEL betrieben werden, indem ein SHard für dieses Rechnermodell geschrieben wird.

Aus Benutzersicht ist wichtig, daß dadurch jegliche Software, die auf irgendeinem Rechner unter EUMEL verfügbar ist, auf jedem anderen Rechner, für den eine EUMEL Portierung existiert, lauffähig ist und gleiches Verhalten zeigt. Eine Vernetzung beliebiger Rechner, auf die EUMEL portiert ist, ist problemlos möglich.

Desweiteren ist für den Benutzer des Systems von Bedeutung, daß er von der hardwarenahen Schicht entfernt ist. Weder die Programmiersprache noch irgendwelche speziellen Systemfunktionen gewähren direkten Zugriff auf den Speicher oder Registerinhalte. Diese Tatsache hat weitreichende Folgen in Hinsicht auf Datenschutz und Systemsicherheit.

Multi – Tasking – /Multi – User – Betrieb

Wie einleitend dargestellt, besteht ein EUMEL – System aus diversen Tasks. Durch eine Aufteilung der Prozessorzeit in Zeitscheiben ist eine (quasi) parallele Bedienung mehrerer Tasks möglich.

Die multi – user – Fähigkeit des Betriebssystems wird durch den Anschluß mehrerer Bildschirmarbeitsplätze (Terminals) an V.24 Schnittstellen des Rechners erreicht. Dabei wird jeder Schnittstelle eine sogenannte Kanalnummer zugeordnet. Jeder Benutzer kann seine Task dann an einen Kanal (= Terminal) koppeln und an diesem Terminal gleichzeitig mit anderen Benutzern arbeiten.

Prozeßkommunikation und Netzwerkfähigkeit

Grundlage der Kommunikation ist die 'Manager – Eigenschaft' von Tasks. Eine Task ist 'Manager', wenn sie Aufträge anderer Tasks annehmen und ausführen kann. Insbesondere kann ein Manager veranlaßt werden, eine an ihn geschickte Datei anzunehmen, bzw. eine ihm gehörende Datei an die fordernde Task zu schicken.

Derartige Kommunikationslinien verlaufen normalerweise in der Baumstruktur des Systems: z.B. ist die Task 'PUBLIC' (vergl. Seite 2) grundsätzlich Manager – Task. Eine unterhalb von PUBLIC liegende Task kann eine Datei an PUBLIC senden, bzw. von PUBLIC holen.

Es ist auch möglich, eine Task für den Zugriff beliebiger anderer Tasks zu öffnen und somit beliebige Kommunikationspfade aufzubauen. Prinzipiell ist damit auch schon der Aufbau eines Netzwerkes beschrieben, denn sendende und empfangende Tasks können sich auf verschiedenen Rechnern befinden.

Durch selbst erstellte Programme kann der Eigentümer einer 'Manager – Task' die Reaktion dieser Task auf einen Auftrag von außen bestimmen. Beispielsweise kann ein Manager derart programmiert werden, daß er nur Dateien empfängt und ausdruckt, aber niemals Dateien verschickt (Spool – Task).

Erweiterbarkeit

Die Programmiersprache ELAN ist im EUMEL – System gleichzeitig Programmier – und System – Kommandosprache (JCL), denn jedes Kommando ist Aufruf einer ELAN – Prozedur und jede vom Benutzer geschriebene ELAN – Prozedur erweitert den Kommandovorrat des Systems.

Da alle EUMEL – Werkzeuge (einschließlich Editor) selbst ELAN – Programme sind, kann das System vom Benutzer selbst durch Hinzufügen eigener ELAN – Programme oder Programmpakete beliebig erweitert werden. Dabei können die bereits implementierten Systemteile (z.B. die Fenstertechnik des Editors) genutzt werden.

Ein Benutzer muß, um alle Möglichkeiten vom EUMEL zu nutzen, nur eine Sprache lernen und nicht – wie bei anderen Betriebssystemen – zwei unterschiedliche, eine Kommando – und eine Programmiersprache.

ELAN selbst ist eine PASCAL – ähnliche Programmiersprache, die mit Hilfe der schrittweisen Verfeinerung (Refinement – Konzept) die Top – Down – Programmierung unterstützt. Das Paketkonzept, das der Modularisierung dient, und die freie Wahl von Bezeichnernamen sind Voraussetzung für übersichtliche und effiziente Programmierung.

Virtuelle Speicherverwaltung

Im EUMEL – System wird der Hauptspeicherplatz nach dem **Demand – Paging – Prinzip** verwaltet. Daten und Programme werden dazu in Seiten von 512 Byte aufgeteilt. Nur diejenigen Seiten, die wirklich benötigt werden, werden vom Hintergrundspeicher (Platte) in den Hauptspeicher geholt. Damit ist für den Benutzer bezüglich seiner Programm – bzw. Dateigrößen nicht mehr der Hauptspeicher, sondern die Hintergrundkapazität von Bedeutung. Die Durchsatzgeschwindigkeit (Performance) ist abhängig von der Größe des RAM – Speichers und der Zugriffsgeschwindigkeit des Hintergrundmediums. Das Demand – Paging – Verfahren ist Grundlage für den Multi – User – Betrieb, wobei der Hauptspeicherplatz möglichst effizient zu nutzen und kein Benutzer zu benachteiligen ist.

Beim Duplizieren eines Datenraumes wird im EUMEL – System lediglich eine logische, keine physische Kopie erzeugt. Zwei Seiten (zweier Datenräume) heißen dann gekoppelt (geshared), wenn beide Seiten physisch demselben Block zugeordnet sind. Erst bei einem Schreibzugriff werden die Seiten entkoppelt (entshared) und tatsächlich physisch kopiert. Daher der Name **"copy – on – write"**.

Dieses Prinzip wird natürlich auch systemintern angewandt. Beispielsweise erbt eine Sohn – Task den Kommandovorrat der Vater – Task, indem der Standard – Datenraum, der die vorübersetzten ELAN – Prozeduren enthält, in der beschriebenen Weise kopiert wird. Prozeduren, die später hinzugefügt werden, werden natürlich nicht vererbt, da die Standard – Datenräume dann entkoppelt werden.

Datensicherheit

Störungen (inklusive Stromausfall) werden systemseitig durch eine automatische **Fixpoint – Rerun – Logik** aufgefangen, indem zum Zeitpunkt eines Fixpunkts der Inhalt des RAM Speichers, der seit dem letzten Fixpunkt verändert wurde auf den permanenten Speicher (Festplatte) geschrieben wird. Somit kann nach einer Störung immer auf den Systemzustand des letzten Fixpunktes aufgesetzt werden und die Datenverluste halten sich in erträglichen Grenzen.

Der Zeitraum zwischen zwei Fixpunkten beträgt standardmäßig 15 Minuten, kann aber vom Benutzer anders eingestellt werden.

Auch bei dieser Sicherung wird das Copy – on – write – Prinzip angewendet, so daß Platz – und Zeitaufwand gering sind und den normalen Ablauf nicht stören.

1.4 Wichtige Begriffe

- **archive.** Spezielle Task zur Verwaltung des Diskettenlaufwerks. Da für die längerfristige Datenhaltung und zur zusätzlichen Datensicherung Dateien auf Disketten geschrieben werden, besitzt das EUMEL – System für diese Aufgabe eine besondere Task, die die Bedienung vereinfacht und exklusiven Zugriff auf das Laufwerk garantiert.

- **configurator.** Besondere Task im Systemzweig des EUMEL – Systems. In dieser Task ist die Konfiguration von Kanälen möglich, d.h. Kanal und angeschlossenenes Gerät werden aufeinander abgestimmt.

- **editor.** Programm zur Dateibearbeitung am Bildschirm. Das Programm wird durch das (Monitor –) Kommando 'edit' und die Eingabe des Namens der gewünschten Datei als Parameter gestartet.

  Da ein Bildschirm normalerweise auf 80 Zeichen Zeilenbreite und 24 Zeilen beschränkt ist, kann der Editor als Fenster betrachtet werden, das über die möglicherweise weitaus größere Datei bewegt wird und durch das der betrachtete Ausschnitt der Datei bearbeitet werden kann.

- **manager task.** Task, die Aufträge von anderen Tasks entgegennehmen und ausführen kann. Beispielsweise ist die Verwaltung von Dateien, die mehreren Benutzern (= anderen Tasks) zugänglich sein sollen, eine typische Aufgabe für einen Manager.

- **Monitor.** Der Empfänger von Kommandos innerhalb einer Task ist der Monitor. Der Monitor ist sichtbar durch eine Zeile, in der 'gib kommando' steht. In diese Zeile werden Kommandos und erforderliche Parameter eingegeben.

- **Supervisor.** Spezielle Task zur Überwachung eines EUMEL – Systems. Ein Benutzer kann durch die Supervisor – Kommandos Leistungen von dieser Task fordern: neue Task einrichten, Task wiederaufnehmen und diverse Informationen.

- **Task.** Beliebig langlebiger Prozeß im EUMEL – System, der die Arbeitsumgebung für Benutzer bildet. Jede Task besitzt einen Standard – Datenraum, der Code und Compilertabellen der Task enthält und kann weitere Datenräume (Dateien) besitzen.

1.5 Die Notation in diesem Buch

Beachten Sie bitte folgende Regeln der Aufschreibung:

- Funktionstasten werden ebenso wie besondere Tastenkombinationen explizit als Tasten dargestellt:

  SV ESC e

  oder im Text: CR

- Alles, was Sie am Bildschirm Ihres Rechners schreiben oder lesen sollen, ist in Textbereiche, die einen Bildschirm darstellen, eingefaßt.

  Beispiel:

```
gib kommando:
edit ("mein programm")
```

- Innerhalb des Handbuchs sind in der Aufschreibung die Konventionen der Programmiersprache ELAN berücksichtigt. Dabei sind folgende Besonderheiten zu beachten:

  1) Kommandos werden grundsätzlich klein geschrieben.

  2) Dateinamen u.ä. sind Textdenoter und werden somit in Klammern und Anführungsstriche gesetzt. In diesem Buch steht an den Stellen, wo ein Dateiname auftaucht *'dateiname'* ; den Namen, den Sie tatsächlich verwenden, können Sie frei wählen.

  3) Falls besondere Begriffe oder Beispiele innerhalb eines normalen Textes auftreten, werden sie in einfache Anführungsstriche gesetzt.

1.6 Die Funktionstasten des EUMEL – Systems

Die Lage der EUMEL – Funktionstasten entnehmen Sie bitte der speziellen Installationsanleitung zu dem von Ihnen benutzten Gerät.

Positionierungstasten

SHIFT — Umschalttaste

CR — Eingabe – / Absatztaste

ESC — Kommandotaste

SV — Supervisortaste

HOP — Verstärkertaste

RUBOUT — Löschtaste

RUBIN — Einfügetaste

TAB — Tabulatortaste

MARK — Markiertaste

STOP — Stoptaste

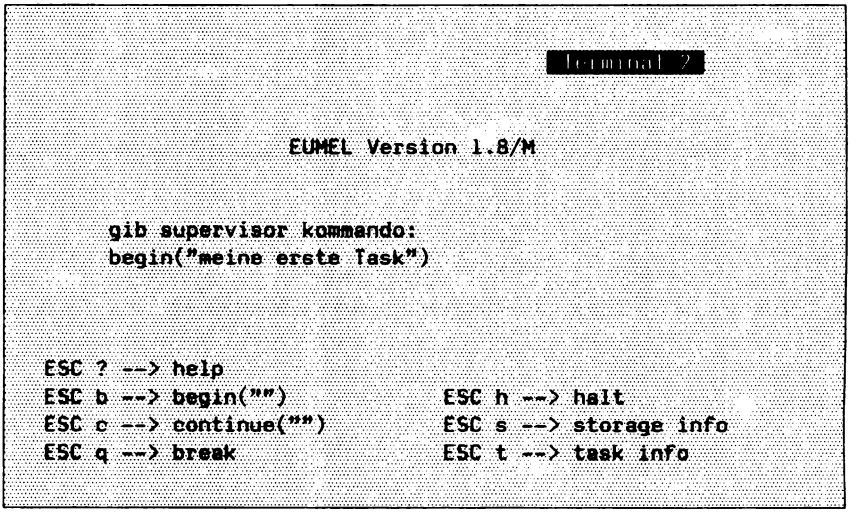
WEITER — Weitertaste

Weitere Informationen hierzu finden Sie in der Installationsanleitung zu dem von Ihnen benutzten Rechner oder Terminal.

1.7 Eine Beispielsitzung

Im Folgenden wird eine Beispielsitzung skizziert, in der ein ELAN – Programm erstellt und getestet wird.

SV SUPERVISOR aufrufen



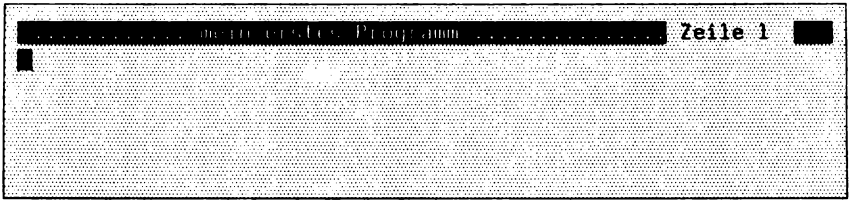
Durch das Kommando 'begin ("meine erste Task")', welches durch CR abgeschlossen werden muß, wird eine Task mit dem Namen 'meine erste Task' im Benutzerzweig, also unterhalb von 'PUBLIC' angelegt. Würde diese Task bereits existieren, so könnten Sie sie mit 'continue ("meine erste Task")' an das Terminal holen.

```
gib kommando :
edit ("mein erstes Programm")
```

In der Task eröffnen Sie eine Datei mit dem Kommando 'edit ("dateiname")'.

```
gib kommando :
edit ("mein erstes Programm")
"mein erstes Programm" neu einrichten (j/n) ? j
```

Falls diese Datei neu ist, erfolgt eine Kontrollfrage (zur Kontrolle der gewünschten Schreibweise des Dateinamens), die Sie durch j bejahen.



In die noch leere Datei tippen Sie nun den Programmtext ein.

```
.............. mein erstes Programm ..................  Zeile 1
INT PROC ggt (INT CONST a, b):
  INT VAR b kopie := abs (b), a kopie := abs (a);
   WHILE b kopie <> 0 REPEAT
     INT VAR rest := a kopie MOD b kopie;
     a kopie := b kopie;
     b kopie := rest
   END REPEAT;
  a kopie
END PROC gt;

REP
  lies 2 zahlen ein;
  gib groessten gemeinsamen teiler aus
UNTIL no ("weitertesten") PER.

lies 2 zahlen ein:
  line; put ("2 Zahlen eingeben:");
  INT VAR a, b;
  get (a); get (b).

gib groessten gemeinsamen teiler aus:
  put ("der größte gemeinsame Teiler von");
  put (a); put ("und"); put (b); put ("ist"); put (ggt (a,b));
  line.
```

In dem Programmbeispiel wird ein Prozedur 'ggt' definiert, die den größten gemeinsamen Teiler zweier Zahlen bestimmt. Die Prozedur soll für verschiedene Beispiele getestet werden; dies geschieht in dem Hauptprogramm, das solange Zahlen einliest und den größten gemeinsamen Teiler ausgibt, bis der Benutzer auf die Frage 'weitertesten (j/n) ?' mit **n** antwortet.

Haben Sie das Programm eingegeben, so können Sie die Bearbeitung dieser Programmdatei durch Drücken der Tasten **ESC** **q** (nacheinander!) beenden.

```
gib kommando :
run ("mein erstes Programm")
```

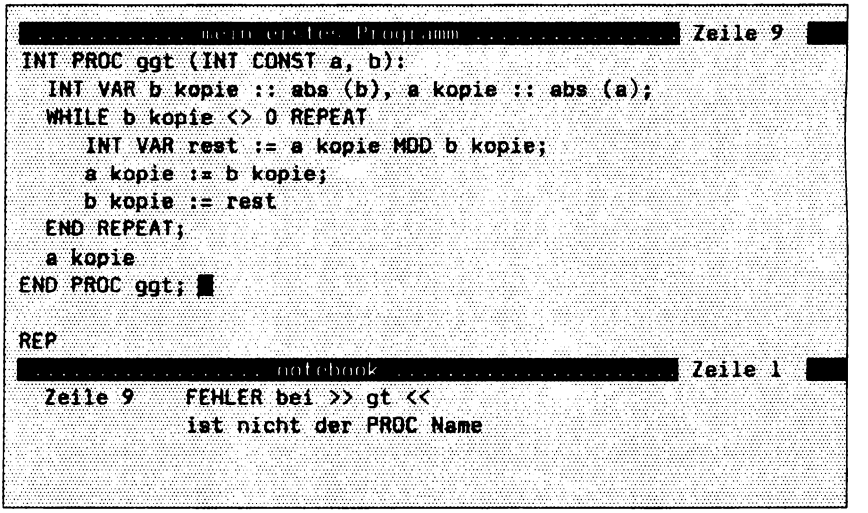
Um Ihr Programm zu übersetzen und auszuführen, geben Sie das Kommando 'run ("dateiname")'.

Der Verlauf der Übersetzung, die zwei Läufe über das Programm erfordert, ist am Zähler, der an der linken Seite des Bildschirms ausgegeben wird, zu erkennen.

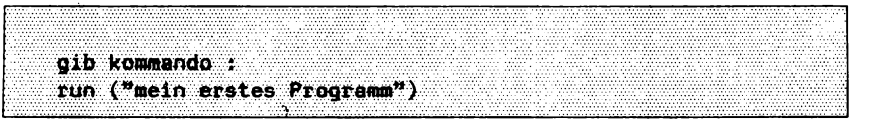
Werden beim Übersetzen des Programms Fehler entdeckt, so werden diese im 'notebook' parallel zur Programmdatei gezeigt. In dem Beispielprogramm wurde ein Schreibfehler in Zeile 9 gemacht.

```
.................. mein erstes Programm ..................  Zeile 1
INT PROC ggt (INT CONST a, b):
  INT VAR b kopie := abs (b), a kopie := abs (a);
  WHILE b kopie <> 0 REPEAT
    INT VAR rest := a kopie MOD b kopie;
    a kopie := b kopie;
    b kopie := rest
  END REPEAT;
  a kopie
END PROC gt;

REP
.................... notebook ....................  Zeile 1
  Zeile 9    FEHLER bei >> gt <<
             ist nicht der PROC Name
```

Diesen Fehler müssen Sie nun verbessern.



Haben Sie das Programm korrigiert, so können Sie die Datei durch Drücken der Tasten **ESC** **q** (nacheinander!) wieder verlassen.



Nach Eingabe von **CR** wird das Programm erneut übersetzt.

```
Keine Fehler gefunden,  136 B Code, 82 B Paketdaten generiert

      *******  ENDE DER UEBERSETZUNG  *******

2 Zahlen eingeben:
```

Das Programm war jetzt fehlerfrei. Nach der Übersetzung wurde die Ausführung gestartet. Nun können Beispiele getestet werden.

```
2 Zahlen eingeben: 125 250
der größte gemeinsame Teiler von 125 und 225 ist 25
weitertesten (j/n) ?
```

Beantwortet man die Frage mit **n**, so wird die Ausführung des Programms beendet.

```
 gib kommando :
```

Um die Arbeit in der Task zu beenden, geben Sie auch an dieser Stelle **ESC** **q** (nacheinander!) ein.

Nach Verlassen der Task ist wiederum die EUMEL – Tapete auf dem Bildschirm. Jede weitere Aktion wird wiederum von hier aus durch **SV** begonnen. Insbesondere vor dem Ausschalten des Geräts muß nach **SV** eine Task des priviliegierten Systemzweigs (oft: 'shutup') mit **ESC** **e** an das Terminal gekoppelt werden, in der das Kommando 'shutup' gegeben wird.

TEIL 2: ELAN

2.1 Besondere Eigenschaften von ELAN

Kerneigenschaften von ELAN sind das Paketkonzept und die Methode des Refinements.

Paketkonzept:
ELAN bietet die Möglichkeit, neue Datentypen sowie Prozeduren und Operatoren auf diesen Datentypen zu definieren. Eine solche Definition von Algorithmen und Datentypen kann zu einer logischen Einheit, einem Paket, zusammengefaßt werden. Pakete können in einer Task vorübersetzt werden und erweitern damit automatisch den Sprachumfang.

Methode des Refinements:
Die Methode des Refinements erlaubt das schrittweise Herleiten von Problemlösungen von der jeweils geeigneten Terminologie herunter zu den von ELAN standardmäßig angebotenen Sprachelementen. Durch diese Vorgehensweise wird in äußerst starkem Maße ein strukturierter Programmentwurf gemäß dem Top – Down – Prinzip gefördert.

Die Programmiersprache ELAN wird im EUMEL – System zu folgenden Zwecken eingesetzt:

- Systemimplementationssprache
- Kommandosprache
- Anwenderprogrammiersprache

2.2 Lexikalische Elemente

Unter lexikalischen Elementen einer Programmiersprache versteht man die Elemente, in denen ein Programm notiert wird.

In ELAN sind dies:

- Schlüsselwörter
- Bezeichner
- Sonderzeichen
- Kommentare

2.2.1 Schlüsselwörter

Einige Wörter haben in ELAN eine feste Bedeutung und können somit nicht frei gewählt werden. Solche Wörter werden im EUMEL – System in Großbuchstaben geschrieben, Leerzeichen dürfen nicht enthalten sein.

Beispiele:

```
VAR
INT
WHILE
```

Wie später beschrieben wird, gibt es in ELAN auch die Möglichkeit, neue Schlüsselwörter einzuführen.

2.2.2 Bezeichner

Bezeichner oder Namen werden benutzt, um Objekte in einem Programmtext zu benennen und zu identifizieren (z.B: Variablennamen, Prozedurnamen).

Namen werden in ELAN folgendermaßen formuliert:

Das erste Zeichen eines Namens muß immer ein Kleinbuchstabe sein. Danach dürfen bis zu 254 Kleinbuchstaben, aber auch Ziffern folgen. Zur besseren Lesbarkeit können Leerzeichen in einem Namen erscheinen, die aber nicht zum Namen zählen. Sonderzeichen sind in Namen nicht erlaubt.

Beispiele für **korrekte** Bezeichner:

```
das ist ein langer name
x koordinate
nr 1
```

Beispiele für **falsche** Bezeichner:

```
x*1
1 exemplar
Nr 1
```

2.2.3 Sonderzeichen

Sonderzeichen sind Zeichen, die weder Klein – oder Großbuchstaben, noch Ziffern sind. Sie werden in ELAN als Trennzeichen oder als Operatoren benutzt.

In ELAN gibt es folgende Trennungszeichen:

- das Semikolon (';') trennt Anweisungen
- der Doppelpunkt (':') trennt Definiertes und Definition
- der Punkt ('.') wird als Endezeichen für bestimmte Programmabschnitte, als Dezimalpunkt und als Selektor – Zeichen für Datenstrukturen benutzt
- das Komma (',') trennt Parameter
- Klammernpaare ('(', ')') werden zum Einklammern von Parameterlisten oder Teilausdrücken benutzt
- mit Anführungszeichen ('"') werden Text – Denoter umrahmt
- eckige Klammernpaare ('[', ']') werden zur Subskription benutzt.

Als Operatornamen sind folgende Sonderzeichen erlaubt:

- ein Sonderzeichen, sofern es nicht als Trennzeichen benutzt wird:
  ! $ % & ' * + - / < = > ? @ ^ ' ~
- eine Kombination von zwei Sonderzeichen. Diese Kombination muß jedoch bereits in ELAN existieren:
  := <= >= <> **

2.2.4 Kommentare

Kommentare dienen ausschließlich der Dokumentation eines Programms. Sie werden vom Compiler überlesen und haben somit keinen Einfluß auf die Ausführung eines Programms. Sie dürfen an beliebigen Stellen eines Programmtextes geschrieben werden, jedoch nicht innerhalb von Schlüsselwörtern und Namen. Ein Kommentar darf über mehrere Zeilen gehen. In ELAN sind Kommentare nur in wenigen Fällen notwendig, da Programme durch andere Mittel gut lesbar geschrieben werden können.

Ein Kommentar in ELAN wird durch Kommentarklammern eingeschlossen. Es gibt folgende Formen von Kommentarklammern:

```
(*      Kommentar    *)
{       Kommentar    }
#(      Kommentar    )#
```

Die letzte Version '`#(  Kommentar  )#`' wird im EUMEL-System nicht unterstützt; statt dessen gibt es noch folgende Möglichkeit:

```
#       Kommentar    #
```

Da bei der Kommentarkennzeichnung mit # für Kommentaranfang und -ende das gleiche Zeichen benutzt wird, ist eine Schachtelung hier nicht möglich.

2.3 Datenobjekte

Eine Klasse von Objekten mit gleichen Eigenschaften wird in Programmiersprachen Datentyp genannt. Dabei hat ein Datentyp immer einen Namen, der die Klasse von Objekten sinnvoll kennzeichnet. Als ein Datenobjekt wird ein Exemplar eines Datentyps (also ein spezielles Objekt einer Klasse) bezeichnet.

Datentypen sind in ELAN ein zentrales Konzept. Jedes der in einem ELAN – Programm verwandten Datenobjekte hat einen Datentyp; somit kann man Datentypen auch als Eigenschaften von Datenobjekten ansehen. Für jeden Datentyp sind nur spezielle Operationen sinnvoll. Man kann nun Compilern die Aufgabe überlassen zu überprüfen, ob stets die richtige Operation auf einen Datentyp angewandt wird.

2.3.1 Elementare Datentypen

Einige Datentypen spielen bei der Programmierung eine besondere Rolle, weil sie häufig benötigt werden.

In ELAN sind das die Datentypen für

- ganze Zahlen (INT)
- reelle Zahlen (REAL)
- Zeichen und Zeichenfolgen (TEXT)
- Wahrheitswerte (BOOL).

Diese Datentypen sind von der Sprache ELAN vorgegeben und werden elementare Datentypen genannt. Für effiziente Rechnungen mit elementaren Datentypen gibt es in den meisten Rechnern spezielle Schaltungen, so daß die Hervorhebung und besondere Rolle, die sie in Programmiersprachen spielen, gerechtfertigt ist. Zudem hat man Werte – Darstellungen (Denoter) innerhalb von Programmen für die elementaren Datentypen vorgesehen.

2.3.1.1 Denoter für elementare Datentypen

Die Darstellung eines Werts in einem Rechner zur Laufzeit eines Programms wird Repräsentation genannt. Wenn es eindeutig ist, daß es sich nur um die Repräsentation im Rechner handelt, spricht man kurz von Werten. Um mit Objekten elementarer Datentypen arbeiten zu können, muß es in einem Programm die Möglichkeit geben, Werte eines Datentyps zu bezeichnen (denotieren). Die Werte – Darstellungen, die in ELAN Denoter genannt werden, sind für jeden Datentyp unterschiedlich. Wie bereits erwähnt, haben alle Datenobjekte in ELAN (also auch Denoter) nur einen – vom Compiler feststellbaren – Datentyp. Aus der Form eines Denoters ist also der Datentyp erkennbar:

INT – Denoter:

Sie bestehen aus einer Aneinanderreihung von Ziffern.

Beispiele:

```
17
007
32767
0
```

Führende Nullen spielen bei der Bildung des Wertes keine Rolle (sie werden vom ELAN – Compiler überlesen). Negative INT – Denoter gibt es nicht. Negative Werte werden durch eine Aufeinanderfolge des monadischen Operators '–' (siehe 2.4.1.1) und eines INT – Denoters realisiert.

REAL – Denoter:

Hier gibt es zwei Formen:
Die erste besteht aus zwei INT – Denotern, die durch einen Dezimalpunkt getrennt werden.

Beispiele:

```
0.314159
17.28
```

Der Dezimalpunkt wird wie ein Komma in der deutschen Schreibweise benutzt. Negative REAL – Denoter gibt es wiederum nicht.

Die zweite Form wird als "wissenschaftliche Notation" bezeichnet. Sie findet dann Verwendung, wenn sehr große Zahlen oder Zahlen, die nahe bei Null liegen, dargestellt werden müssen.

Beispiele:

```
3.0 e5
3.0e-5
```

Der INT – Denoter hinter dem Buchstaben e gibt an, wie viele Stellen der Dezimalpunkt nach rechts (positive Werte) bzw. nach links (negative Werte) zu verschieben ist. Dieser Wert wird Exponent und der Teil vor dem Buchstaben e Mantisse genannt.

TEXT – Denoter:

Sie werden in Anführungszeichen eingeschlossen.

Beispiele:

```
"Das ist ein TEXT-Denoter"
"Jetzt ein TEXT-Denoter ohne ein Zeichen: ein leerer Text"
""
```

Zu beachten ist, daß das Leerzeichen ebenfalls ein Zeichen ist. Soll ein Anführungszeichen in einem TEXT erscheinen (also gerade das Zeichen, welches einen Denoter beendet), so muß es doppelt geschrieben werden.

Beispiele:

```
"Ein TEXT mit dem ""-Zeichen"
"Ein TEXT-Denoter nur mit dem ""-Zeichen:"
"""""
```

Manchmal sollen Zeichen in einem TEXT – Denoter enthalten sein, die auf dem Eingabegerät nicht zur Verfügung stehen. In diesem Fall kann der Code – Wert des Zeichens angegeben werden.

Beispiel:

```
"da"251""
```

ist gleichbedeutend mit "daß". Der Code – Wert eines Zeichens ergibt sich aus der EUMEL – Code – Tabelle (siehe 5.2.4.1), in der jedem Zeichen eine ganze Zahl zugeordnet ist.

BOOL – Denoter:

Es gibt nur zwei BOOL – Denoter:
TRUE für "wahr" und FALSE für "falsch".

2.3.1.2 LET – Konstrukt für Denoter

Neben der Funktion der Abkürzung von Datentypen (siehe 2.6.3) kann das LET – Konstrukt auch für die Namensgebung für Denoter verwandt werden.

Die LET – Vereinbarung sieht folgendermaßen aus:

```
LET Name = Denoter
```

Mehrere Namensgebungen können durch Komma getrennt werden.

Beispiele:

```
LET anzahl = 27;
LET pi     = 3.14159,
    blank  = " ";
```

Der Einsatz von LET – Namen für Denoter hat zwei Vorteile:

- feste Werte im Programm sind leicht zu ändern, da nur an einer Stelle des Programms der Denoter geändert werden muß
  (z.B.: In Vereinbarungen von Reihungen (siehe 2.6.1) können LET – Denoter, im Gegensatz zu Konstanten, als Obergrenze angegeben werden. Dieser Wert kann dann auch an anderen Stellen des Programms, z.B. in Schleifen (siehe 2.4.2.5), benutzt werden. Bei Änderung der Reihungsgröße braucht dann nur an einer Stelle des Programms der Wert geändert zu werden.)
- der Name gibt zusätzliche Information über die Bedeutung des Denoters.

2.3.2 Zugriffsrecht

Von manchen Datenobjekten weiß man, daß sie nur einmal einen Wert erhalten sollen. Sie sollen also nicht verändert werden. Oder man weiß, daß in einem Programmbereich ein Datenobjekt nicht verändert werden soll. Um ein unbeabsichtigtes Verändern zu verhindern, wird in ELAN dem Datenobjekt ein zusätzlicher Schutz mitgegeben: das Zugriffsrecht oder Accessrecht.

In der Deklaration eines Datenobjekts können folgende Accessattribute angegeben werden:

- **VAR** für lesenden und schreibenden (verändernden) Zugriff
- **CONST** für nur lesenden Zugriff.

2.3.3 Deklaration

Damit man Datenobjekte in einem Programm ansprechen kann, gibt man einem Datenobjekt einen Namen (wie z.B. einen Personennamen, unter der sich eine wirkliche Person "verbirgt"). Will man ein Datenobjekt in einem Programm verwenden, so muß man dem Compiler mitteilen, welchen Datentyp und welches Accessrecht das Objekt haben soll. Das dient u.a. dazu, nicht vereinbarte Namen (z.B. verschriebene) vom Compiler entdecken zu lassen. Weiterhin ist aus dem bei der Deklaration angegebenen Datentyp zu entnehmen, wieviel Speicherplatz für das Objekt zur Laufzeit zu reservieren ist.

Eine Deklaration oder Vereinbarung besteht aus der Angabe von

- Datentyp
- Zugriffsrecht (**VAR** oder **CONST**)
- Name des Datenobjekts.

Beispiel:

```
INT VAR mein datenobjekt;
```

Verschiedene Datenobjekte mit gleichem Datentyp und Accessrecht dürfen in einer Deklaration angegeben werden; sie werden durch Kommata getrennt. Mehrere Deklarationen werden – genauso wie Anweisungen – durch das Trennzeichen Semikolon voneinander getrennt.

Beispiele:

```
INT VAR mein wert, dein wert, unser wert;
BOOL VAR listen ende;
TEXT VAR zeile, wort;
```

2.3.4 Initialisierung

Um mit den vereinbarten Datenobjekten arbeiten zu können, muß man ihnen einen Wert geben. Hat ein Datenobjekt noch keinen Wert erhalten, so sagt man, sein Wert sei undefiniert. Das versehentliche Arbeiten mit undefinierten Werten ist eine beliebte Fehlerquelle. Deshalb wird von Programmierern streng darauf geachtet, diese Fehlerkuelle zu vermeiden. Eine Wertgebung an ein Datenobjekt kann (muß aber nicht) bereits bei der Deklaration erfolgen. In ELAN wird dies Initialisierung genannt. Für mit CONST vereinbarte Datenobjekte ist die Initialisierung die einzige Möglichkeit, ihnen einen Wert zu geben. Die Initialisierung von Konstanten ist zwingend vorgeschrieben und wird vom Compiler überprüft.

Die Initialisierung besteht aus der Angabe von

- Datentyp
- Zugriffsrecht (`VAR` oder `CONST`)
- Name des Datenobjekts
- Operator `::` oder `:=`
- Wert, den das Datenobjekt erhalten soll (Denoter, Ausdruck).

Beispiele:

```
INT CONST gewuenschtes gehalt :: 12 000;
TEXT VAR zeile :: "";
REAL CONST pi :: 3.14159, zwei pi := 2.0 * pi;
BOOL VAR bereits sortiert :: TRUE;
```

2.4 Programmeinheiten

Neben Deklarationen (Vereinbarungen) sind Programmeinheiten die Grundbestandteile von ELAN.

Programmeinheiten können sein:

- **elementare Programmeinheiten**
  - Ausdruck
  - Zuweisung
  - Refinementanwendung
  - Prozeduraufruf

- **zusammengesetzte Programmeinheiten**
  - Folge
  - Abfrage
  - Auswahl
  - Wiederholung

- **abstrahierende Programmeinheiten**
  - Refinementbvereinbarung
  - Prozedurvereinbarung
  - Operatorvereinbarung
  - Paketvereinbarung.

2.4.1 Elementare Programmeinheiten

2.4.1.1 Ausdruck

Ausdrücke sind eine Zusammenstellung von Datenobjekten (Denoter, VAR – oder CONST – Objekte) und Operatoren. Jeder korrekte Ausdruck liefert einen Wert. Der Typ des Ausdrucks wird bestimmt durch den Typ des Wertes, den der Ausdruck liefert.

Operatoren

Operatoren werden in ELAN durch ein oder zwei Sonderzeichen oder durch Großbuchstaben als Schlüsselwort dargestellt (siehe 2.4.3.3).

Als Operanden (also die Datenobjekte, auf die ein Operator "wirken" soll) dürfen VAR – und CONST – Datenobjekte, Denoter oder Ausdrücke verwendet werden. Typ der Operanden und des Resultats eines Operators werden in der Operatorvereinbarung festgelegt (siehe 2.4.3.3).

Man unterscheidet zwei Arten von Operatoren:

- **monadische Operatoren**
  Monadischen Operatoren haben nur einen Operanden, der rechts vom Operatorzeichen geschrieben werden muß.

  Beispiel:

  ```
  - a
  NOT x
  ```

  Der '–' – Operator liefert den Wert von a mit umgekehrten Vorzeichen. a muß dabei vom Datentyp INT oder REAL sein.
  Der Operator 'NOT' realisiert die logische Negation. y muß vom Datentyp BOOL sein.

- **dyadische Operatoren**
  Dyadische Operatoren haben zwei Operanden. Das Operatorzeichen steht zwischen den beiden Operanden.

  Beispiele:

```
a + b
a - b
a * b
a DIV b
a ** b
x < y
x <> y
x AND y
x OR y
```

  In den ersten fünf Beispielen werden jeweils die Werte von zwei INT – Objekten a und b addiert (Operatorzeichen: '+'), subtrahiert ('–'), multipliziert ('*'), dividiert (ganzzahlige Division ohne Rest: 'DIV') und potenziert ('**').
  Im sechsten und siebten Beispiel werden zwei BOOL – Werte x und y verglichen und im achten und neunten Beispiel die logische Operation 'und' (Operator 'AND') bzw. 'oder' (Operator 'OR') durchgeführt.

Priorität von Operatoren

Es ist erlaubt, einen Ausdruck wiederum als Operanden zu verwenden. Praktisch bedeutet dies, daß mehrere Operatoren und Datenobjekte zusammen in einem Ausdruck geschrieben werden dürfen.

Beispiele:

```
a + 3 - b * c
- a * b
```

Die Reihenfolge der Auswertung kann man durch Angabe von Klammern steuern.

Beispiel:

```
(a + b) * (a + b)
```

Es wird jeweils erst 'a + b' ausgewertet und dann erst die Multiplikation durchgeführt. In ELAN ist es erlaubt, beliebig viel Klammernpaare zu verwenden (Regel: die innerste Klammer wird zuerst ausgeführt). Es ist sogar zulässig, Klammern zu verwenden, wo keine notwendig sind, denn überflüssige Klammernpaare werden überlesen. Man muß jedoch beachten, daß Ausdrücke, und damit auch z.B. **(a)**, immer Accessrecht CONST haben.

Beispiel:

```
((a - b)) * 3 * ((c + d) * (c - d))
```

Somit können beliebig komplizierte Ausdrücke formuliert werden.

Um solche Ausdrücke einfacher zu behandeln und sie so ähnlich schreiben zu können, wie man es in der Mathematik gewohnt ist, wird in Programmiersprachen die Reihenfolge der Auswertung von Operatoren festgelegt. In ELAN wurden neun Ebenen, Prioritäten genannt, festgelegt:

Priorität	Operatoren
9	alle monadischen Operatoren
8	**
7	*, /, DIV, MOD
6	+, -
5	=, <>, <, <=, >, >=
4	AND
3	OR
2	alle übrigen, nicht in dieser Tabelle aufgeführten dyadischen Operatoren
1	:=

(Die erwähnten Operatoren in der Tabelle werden in der Beschreibung der Standardprozeduren und -Operatoren besprochen).

Operatoren mit der höchsten Priorität werden zuerst ausgeführt, dann die mit der nächst niedrigeren Priorität usw.. Operatoren mit gleicher Priorität werden von links nach rechts ausgeführt. Dadurch ergibt sich die gewohnte Abarbeitungsfolge wie beim Rechnen.

Beispiel:

```
-2 + 3 * 2 ** 3

a) -2
b) 2 ** 3
c) 3 * (2 ** 3)
d) ((-2)) + (3 * (2 ** 3))
```

Wie bereits erwähnt, ist es immer erlaubt, Klammern zu setzen. Ist man sich also über die genaue Abarbeitungsfolge nicht im Klaren, so kann man Klammern verwenden.

2.4.1.2 Zuweisung

Ein spezieller Operator ist die Zuweisung.

Form:

```
Variable := Wert
```

Dieser Operator hat immer die geringste Priorität, wird also immer als letzter einer Anweisung ausgeführt. Die Zuweisung wird verwendet, um einer Variablen einen neuen Wert zu geben. Der Operator ':=' liefert kein Resultat (man sagt auch, er liefert keinen Wert) und verlangt als linken Operanden ein VAR – Datenobjekt, an den der Wert des rechten Operanden zugewiesen werden soll). Der Wert des linken Operanden wird also verändert. Der rechte Operand wird nur gelesen.

Beispiel:

```
a := b;
```

Hier wird der Wert von 'b' der Variablen 'a' zugewiesen. Der vorher vorhandene Wert von 'a' geht dabei verloren. Man sagt auch, der Wert wird überschrieben.

Als rechter Operand des ':=' – Operators darf auch ein Ausdruck stehen.

Beispiel:

```
a := b + c;
```

Hier wird das Resultat von 'b + c' an die Variable 'a' zugewiesen. Man beachte dabei die Prioritäten der Operatoren '+' (Priorität 6) und ':=' (Priorität 1): die Addition wird vor der Zuweisung ausgeführt. Die Auswertung von Zuweisungen mit Ausdrücken muß immer so verlaufen, da die Zuweisung stets die niedrigste Priorität aller Operatoren hat.

Oft kommt es vor, daß ein Objekt auf der linken und rechten Seite des Zuweisungsoperators erscheint, z.B. wenn ein Wert erhöht werden soll.

Beispiel:

```
a := a + 1;
```

Hier wird der "alte", aktuelle Wert von 'a' genommen, um '1' erhöht und dem Objekt 'a' zugewiesen. Man beachte, daß hier in einer Anweisung ein Datenobjekt unterschiedliche Werte zu unterschiedlichen Zeitpunkten haben kann.

2.4.1.3 Refinementanwendung

In ELAN ist es möglich, Namen für Ausdrücke oder eine bzw. mehrere Anweisungen zu vergeben. Das Sprachelement, das diese Namensgebung ermöglicht, heißt Refinement. Die Ausführung eines solchen Namens heißt Refinementanwendung, die Namensgebung heißt Refinementvereinbarung (siehe 2.4.3.1). Die Ausdrücke oder Anweisungen bilden den Refinementrumpf. Ein Refinement kann man in einem Programm unter dem Refinementnamen ansprechen. Man kann sich die Ausführung so vorstellen, als würden der Refinementrumpf immer dort eingesetzt, wo der Name des Refinements als Operation benutzt wird.

2.4.1.4 Prozeduraufruf

Eine Prozedur ist eine Sammlung von Anweisungen und Daten, die zur Lösung einer bestimmten Aufgabe benötigt werden. Eine Prozedur wird in einer Prozedurvereinbarung definiert (siehe 2.4.3.2). Eine solche Prozedur kann man in einem Programm unter einem Namen (eventuell unter Angabe von Parametern) ansprechen. Man spricht dann vom Aufruf einer Prozedur oder einer Prozeduranweisung.

Formen des Prozeduraufrufs:

- **Prozeduren ohne Parameter** werden durch den Prozedurnamen angesprochen.

Beispiel:
```
pause;
```

(Die Prozedur 'pause' wartet bis ein Zeichen eingegeben wird)

- **Prozeduren mit Parameter** werden durch

  Prozedurnamen (aktuelle Parameterliste)

  aufgerufen. Eine Parameterliste ist entweder ein Datenobjekt oder mehrere durch Kommata getrennte Datenobjekte.

Beispiel:
```
pause (10);
```

(Mit der Prozedur 'pause (INT CONST zeitgrenze)' kann für eine Zeitdauer von 'zeitgrenze' in Zehntel – Sekunden gewartet werden. Die Wartezeit wird durch Erreichen der Zeitgrenze oder durch Eingabe eines Zeichens abgebrochen)

Bei den aktuellen Parametern ist folgendes zu beachten:

a) Wird ein VAR – Parameter in der Definition der Prozedur vorgeschrieben, darf kein Ausdruck als aktueller Parameter "übergeben" werden, weil an einen Ausdruck nichts zugewiesen werden kann. Ausdrücke haben – wie bereits erwähnt – das Accessrecht CONST.

Gegenbeispiel:

```
TEXT VAR text1, text2;
text1 := "Dieses Beispiel ";
text2 := "Fehlermeldung";
insert char (text1 + text2, "liefert eine", 17);
```

(Die Prozedur 'insert char (TEXT VAR string, TEXT CONST char, INT CONST pos)' fügt das Zeichen 'char' in den Text 'string' an der Position 'pos' ein)

b) Wird ein CONST – Parameter verlangt, dann darf in diesem Fall ein Ausdruck als aktueller Parameter geschrieben werden. Aber auch ein VAR – Datenobjekt darf angegeben werden. In diesem Fall wird eine Wandlung des Accessrechts (CONSTing) vorgenommen: der aktuelle Parameter erhält sozusagen für die Zeit der Abarbeitung der Prozedur das Accessrecht CONST.

In ELAN sind auch Prozeduren als Parameter erlaubt. Die Prozedur als aktueller Parameter wird in der Parameterliste folgendermaßen angegeben:

```
Resultattyp PROC ( virtuelle Parameterliste ) Procname
```

Die Angabe des Resultattyps entfällt, wenn es sich nicht um eine wertliefernde Prozedur handelt. Die virtuelle Parameterliste inklusive der Klammern entfällt, falls die Prozedur keine Parameter hat. Die virtuelle Parameterliste beschreibt die Parameter der Parameterprozedur. Es werden Datentyp und Zugriffsrecht eines jeden Parameters angegeben, jedoch ohne Namen.

Beispiel:

```
wertetabelle (REAL PROC (REAL CONST) sin,
              untergrenze, obergrenze, schrittweite);
```

(Die Prozedur 'sin' wird an die Prozedur 'wertetabelle' übergeben)

2.4.2 Zusammengesetzte Programmeinheiten

2.4.2.1 Folge

Mehrere in einer bestimmten Reihenfolge auszuführende Anweisungen werden als Folge bezeichnet. In ELAN kann man eine oder mehrere Anweisungen in eine Programmzeile schreiben oder eine Anweisung über mehrere Zeilen. Das setzt jedoch voraus, daß die Anweisungen voneinander getrennt werden. Die Trennung von Anweisungen erfolgt in ELAN durch das Trennsymbol Semikolon. Es bedeutet soviel wie: "führe die nächste Anweisung aus".

Beispiel:

```
put ("main");
put (1);
put (". Programm")
```

(Die Prozedur 'put' gibt den als Parameter angegebenen Wert auf dem Ausgabegerät aus)

2.4.2.2 Abfrage

Mit Abfragen steuert man die bedingte Ausführung von Anweisungen. Abhängig von einer Bedingung wird in zwei verschiedene Programmabschnitte verzweigt.

Der formale Aufbau einer Abfrage sieht folgendermaßen aus:

```
IF Bedingung
   THEN Abschnitt
   ELSE Abschnitt
END IF
```

Der ELSE-Teil darf dabei auch fehlen. Anstelle von END IF darf auch die Abkürzung FI (IF von hinten gelesen) benutzt werden.

In folgenden Beispielen wird der Absolutbetrag von 'a' ausgegeben:

Beispiel 1:

```
INT VAR a;
get (a);
IF a < 0
   THEN a := -a
END IF;
put (a)
```

Die Umkehrung des Vorzeichens von a im THEN – Teil wird nur durchgeführt, wenn der BOOLesche Ausdruck ('a < 0') den Wert TRUE liefert. Liefert er den Wert FALSE, wird die Anweisung, die der bedingten Anweisung folgt (nach END IF), ausgeführt. Das obige Programm kann auch anders geschrieben werden:

```
.......................... Beispiel: ..........................
INT VAR a;
get (a);
IF a < 0
   THEN put (-a)
   ELSE put (a)
END IF
```

Der THEN – Teil wird wiederum ausgeführt, wenn die BOOLesche Bedingung erfüllt ist. Liefert sie dagegen FALSE, wird der ELSE – Teil ausgeführt.

Die bedingte Anweisung ermöglicht es, abhängig von einer Bedingung eine oder mehrere Anweisungen ausführen zu lassen. Dabei können im THEN – bzw. ELSE – Teil wiederum bedingte Anweisungen enthalten sein.

Abfragekette

Bei Abfrageketten kann das ELIF – Konstrukt eingesetzt werden. (ELIF ist eine Zusammenziehung der Worte ELSE und IF).

Anstatt

Beispiel:

```
IF bedingung1
   THEN aktion1
   ELSE IF bedingung2
           THEN aktion2
           ELSE aktion3
        END IF
   END IF;
```

kann man besser

Beispiel:

```
IF bedingung1
   THEN aktion1
ELIF bedingung2
   THEN aktion2
   ELSE aktion3
END IF;
```

schreiben.

2.4.2.3 Auswahl

Die Auswahl wird benutzt, wenn alternative Anwendungen in Abhängigkeit von Werten eines Datenobjekts ausgeführt werden sollen.

Der formale Aufbau der Auswahl sieht folgendermaßen aus:

```
SELECT INT-Ausdruck OF
   CASE 1. Liste von INT-Denotern : Abschnitt
   CASE 2. Liste von INT-Denotern : Abschnitt
                  .
                  .
                  .
   CASE n. Liste von INT-Denotern : Abschnitt
   OTHERWISE Abschnitt
END SELECT
```

Eine Liste von INT – Denotern besteht aus einem oder mehreren durch Kommata getrennten INT – Denotern. Der OTHERWISE – Teil darf auch fehlen. Man sollte ihn jedoch verwenden, um Fehlerfälle abzufangen.

```
SELECT monat OF
   CASE 2: IF schaltjahr
              THEN tage := 29
              ELSE tage := 28
           END IF
   CASE 4, 6, 9, 11: tage := 30
   CASE 1, 3, 5, 7, 8, 10 ,12: tage := 31
   OTHERWISE kein monat
END SELECT;
```

(In diesem Programmausschnitt werden die Tage eines Monats bestimmt)

2.4.2.4 Wertliefernde Abfrage und wertliefernde Auswahl

Soll eine Abfrage oder eine Auswahl einen Wert liefern, dann darf der ELSE– bzw. der OTHERWISE–Teil nicht fehlen und alle Zweige müssen einen Wert liefern.

Beispiel:

```
SELECT monat OF
   CASE 2: IF schaltjahr
              THEN 29
              ELSE 28
           END IF
   CASE 4, 6, 9, 11: 30
   CASE 1, 3, 5, 7, 8, 10 ,12: 31
   OTHERWISE kein monat; 0
END SELECT;
```

2.4.2.5 Wiederholung

Die Wiederholung dient zur mehrfachen Ausführung von Anweisungen, meist in Abhängigkeit von einer Bedingung. Darum wird die Wiederholungsanweisung oft auch Schleife genannt und die in ihr enthaltenen Anweisungen Schleifenrumpf.

Es gibt verschiedene Schleifentypen:

- Endlosschleife
- abweisende Schleife
- nicht abweisende Schleife
- Zählschleife.

Endlosschleife

Bei der Endlosschleife wird nicht spezifiziert, wann die Schleife beendet werden soll.

Form:

```
REPEAT
   Abschnitt
END REPEAT
```

Anstelle von **REPEAT** darf die Abkürzung **REP** und anstelle von **END REPEAT** das Schlüsselwort **PER** (REP von hinten gelesen) benutzt werden.

Beispiel:

```
break;
REPEAT
   fixpoint;
   pause (18000)
END REPEAT
```

Wird dieses Programm in einer Task im SYSUR – Zweig ausgeführt, so führt diese Task Fixpunkte im Abstand von 30 Minuten durch.

Abweisende Schleife

Bei der abweisenden Schleife wird die Abbruchbedingung an den Anfang der Schleife geschrieben.

Form:

```
WHILE Bedingung REPEAT
   Abschnitt
END REPEAT
```

Bei jedem erneuten Durchlauf der Schleife wird überprüft, ob der BOOLesche Ausdruck den Wert TRUE liefert. Ist das nicht der Fall, wird die Bearbeitung mit der Anweisung fortgesetzt, die auf das Schleifenende folgt. Die Schleife wird abweisende Schleife genannt, weil der Schleifenrumpf nicht ausgeführt wird, wenn die Bedingung vor Eintritt in die Schleife bereits FALSE liefert.

Nicht abweisende Schleife

Anders verhält es sich bei der nicht abweisenden Schleife. Bei der nicht abweisenden Schleife wird die Abbruchbedingung an das Ende der Schleife geschrieben.

Form:

```
REPEAT
   Abschnitt
UNTIL Bedingung END REPEAT
```

Hier wird der Schleifenrumpf auf jeden Fall einmal bearbeitet. Am Ende des Rumpfes wird die BOOLesche Bedingung abgefragt. Liefert sie den Wert FALSE, wird die Schleife erneut abgearbeitet. Liefert die Bedingung den Wert TRUE, wird die Schleife abgebrochen und mit der ersten Anweisung hinter der Schleife in der Bearbeitung fortgefahren.

Bei den beiden letztgenannten Arten der Wiederholungsanweisung ist es wichtig, daß Elemente der BOOLeschen Bedingung in der Schleife verändert werden, damit das Programm terminieren kann, d.h. die Schleife abgebrochen wird.

```
Beispiel
TEXT VAR wort, satz := "";
REPEAT
  get (wort);
  satz CAT wort;
  satz CAT " "
UNTIL wort = "." PER;
```

Dises Programm liest solange Wörter ein und verbindet diese zu einem Satz, bis ein Punkt eingegeben wurde.

Zählschleife

Zählschleifen werden eingesetzt, wenn die genaue Anzahl der Schleifendurchläufe bekannt ist.

Form:

```
FOR Laufvariable FROM Anfangswert UPTO Endwert REPEAT
   Abschnitt
END REPEAT
```

Bei Zählschleifen wird eine Laufvariable verwendet, die die INT – Werte von 'Anfangswert' bis 'Endwert' in Schritten von 1 durchläuft. 'Anfangswert' und 'Endwert' können beliebige INT – Ausdrücke sein. Diese Schleife zählt "aufwärts". Wird anstatt UPTO das Schlüsselwort DOWNTO verwendet, wird mit Schritten von 1 "abwärts" gezählt.

Form:

```
FOR Laufvariable FROM Endwert DOWNTO Anfangswert REPEAT
   Abschnitt
END REPEAT
```

Die Laufvariable darf in der Schleife nicht verändert werden. Nach dem normalen Schleifenende ist der Wert der Laufvariablen nicht definiert.

Beispiel:

```
INT VAR summe := 0, i;
FOR i FROM 1 UPTO 100 REPEAT
   summe INCR i
END REPEAT
```

Dieses Programm berechnet die Summe der natürlichen Zahlen von 1 bis 100.

Die verschiedenen Schleifenarten können kombiniert werden:

```
FOR Laufvariable FROM Anfangswert UPTO Endwert
WHILE Bedingung REPEAT
   Abschnitt
END REPEAT
```

```
FOR Laufvariable FROM Anfangswert UPTO Endwert REPEAT
   Abschnitt
UNTIL Bedingung END REPEAT
```

```
WHILE Bedingung REPEAT
   Abschnitt
UNTIL Bedingung END REPEAT
```

2.4.3 Abstrahierende Programmeinheiten

2.4.3.1 Refinementvereinbarung

In ELAN ist es möglich, Namen für Ausdrücke oder eine bzw. mehrere Anweisungen zu vergeben. Das Sprachelement, das diese Namensgebung ermöglicht, heißt Refinement. Die Ausführung eines solchen Namens heißt Refinementanwendung (siehe 2.4.1.3), die Namensgebung heißt Refinementvereinbarung. Die Ausdrücke oder Anweisungen bilden den Refinementrumpf.

Werden in einem Programm Refinements benutzt, dann wird der Programmteil bis zum ersten Refinement durch einen Punkt abgeschlossen. Die Refinementvereinbarung sieht folgendermaßen aus:

```
Name :
   Abschnitt .
```

Beispiel:

```
INT VAR a, b, x;
einlesen von a und b;
vertauschen von a und b;
vertauschte werte ausgeben.

einlesen von a und b:
   get (a);
   get (b).

vertauschen von a und b:
   x := a;
   a := b;
   b := x.

vertauschte werte ausgeben:
   put (a);
   put (b).
```

Für den Namen 'einlesen von a und b' werden die Anweisungen 'get (a); get (b)' vom ELAN – Compiler eingesetzt. Man kann also die ersten vier Zeilen des Programms als eigentliches Programm ansehen, wobei die Namen durch die betreffenden Anweisungen ersetzt werden. Ein Refinement hat also keinen eigenen Datenbereich, d.h. Vereinbarungen, die in Refinements gemacht werden, gelten auch außerhalb des Refinements.

Vorteile der Refinementanwendung

Durch die sinnvolle Verwendung von Refinements wird ein Programm im Programm und nicht in einer separaten Beschreibung dokumentiert. Weiterhin kann ein Programm "von oben nach unten" ("top down") entwickelt werden: Das obige – zugegeben einfache – Beispielprogramm wurde in drei Teile zerlegt und diese durch Namen beschrieben. Bei der Beschreibung von Aktionen durch Namen wird gesagt was gemacht werden soll. Es wird noch nicht beschrieben wie, denn auf dieser Stufe der Programmentwicklung braucht man sich um die Realisierung der Refinements (noch) keine Sorgen zu machen. Das erfolgt erst, wenn das Refinement programmiert werden muß. Dabei können wiederum Refinements verwendet werden usw., bis man auf eine Ebene "heruntergestiegen" ist, bei der eine (jetzt: Teil –) Problemlösung sehr einfach ist und man sie direkt hinschreiben kann. Man beschäftigt sich also an jedem Punkt der Problemlösung nur mit einem Teilaspekt des gesamten Problems. Zudem sieht man – wenn die Refinements einigermaßen vernünftig verwendet werden – dem Programm an, wie die Problemlösung entstanden ist.

Die Verwendung von Refinements hat also eine Anzahl von Vorteilen.
Refinements ermöglichen:

- "top down" – Programmierung
- Strukturierung von Programmen und damit effiziente Fehlersuche und gute Wartbarkeit
- Dokumentation im Programmtext.

Wertliefernde Refinements

Refinements können auch dort verwendet werden, wo ein Wert erwartet wird, z.B. in einem Ausdruck oder einer 'put' – Anweisung. In diesem Fall muß die letzte Anweisung des Refinements einen Wert liefert.

Beispiel:

```
INT VAR a :: 1, b :: 2, c :: 3;
put (resultat).

resultat:
   (a * b + c) ** 3.
```

Man kann auch ein wertlieferndes Refinement mit mehreren Anweisungen schreiben.

Allgemeine Regel:
Die letzte Anweisung eines Refinements bestimmt, ob es einen Wert liefert – und wenn ja, von welchen Datentyp.

2.4.3.2 Prozedurvereinbarung

Eine Prozedur ist eine Sammlung von Anweisungen und Daten, die zur Lösung einer bestimmten Aufgabe benötigt werden.

Der formale Aufbau einer Prozedur sieht folgendermaßen aus:

```
PROC Prozedurname :
   Prozedurrumpf
END PROC Prozedurname
```

Der Prozedurrumpf kann Deklarationen, Anweisungen und Refinements enthalten.

Beispiel:

```
PROC loesche bildschirm ab aktueller cursorposition:
   out (""4"")
END PROC loesche bildschirm ab aktueller cursorposition
```

Verwendung von Prozeduren

Prozeduren werden verwendet, wenn

- Anweisungen und Datenobjekte unter einem Namen zusammengefaßt werden sollen ("Abstraktion")
- gleiche Anweisungen von mehreren Stellen eines Programms verwandt werden sollen (Codereduktion), u.U. mit verschieden Datenobjekten (Parameter)
- Datenobjekte nur innerhalb eines Programmteils benötigt werden und diese nicht von dem gesamten Programm angesprochen werden sollen.

In den folgenden Programmfragmenten werden zwei Werte vertauscht. In der ersten Lösung wird ein Refinement, in der zweiten eine Prozedur verwandt.

Beispiel:

```
IF a > b
  THEN vertausche a und b
END IF;
put (a);
put (b);
vertausche a und b.

vertausche a und b:
   INT CONST x :: a;
   a := b;
   b := x.
```

Beispiel:

```
PROC vertausche a und b:
   INT CONST x :: a;
   a := b;
   b := x
END PROC vertausche a und b;

IF a > b
  THEN vertausche a und b
END IF;
put (a);
put (b);
vertausche a und b;
```

Beim ersten Hinsehen leisten beide Programme das Gleiche. Es gibt jedoch drei wichtige Unterschiede:

1) Das Refinement 'vertausche a und b' wird zweimal (vom ELAN – Compiler) eingesetzt, d.h. der Code ist zweimal vorhanden. Die Prozedur dagegen ist vom Code nur einmal vorhanden, wird aber zweimal – durch das Aufführen des Prozedurnamens – aufgerufen.

2) Die Variable 'x' ist in der ersten Programmversion während des gesamten Ablauf des Programms vorhanden, d.h. ihr Speicherplatz ist während dieser Zeit belegt. Solche Datenobjekte nennt man statische Datenobjekte oder auch (aus Gründen, die erst etwas später offensichtlich werden) Paket – Objekte. Das Datenobjekt 'x' der rechten Version dagegen ist nur während der Bearbeitung der Prozedur vorhanden, sein Speicherplatz wird danach freigegeben. Solche Datenobjekte, die nur kurzfristig Speicher belegen, werden dynamische Datenobjekte genannt.

   Prozeduren sind also ein Mittel, um die Speicherbelegung zu beeinflussen.

3) Da Refinements keinen eigenen Datenbereich haben, kann die Variable 'x' in der ersten Programmversion – obwohl sie in einem Refinement deklariert wurde – von jeder Stelle des Programms angesprochen werden. Solche Datenobjekte werden globale Datenobjekte genannt. Das Datenobjekt 'x' der Prozedur dagegen kann nur innerhalb der Prozedur angesprochen werden, es ist also ein lokales Datenobjekt der Prozedur. Innerhalb der Prozedur dürfen globale Datenobjekte (also Objekte, die außerhalb von Prozeduren deklariert wurden) auch angesprochen werden.

   Eine Prozedur in ELAN bildet im Gegensatz zu Refinements einen eigenen Gültigkeitsbereich hinsichtlich Datenobjekten und Refinements, die innerhalb der Prozedur deklariert werden. Prozeduren sind somit ein Mittel, um die in ihr deklarierten Datenobjekte hinsichtlich der Ansprechbarkeit nach Außen "abzuschotten".

Prozeduren mit Parametern

Prozeduren mit Parametern erlauben es, gleiche Anweisungen mit unterschiedlichen Datenobjekten auszuführen.

Form:

```
PROC Prozedurname ( formale Parameterliste ) :
   Prozedurrumpf
END PROC Prozedurnamen
```

Die Parameterliste besteht aus einem oder mehreren durch Kommata getrennten Parametern. Ein Parameter wird mit Datentyp, Accessrecht und Namen angegeben.
Ähnlich wie bei der Datendeklaration braucht man für aufeinanderfolgende Parameter mit gleichem Datentyp und gleichem Accessrecht die Attribute nur einmal anzugeben.
Parameter mit Accessrecht **CONST** sind Eingabeparameter, Parameter mit Accessrecht **VAR** realisieren Ein-/Ausgabeparameter.

Beispiel:

```
PROC vertausche (INT VAR a, b):
   INT VAR x :: a;
   a := b;
   b := x
END PROC vertausche;

INT VAR eins :: 1,
        zwei :: 2,
        drei :: 3;
vertausche (eins, zwei);*
vertausche (zwei, drei);
vertausche (eins, zwei);
put (eins); put (zwei); put (drei)
```

Die Datenobjekte 'a' und 'b' der Prozedur 'vertausche' werden formale Parameter genannt. Sie stehen als Platzhalter für die bei einem Prozeduraufruf einzusetzenden aktuellen Parameter (in obigen Beispiel die Datenobjekte 'eins', 'zwei' und 'drei').

Prozeduren als Parameter

Es ist auch möglich, Prozeduren als Parameter zu definieren.

Eine Prozedur als Parameter wird folgendermaßen in der Parameterliste spezifiziert:

`Resultattyp PROC ( virtuelle Parameterliste ) Prozedurname`

Die Angabe des Resultattyps entfällt, wenn es sich nicht um eine wertliefernde Prozedur handelt. Die virtuelle Parameterliste inklusive der Klammern entfällt, falls die Prozedur keine Parameter hat. Die virtuelle Parameterliste beschreibt die Parameter der Parameterprozedur. Es werden Datentyp und Zugriffsrecht eines jeden Parameters angegeben, jedoch ohne Namen.

Beispiel:

```
PROC wertetabelle (REAL PROC (REAL CONST) funktion,
                   REAL CONST untergrenze, obergrenze,
                        schrittweite):

REAL VAR wert;
putline ("W E R T E T A B E L L E");
putline ("-----------------------");
wert := untergrenze;
REPEAT
   put (text (wert, 10, 5));
   put (text (funktion (wert), 10, 5));
   line;
   wert INCR schrittweite
UNTIL wert > obergrenze PER

END PROC wertetabelle;

(* Prozeduraufruf: *)
wertetabelle (REAL PROC (REAL CONST) sin, 0.0, pi, 0.2)
```

Wertliefernde Prozeduren

Eine wertliefernde Prozedur sieht folgendermaßen aus:

```
Resultattyp PROC Prozedurname ( formale Parameterliste ) :
   wertliefernder Prozedurrumpf
END PROC Prozedurnamen
```

Die Parameterliste inklusive Klammerung kann fehlen. Der Prozedurrumpf muß einen Wert mit dem in Resultattyp angegeben Datentyp liefern.

Beispiel:

```
INT PROC max (INT CONST a, b):
   IF a > b
     THEN a
     ELSE b
   END IF
END PROC max;

put (max (3, 4))
```

(In diesem Beispiel wird das Maximum von 'a' und 'b' ermittelt und ausgegeben)

2.4.3.3 Operatorvereinbarung

Operatoren können in ELAN ähnlich wie Prozeduren definiert werden. Operatoren müssen einen und können maximal zwei Operatoren besitzen (monadische und dyadische Operatoren).

Form:

```
Resultattyp OP Opname ( ein oder zwei Parameter ) :
   Operatorrumpf
END OP Opname
```

Der Resultattyp wird nur bei wertliefernden Operatoren angegeben.

Als Operatornamen sind erlaubt:

- ein Sonderzeichen, sofern es nicht als Trennzeichen benutzt wird:
  ! $ % & ' * + - / < = > ? @ ^ ` ~
- eine Kombination von zwei Sonderzeichen. Diese Kombination muß jedoch bereits in ELAN existieren:
  := <= >= <> **
- ein Schlüsselwort (siehe 2.2.1).

Vereinbarung eines monadischen Operators

Beispiel:

```
INT OP SIGN (REAL CONST argument):
   IF    argument < 0.0  THEN  -1
   ELIF  argument = 0.0  THEN  0
                         ELSE  1
   FI
END OP SIGN
```

(Der Operator 'SIGN' liefert abhängig vom Vorzeichen des übergebenen Wertes den INT – Wert – 1, 0 oder 1)

Vereinbarung eines dyadischen Operators

Beispiel:

```
TEXT OP * (INT CONST anzahl, TEXT CONST t):
   INT  VAR zaehler :: anzahl;
   TEXT VAR ergebnis :: "";
   WHILE zaehler > 0 REP
      ergebnis := ergebnis + t;
      zaehler := zaehler - 1
   END REP;
   ergebnis
END OP *;
```

(Der Operator '*' verkettet 'anzahl' – mal den Text 't')

2.4.3.4 Paketvereinbarung

Pakete sind in ELAN eine Zusammenfassung von Datenobjekten, Prozeduren, Operatoren und Datentypen. Diese bilden den Paketrumpf. Elemente eines Pakets (Prozeduren, Operatoren, Datentypen) können außerhalb des Pakets nur angesprochen werden, wenn sie in der Schnittstelle des Pakets, die auch "interface" genannt wird, aufgeführt werden. Mit anderen Worten: es können alle Elemente eines Pakets von außen nicht angesprochen werden, sofern sie nicht über die Schnittstelle "nach außen gereicht" werden. Pakete können separat übersetzt werden, so daß der "Zusammenbau" eines umfangreichen Programms aus mehreren Paketen möglich ist.

Der formale Aufbau eines Pakets sieht folgendermaßen aus:

```
PACKET Paketname DEFINES Schnittstelle :
   Paketrumpf
END PACKET Paketname
```

In der Schnittstelle werden Prozeduren und Operatoren nur mit ihrem Namen, durch Kommata getrennt, angegeben. Weiterhin können Datentypen und mit CONST vereinbarte Datenobjekte in der Schnittstelle aufgeführt werden, aber keine VAR – Datenobjekte, weil diese sonst über Paket – Grenzen hinweg verändert werden könnten.

Im Gegensatz zu einer Prozedur kann ein PACKET nicht aufgerufen werden (nur die Elemente der Schnittstelle können benutzt werden).

Pakete werden zu folgenden Zwecken eingesetzt:

- Spracherweiterung
- Schutz vor fehlerhaftem Zugriff auf Datenobjekte
- Realisierung von abstrakten Datentypen.

Spracherweiterung

Beispiel:

```
PACKET fuer eine prozedur DEFINES swap:

PROC swap (INT VAR a, b):
   INT CONST x :: a;
   b := a;
   a := x
END PROC swap

END PACKET fuer eine prozedur
```

Dies ist ein Paket, das eine Tausch – Prozedur für INT – Datenobjekte bereitstellt. Das PACKET kann übersetzt und dem ELAN – Compiler bekannt gemacht werden (EUMEL: "insertieren"). Ist das geschehen, kann man 'swap' wie alle anderen Prozeduren (z.B. 'put', 'get') in einem Programm verwenden. Tatsächlich werden die meisten Prozeduren und Operatoren (aber auch einige Datentypen), die in ELAN zur Verfügung stehen, nicht durch den ELAN – Compiler realisiert, sondern durch solche PACKETs. Um solche Objekte einigermaßen zu standardisieren, wurde in der ELAN – Sprachbeschreibung festgelegt, welche Datentypen, Prozeduren und Operatoren in jedem ELAN – System vorhanden sein müssen. Solche Pakete werden Standard – Pakete genannt. Jeder Installation – aber auch jedem Benutzer – steht es jedoch frei, zu den Standard – Paketen zusätzliche Pakete dem Compiler bekanntzugeben, und damit den ELAN – Sprachumfang zu erweitern.

Schutz vor fehlerhaftem Zugriff auf Datenobjekte

Beispiel:

```
PACKET stack handling DEFINES push, pop, init stack:

LET max = 1000;
ROW max INT VAR stack; (* siehe Kapitel Reihung, 2.6.1. *)
INT VAR stack pointer;

PROC init stack:
   stack pointer := 0
END PROC init stack;

PROC push (INT CONST dazu wert):
   stack pointer INCR 1;
   IF stack pointer > max
     THEN errorstop ("stack overflow")
     ELSE stack [stack pointer] := dazu wert
   END IF
END PROC push;

PROC pop (INT VAR von wert):
   IF stack pointer = 0
     THEN errorstop ("stack empty")
     ELSE von wert := stack [stack pointer];
          stack pointer DECR 1
   END IF
END PROC pop

END PACKET stack handling;
```

Dieses Packet realisiert einen Stack. Den Stack kann man über die Prozeduren 'init stack', 'push' und 'pop' benutzen.

Beispiel:

```
init stack;
werte einlesen und pushen;
werte poppen und ausgeben.

werte einlesen und pushen:
   INT VAR anzahl := 0, wert;
   REP
      get (wert);
      push (wert);
      anzahl INCR 1
   UNTIL ende kriterium END REP.

werte poppen und ausgeben:
   INT VAR i;
   FOR i FROM 1 UPTO anzahl REP
      pop (wert);
      put (wert)
   END REP.
```

Die Datenobjekte 'stack' und 'stack pointer' haben nur Gültigkeit innerhalb des PACKETs 'stack handling'.

Anweisungen wie z.B.

Beispiel:

```
put (stack [3]);
stack [27] := 5
```

außerhalb des PACKETs 'stack handling' sind also verboten und werden vom ELAN – Compiler entdeckt.

Ein PACKET bietet also auch einen gewissen Schutz vor fehlerhafter Verwendung von Programmen und Datenobjekten. Wichtig ist weiterhin, daß die Realisierung des Stacks ohne weiteres geändert werden kann, ohne daß Benutzerprogramme im 'main packet' geändert werden müssen, sofern die Schnittstelle nicht verändert wird. Beispielsweise kann man sich entschließen, den Stack nicht durch eine Reihung, sondern durch eine Struktur zu realisieren. Davon bleibt ein Benutzerprogramm unberührt.

Realisierung von abstrakten Datentypen

Der Vollständigkeit halber wird folgendes Beispiel hier gezeigt. Wie neue Datentypen definiert werden, wird in Kapitel 2.7.1. erklärt.

Beispiel:

```
PACKET widerstaende DEFINES WIDERSTAND, REIHE, PARALLEL,
                            :=, get, put:

TYPE WIDERSTAND = INT;

OP := (WIDERSTAND VAR l, WIDERSTAND CONST r):
   CONCR (l) := CONCR (r)
END OP :=;

PROC get (WIDERSTAND VAR w):
   INT VAR i;
   get (i);
   w := WIDERSTAND : (i)
END PROC get;

PROC put (WIDERSTAND CONST w):
   put (CONCR (w))
END PROC put;

WIDERSTAND OP REIHE (WIDERSTAND CONST l, r):
   WIDERSTAND : ( CONCR (l) + CONCR (r))
END OP REIHE;

WIDERSTAND OP PARALLEL (WIDERSTAND CONST l, r):
  WIDERSTAND :
    ((CONCR (l) * CONCR (r)) DIV (CONCR (l) + CONCR (r)))
END OP PARALLEL

END PACKET widerstaende
```

Dieses Programm realisiert den Datentyp WIDERSTAND und mit den Operationen eine Fachsprache.

2.4.4 Terminatoren für Refinements, Prozeduren und Operatoren

Das LEAVE – Konstrukt wird verwendet, um eine benannte Anweisung (Refinement, Prozedur oder Operator) vorzeitig zu verlassen. Es ist auch möglich, geschachtelte Refinements zu verlassen.

Form:

LEAVE Name

Durch eine (optionale) WITH – Angabe kann auch eine wertliefernde benannte Anweisung verlassen werden.

Form:

LEAVE Name **WITH** Ausdruck

Beispiel:

```
INT OP ** (INT CONST basis, exp):
    IF exp = 0
       THEN LEAVE ** WITH 1
    ELIF exp < 0
       THEN LEAVE ** WITH 0
    FI;

    INT VAR  zaehler, ergebnis;
    ergebnis := basis;
    FOR zaehler FROM 2 UPTO exp REP
       ergebnis := ergebnis * basis
    PER;
    ergebnis
END OP **
```

(Diese Operation realisiert die Exponentiation für INT – Werte)

2.4.5 Generizität von Prozeduren und Operatoren

In ELAN ist es möglich, unterschiedlichen Prozeduren bzw. Operatoren gleiche Namen zu geben. Solche Prozeduren (Operatoren) werden generische Prozeduren (Operatoren) genannt. Die Identifizierung erfolgt durch Anzahl, Reihenfolge und Datentyp der Parameter (Operanden).

Deshalb werden Prozeduren und Operatoren unter Angabe des Prozedur- bzw. des Operatorkopfes dokumentiert.

Beispiele:

```
INT  OP MOD (INT CONST  l, r)
REAL OP MOD (REAL CONST l, r)
```

Der MOD-Operator liefert den Rest einer Division. Er ist sowohl für INT- wie auch für REAL-Datenobjekte definiert.

```
PROC put (INT CONST wert)
PROC put (REAL CONST wert)
PROC put (TEXT CONST wert)
```

Die put-Prozedur ist für INT-, REAL- und TEXT-Datenobjekte definiert.

Priorität von generischen Operatoren

Bei der Neudefinition von Operatoren kann man bereits benutzte Sonderzeichen oder Schlüsselwörter benutzen. In diesem Fall bekommt der neudefinierte Operator die gleiche Priorität wie der bereits vorhandene Operator.

2.4.6 Rekursive Prozeduren und Operatoren

Alle Prozeduren und Operatoren dürfen in ELAN rekursiv sein.

Beispiel:

```
INT PROC fakultaet (INT CONST n):
   IF n > 0
     THEN fakultaet (n-1) * n
     ELSE 1
   END IF
END PROC fakultaet
```

Die Fakultätsfunktion ist kein gutes Beispiel für eine Rekursion, denn das Programm kann leicht in eine iterative Version umgewandelt werden:

Beispiel:

```
INT PROC fakultaet (INT CONST n):
   INT VAR prod := 1, i;
   FOR i FROM 2 UPTO n REP
     prod := prod * i
   END REP;
   prod
END PROC fakultaet
```

Die Umwandlung von einem rekursiven Programm in ein iteratives ist übrigens immer möglich, jedoch oft nicht so einfach, wie in dem Beispiel der Ackermann – Funktion:

Beispiel:

```
INT PROC acker (INT CONST m, n):
   IF m = 0
     THEN n + 1
   ELIF n = 0
     THEN acker (m-1, 0)
     ELSE acker (m - 1, acker (m, n - 1))
   ENDIF
END PROC acker
```

Das eigentliche Einsatzgebiet von rekursiven Algorithmen liegt aber bei den 'backtrack' – Verfahren. Diese werden eingesetzt, wenn eine exakte algorithmische Lösung nicht bekannt ist oder nicht gefunden werden kann und man verschiedene Versuche machen muß, um zu einem Ziel (oder Lösung) zu gelangen.

2.5 Programmstruktur

Ein ELAN – Programm kann aus mehreren Moduln (Bausteinen) aufgebaut sein, die in ELAN PACKETs genannt werden. Das letzte PACKET wird "main packet" genannt, weil in diesem das eigentliche Benutzerprogramm (Hauptprogramm) enthalten ist.
Dies soll eine Empfehlung sein, in welcher Reihenfolge die Elemente eines PACKETs geschrieben werden sollen:

Ein "main packet" kann aus folgenden Elementen bestehen:

a) Deklarationen und Anweisungen. Diese müssen nicht in einer bestimmten Reihenfolge im Programm erscheinen, sondern es ist möglich, erst in dem Augenblick zu deklarieren, wenn z.B. eine neue Variable benötigt wird. Es ist jedoch gute Programmierpraxis, die meisten Deklarationen an den Anfang eines Programms oder Programmteils (Refinement, Prozedur) zu plazieren.

   <Deklarationen> ;
   <Anweisungen>

Beispiel:

```
INT VAR erste zahl, zweite zahl;

page;
put ("erste Zahl = "); get (erste zahl);
put ("zweite Zahl ="); get (zweite zahl)
```

b) Deklarationen, Refinements und Anweisungen. In diesem Fall ist es notwendig, die Refinements hintereinander zu plazieren. Refinement – Aufrufe und/oder Anweisungen sollten textuell vorher erscheinen.

   <Deklarationen> ;
   <Refinement – Aufrufe und/oder Anweisungen> .
   <Refinements>

   Innerhalb der Refinements sind Anweisungen und/oder Deklarationen möglich.

Beispiel:

```
INT VAR erste zahl, zweite zahl;

loesche bildschirm;
lies zwei zahlen ein.

loesche bildschirm:
  page.

lies zwei zahlen ein:
  put ("erste Zahl = "); get (erste zahl);
  put ("zweite Zahl ="); get (zweite zahl).
```

c) Deklarationen, Prozeduren und Anweisungen. Werden Prozeduren vereinbart, sollte man sie nach den Deklarationen plazieren. Danach sollten die Anweisungen folgen:

<Deklarationen> ;
<Prozeduren> ;
<Anweisungen>

Mehrere Prozeduren werden durch ";" voneinander getrennt. In diesem Fall sind die Datenobjekte aus den Deklarationen außerhalb von Prozeduren statisch, d.h. während der gesamten Laufzeit des Programm vorhanden. Solche Datenobjekte werden auch PACKET – Daten genannt. Im Gegensatz dazu sind die Datenobjekte aus Deklarationen in Prozeduren dynamische Datenobjekte, die nur während der Bearbeitungszeit der Prozedur existieren. Innerhalb einer Prozedur dürfen wiederum Refinements verwendet werden. Ein Prozedur – Rumpf hat also den formalen Aufbau wie unter a) oder b) geschildert.

Die Refinements und Datenobjekte, die innerhalb einer Prozedur deklariert wurden, sind lokal zu dieser Prozedur, d.h. können von außerhalb nicht angesprochen werden.

Beispiel:

```
INT VAR erste zahl, zweite zahl;

PROC vertausche (INT VAR a, b):
  INT VAR x;

  x := a;
  a := b;
  b := x
END PROC vertausche;

put ("erste Zahl = "); get (erste zahl);
put ("zweite Zahl ="); get (zweite zahl);
IF erste zahl > zweite zahl
   THEN vertausche (erste zahl, zweite zahl)
FI
```

d) Deklarationen, Prozeduren, Anweisungen und PACKET – Refinements. Zusätzlich zu der Möglichkeit c) ist es erlaubt, neben den Anweisungen außerhalb einer Prozedur auch Refinements zu verwenden:

<Deklarationen> ;
<Prozeduren> ;
<Anweisungen> .
<Refinements>

Diese Refinements können nun in Anweisungen außerhalb der Prozeduren benutzt werden oder auch durch die Prozeduren (im letzteren Fall spricht man analog zu globalen PACKET – Daten auch von PACKET – Refinements oder globalen Refinements). In PACKET – Refinements dürfen natürlich keine Datenobjekte verwandt werden, die lokal zu einer Prozedur sind.

Beispiel:

```
INT VAR erste zahl, zweite zahl;

PROC vertausche (INT VAR a, b):
  INT VAR x;

  x := a;
  a := b;
  b := x
END PROC vertausche;

loesche bildschirm;
lies zwei zahlen ein;
ordne die zahlen.

loesche bildschirm:
  page.

lies zwei zahlen ein:
  put ("erste Zahl = "); get (erste zahl);
  put ("zweite Zahl ="); get (zweite zahl).

ordne die zahlen:
  IF erste zahl > zweite zahl
     THEN vertausche (erste zahl, zweite zahl)
  FI
```

2.6 Zusammengesetzte Datentypen

In ELAN gibt es die Möglichkeit, gleichartige oder ungleichartige Datenobjekte zu einem Objekt zusammenzufassen.

2.6.1 Reihung

Die Zusammenfassung gleichartiger Datenobjekte, wird in ELAN eine Reihung (ROW) genannt. Die einzelnen Objekte einer Reihung werden Elemente genannt.

Eine Reihung wird folgendermaßen deklariert:

- Schlüsselwort ROW
- Anzahl der zusammengefaßten Elemente
  (INT – Denoter oder durch LET definierter Name)
- Datentyp der Elemente
- Zugriffsrecht (VAR oder CONST)
- Name der Reihung.

Beispiel:

```
ROW 10 INT VAR feld
```

Im obigen Beispiel wird eine Reihung von 10 INT – Elementen deklariert. ROW 10 INT ist ein (neuer, von den elementaren unterschiedlicher) Datentyp, für den keine Operationen definiert sind, außer der Zuweisung. Das Accessrecht (VAR im obigen Beispiel) und der Name ('feld') gilt – wie bei den elementaren Datentypen – für diesen neuen Datentyp, also für alle 10 Elemente.

Warum gibt es keine Operationen außer der Zuweisung? Das wird sehr schnell einsichtig, wenn man bedenkt, daß es ja sehr viele Datentypen (zusätzlich zu den elementaren) gibt, weil Reihungen von jedem Datentyp gebildet werden können:

```
ROW 1 INT              ROW 1 REAL
ROW 2 INT              ROW 2 REAL
 :                      :
ROW maxint INT         ROW maxint REAL

ROW 1 TEXT             ROW 1 BOOL
ROW 2 TEXT             ROW 2 BOOL
 :                      :
ROW maxint TEXT        ROW maxint BOOL
```

Für die elementaren INT-, REAL-, BOOL- und TEXT-Datentypen sind unterschiedliche Operationen definiert. Man müßte nun für jeden dieser zusammengesetzten Datentypen z.B. auch 'get'- und 'put'-Prozeduren schreiben, was allein vom Schreibaufwand sehr aufwendig wäre. Das ist der Grund dafür, daß es keine vorgegebene Operationen auf zusammengesetzte Datentypen gibt.

Zugegebenermaßen könnte man mit solchen Datentypen, die nur über eine Operation verfügen (Zuweisung), nicht sehr viel anfangen, wenn es nicht eine weitere vorgegebene Operation gäbe, die Subskription. Sie erlaubt es, auf die Elemente einer Reihung zuzugreifen und den Datentyp der Elemente "aufzudecken".

Form:

Rowname [Indexwert]

Beispiel:

```
feld [3]
```

bezieht sich auf das dritte Element der Reihung 'feld' und hat den Datentyp INT. Für INT-Objekte haben wir aber einige Operationen, mit denen wir arbeiten können.

Beispiele:

```
feld [3] := 7;
feld [4] := feld [3] + 4;
```

Eine Subskription "schält" also vom Datentyp ein ROW ab und liefert ein Element der Reihung. Die Angabe der Nummer des Elements in der Reihung nennt man Subskript oder Index (in obigem Beispiel '3'). Der Subskript wird in ELAN in eckigen Klammern angegeben, um eine bessere Unterscheidung zu den runden Klammern in Ausdrücken zu erreichen. Ein subskribiertes ROW – Datenobjekt kann also überall da verwendet werden, wo ein entsprechender Datentyp benötigt wird (Ausnahme: nicht als Schleifenvariable).

Beispiel:

```
PROC get (ROW 10 INT VAR feld):
   INT VAR i;
   FOR i FROM 1 UPTO 10 REP
     put (i); put ("tes Element bitte:");
     get (feld [i]);
     line
   END REP
END PROC get;

PROC put (ROW 10 INT CONST feld):
   INT VAR i;
   FOR i FROM 1 UPTO 10 REP
     put (i); put ("tes Element ist:");
     put (feld [i]);
     line
   END REP
END PROC put
```

In diesen Beispielen werden Reihungen als Parameter benutzt.

Diese beiden Prozeduren werden im folgenden Beispiel benutzt um 10 Werte einzulesen und die Summe zu berechnen:

```
.....Beispiel:.....
ROW 10 INT VAR werte;
lies werte ein;
summiere sie;
drucke die summe und einzelwerte.

lies werte ein:
   get (werte).

summiere sie:
   INT VAR summe := 0, i;
   FOR i FROM 1 UPTO 10 REP
     summe INCR werte [i]
   END REP.

drucke die summe und einzelwerte:
   put (werte);
   line;
   put ("Summe:"); put (summe).
```

Da es möglich ist, von jedem Datentyp eine Reihung zu bilden, kann man natürlich auch von einer Reihung eine Reihung bilden:

```
.....Beispiel:.....
ROW 5 ROW 10 INT VAR matrix
```

Für eine "doppelte" Reihung gilt das für "einfache" Reihungen gesagte. Wiederum existieren keine Operationen für dieses Datenobjekt (außer der Zuweisung), jedoch ist es durch Subskription möglich, auf die Elemente zuzugreifen:

```
matrix [3]
```

liefert ein Datenobjekt mit dem Datentyp ROW 10 INT.

Subskribiert man jedoch 'matrix' nochmals, so erhält man ein INT:

```
matrix [2] [8]
```

(jede Subskription "schält" von Außen ein ROW vom Datentyp ab).

2.6.2 Struktur

Strukturen sind Datenverbunde wie Reihungen, aber die Komponenten können ungleichartige Datentypen haben. Die Komponenten von Strukturen heißen Felder (Reihungen: Elemente) und der Zugriff auf ein Feld Selektion (Reihungen: Subskription). Eine Struktur ist – genauso wie bei Reihungen – ein eigener Datentyp, der in einer Deklaration angegeben werden muß.

Die Deklaration einer Struktur sieht folgendermaßen aus:

- Schlüsselwort **STRUCT**
- unterschiedliche Datenobjekte in Klammern. Die Datenobjekte werden mit Datentyp und Namen angegeben
- Zugriffsrecht (**VAR** oder **CONST**)
- Name der Struktur.

Beispiel:

```
STRUCT (TEXT name, INT alter) VAR ich
```

Wiederum existieren keine Operationen auf Strukturen außer der Zuweisung und der Selektion, die es erlaubt, Komponenten aus einer Struktur herauszulösen.

Die Selektion hat folgende Form:

Objektname . Feldname

Beispiele:

```
ich . name
ich . alter
```

Die erste Selektion liefert einen TEXT–, die zweite ein INT–Datenobjekt. Mit diesen (selektierten) Datenobjekten kann – wie gewohnt – gearbeitet werden (Ausnahme: nicht als Schleifenvariable).

Zum Datentyp einer Struktur gehören auch die Feldnamen:

```
Beispiel:
STRUCT (TEXT produkt name, INT artikel nr) VAR erzeugnis
```

Die obige Struktur ist ein anderer Datentyp als im ersten Beispiel dieses Abschnitts, da die Namen der Felder zur Unterscheidung hinzugezogen werden. Für Strukturen – genauso wie bei Reihungen – kann man sich neue Operationen definieren.

Im folgenden Programm werden eine Struktur, die Personen beschreibt, die Prozeduren 'put', 'get' und der dyadische Operator HEIRATET definiert. Anschließend werden drei Paare verHEIRATET.

```
Beispiel:
PROC get (STRUCT (TEXT name, vorname, INT alter) VAR p):
   put ("bitte Nachname:"); get ( p.name);
   put ("bitte Vorname:");  get ( p.vorname);
   put ("bitte Alter:");    get ( p.alter);
   line
END PROC get;

PROC put (STRUCT (TEXT name, vorname, INT alter) CONST p):
   put (p.vorname); put (p.name);
   put ("ist");
   put (p.alter);
   put ("Jahre alt");
   line
END PROC put;

OP HEIRATET
   (STRUCT (TEXT name, vorname, INT alter) VAR w,
    STRUCT (TEXT name, vorname, INT alter) CONST m):
   w.name := m.name
END OP HEIRATET;
```

Beispiel:

```
ROW 3 STRUCT (TEXT name, vorname, INT alter) VAR frau,
                                                 mann;

personendaten einlesen;
heiraten lassen;
paardaten ausgeben.

personendaten einlesen:
   INT VAR i;
   FOR i FROM 1 UPTO 3 REP
     get (frau [i]);
     get (mann [i])
   END REP.

heiraten lassen:
   FOR i FROM 1 UPTO 3 REP
     frau [i] HEIRATET mann [i]
   END REP.

paardaten ausgeben:
   FOR i FROM 1 UPTO 3 REP
     put (frau [i]);
     put ("hat geheiratet:"); line;
     put (mann [i]); line
   END REP.
```

Reihungen und Strukturen dürfen miteinander kombiniert werden, d.h. es darf eine Reihung in einer Struktur erscheinen oder es darf eine Reihung von einer Struktur vorgenommen werden. Selektion und Subskription sind in diesen Fällen in der Reihenfolge vorzunehmen, wie die Datentypen aufgebaut wurden (von außen nach innen).

2.6.3 LET – Konstrukt für zusammengesetzte Datentypen

Die Verwendung von Strukturen oder auch Reihungen kann manchmal schreibaufwendig sein. Mit dem LET – Konstrukt darf man Datentypen einen Namen geben. Dieser Name steht als Abkürzung und verringert so die Schreibarbeit. Zusätzlich wird durch die Namensgebung die Lesbarkeit des Programms erhöht.

Form:

LET Name = Datentyp

Der Name darf nur aus Großbuchstaben (ohne Blanks) bestehen.

Beispiel:

```
LET PERSON = STRUCT (TEXT name, vorname, INT alter);

PROC get (PERSON VAR p):
   put ("bitte Nachname:"); get ( p.name);
   put ("bitte Vorname:");  get ( p.vorname);
   put ("bitte Alter:");    get ( p.alter);
   line
END PROC get;

PROC put (PERSON CONST p):
   put (p.vorname); put (p.name); put ("ist");
   put (p.alter);   put ("Jahre alt"); line
END PROC put;

OP HEIRATET (PERSON VAR f, PERSON CONST m):
   f.name := m.name
END OP HEIRATET;

ROW 3 PERSON VAR mann, frau;
```

Überall, wo der abzukürzende Datentyp verwandt wird, kann stattdessen der Name PERSON benutzt werden. Wohlgemerkt: PERSON ist kein neuer Datentyp, sondern nur ein Name, der für STRUCT (....) steht. Der Zugriff auf die Komponenten des abgekürzten Datentyps bleibt erhalten (was bei abstrakten Datentypen, die später erklärt werden, nicht mehr der Fall ist).

Neben der Funktion der Abkürzung von Datentypen kann das LET – Konstrukt auch zur Namensgebung für Denoter verwandt werden (siehe 2.3.1.2).

2.6.4 Denoter für zusammengesetzte Datentypen (Konstruktor)

Oft ist es notwendig, Datenverbunden Werte zuzuweisen (z.B.: bei der Initialisierung). Dies kann durch normale Zuweisungen erfolgen:

```
LET PERSON = STRUCT (TEXT name, vorname, INT alter);

PERSON VAR mann;

mann.name := "meier";
mann.vorname := "egon";
mann.alter := 27
```

Eine andere Möglichkeit für die Wertbesetzung von Datenverbunden ist der Konstruktor:

Form:

Datentyp : (Wertliste)

In der Wertliste wird für jede Komponente des Datentyps, durch Kommata getrennt, ein Wert aufgeführt. Besteht eine der Komponenten wiederum aus einem Datenverbund, muß innerhalb des Konstruktors wiederum ein Konstruktor eingesetzt werden.

```
LET PERSON = STRUCT (TEXT name, vorname, INT alter);

PERSON VAR mann, frau;

frau := PERSON : ( "schmitz", "lise", 25);
frau HEIRATET PERSON : ( "meier", "egon", 27)
```

Ein Konstruktor ist also ein Mechanismus, um ein Datenobjekt eines Datenverbundes in einem Programm zu notieren.

Konstruktoren sind natürlich für Reihungen auch möglich:

```
ROW 7 INT VAR feld;
feld := ROW 7 INT : ( 1, 2, 3, 4, 5, 6, 7);
```

2.7 Abstrakte Datentypen

2.7.1 Definition neuer Datentypen

Im Gegensatz zur LET – Vereinbarung für Datentypen, bei der lediglich ein neuer Name für einen bereits vorhandenen Datentyp eingeführt wird und bei der somit auch keine neuen Operationen definiert werden müssen (weil die Operationen für den abzukürzenden Datentyp verwandt werden können), wird durch eine TYPE – Vereinbarung ein gänzlich neuer Datentyp eingeführt.

Form:

TYPE Name = Feinstruktur

Der Name darf nur aus Großbuchstaben (ohne Blanks) bestehen. Die Feinstruktur (konkreter Typ, Realisierung des Datentyps) kann jeder bereits definierte Datentyp sein.

Beispiel:

```
TYPE PERSON = STRUCT (TEXT name, vorname, INT alter)
```

Der neudefinierte Datentyp wird abstrakter Datentyp genannt. Im Gegensatz zu Strukturen und Reihungen stehen für solche Datentypen noch nicht einmal die Zuweisung zur Verfügung. Ein solcher Datentyp kann genau wie alle anderen Datentypen verwendet werden (Deklarationen, Parameter, wertliefernde Prozeduren, als Komponenten in Reihungen und Strukturen usw.).

Wird der Datentyp über die Schnittstelle des PACKETs anderen Programmteilen zur Verfügung gestellt, so müssen Operatoren und/oder Prozeduren für den Datentyp ebenfalls "herausgereicht" werden. Da dann der neudefinierte Datentyp genauso wie alle anderen Datentypen verwandt werden kann, aber die Komponenten (Feinstruktur) nicht zugänglich sind, spricht man von abstrakten Datentypen.

Welche Operationen sollten für einen abstrakten Datentyp zur Verfügung stehen? Obwohl das vom Einzelfall abhängt, werden meistens folgende Operationen und Prozeduren definiert:

- 'get' – und 'put' – Prozeduren.
- Zuweisung (auch für die Initialisierung notwendig).
- Denotierungs – Prozedur (weil kein Konstruktor für den abstrakten Datentyp außerhalb des definierenden PACKETs zur Verfügung steht)

2.7.2 Konkretisierung

Um neue Operatoren und/oder Prozeduren für einen abstrakten Datentyp zu schreiben, ist es möglich, auf die Komponenten des Datentyps (also auf die Feinstruktur) mit Hilfe des Konkretisierers zuzugreifen. Der Konkretisierer arbeitet ähnlich wie die Subskription oder Selektion: er ermöglicht eine typmäßige Umbetrachtung vom abstrakten Typ zum Datentyp der Feinstruktur.

Form:

CONCR (Ausdruck)

```
Beispiel:
TYPE MONAT = INT;

PROC put (MONAT CONST m):
   put ( CONCR (m))
END PROC put;
```

Der Konkretisierer ist bei Feinstrukturen notwendig, die von elementarem Datentyp sind. Besteht dagegen die Feinstruktur aus Reihungen oder Strukturen, dann wird durch eine Selektion oder Subskription eine implizite Konkretisierung vorgenommen.

```
Beispiel:
TYPE LISTE = ROW 100 INT;

LISTE VAR personal nummer;
...
personal nummer [3] := ...
(* das gleiche wie *)
CONCR (personal nummer) [3] := ...
```

2.7.3 Denoter für abstrakte Datentypen (Konstruktor)

Denoter für neudefinierte Datentypen werden mit Hilfe des Konstruktors gebildet:

Form:

Datentyp :: (Wertliste)

In der Wertliste wird für jede Komponente des Datentyps, durch Kommata getrennt, ein Wert aufgeführt. Besteht eine der Komponenten wiederum aus einem Datenverbund, muß innerhalb des Konstruktors wiederum ein Konstruktor eingesetzt werden.

Beispiel:

```
TYPE GEHALT = INT;

GEHALT VAR meins :: GEHALT : (10000);
```

Besteht die Feinstruktur aus einem Datenverbund, muß der Konstruktor u.U. mehrfach geschachtelt angewandt werden:

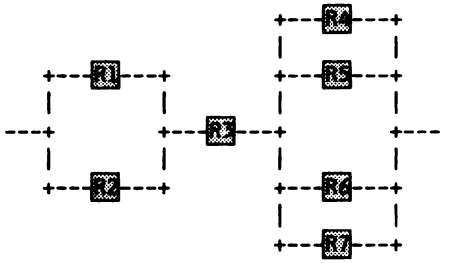
Beispiel:

```
TYPE KOMPLEX = ROW 2 REAL;

KOMPLEX CONST x :: KOMPLEX : ( ROW 2 REAL : ( 1.0, 2.0));
```

Auf die Feinstruktur über den Konkretisierer eines neudefinierten Datentyps darf nur in dem PACKET zugegriffen werden, in dem der Datentyp definiert wurde. Der Konstruktor kann ebenfalls nur in dem typdefinierenden PACKET verwandt werden.

```
PACKET widerstaende DEFINES WIDERSTAND, REIHE, PARALLEL,
                            :=, get, put:

TYPE WIDERSTAND = INT;

OP := (WIDERSTAND VAR l, WIDERSTAND CONST r):
   CONCR (l) := CONCR (r)
END OP :=;

PROC get (WIDERSTAND VAR w):
   INT VAR i;
   get (i);
   w := WIDERSTAND : (i)
END PROC get;

PROC put (WIDERSTAND CONST w):
   put (CONCR (w))
END PROC put;

WIDERSTAND OP REIHE (WIDERSTAND CONST l, r):
   WIDERSTAND : ( CONCR (l) + CONCR (r))
END OP REIHE;

WIDERSTAND OP PARALLEL (WIDERSTAND CONST l, r):
  WIDERSTAND :
    ((CONCR (l) * CONCR (r)) DIV (CONCR (l) + CONCR (r)))
END OP PARALLEL

END PACKET widerstaende
```

Dieses Programm realisiert den Datentyp WIDERSTAND und mit den Operationen eine Fachsprache, mit dem man nun leicht WIDERSTANDs – Netzwerke berechnen kann, wie z.B. folgendes:



Zur Berechnung des Gesamtwiderstandes kann nun folgendes Programm geschrieben werden:

```
Beispiel:
ROW 7 WIDERSTAND VAR r;
widerstaende einlesen;
gesamtwiderstand berechnen;
ergebnis ausgeben.

widerstaende einlesen:
   INT VAR i;
   FOR i FROM 1 UPTO 7 REP
      put ("bitte widerstand R"); put (i); put (":");
      get (r [i]);
   END REP.

gesamtwiderstand berechnen:
   WIDERSTAND CONST rgesamt :: (r [1] PARALLEL r [2]) REIHE
     r [3] REIHE (r [4] PARALLEL r [5] PARALLEL r [6]
     PARALLEL r [7]).

ergebnis ausgeben:
   line;
   put (rgesamt).
```

2.8 Dateien

Dateien werden benötigt, wenn

- Daten über die Abarbeitungszeit eines Programms aufbewahrt werden sollen;
- der Zeitpunkt oder Ort der Datenerfassung nicht mit dem Zeitpunkt oder Ort der Datenverarbeitung übereinstimmt;
- die gesamte Datenmenge nicht auf einmal in den Zentralspeicher eines Rechners paßt;
- die Anzahl und/oder Art der Daten nicht von vornherein bekannt sind.

Eine Datei ("file") ist eine Zusammenfassung von Daten, die auf Massenspeichern aufbewahrt wird. Dateien sind in bestimmten Informationsmengen, den Sätzen ("records") organisiert.

2.8.1 Datentypen FILE und DIRFILE

In ELAN gibt es zwei Arten von Dateien. Sie werden durch die Datentypen FILE und DIRFILE realisiert:

FILE:

sequentielle Dateien. Die Sätze können nur sequentiell gelesen bzw. geschrieben werden. Eine Positionierung ist nur zum nächsten Satz möglich.

DIRFILE:

indexsequentielle Dateien. Die Positionierung erfolgt direkt mit Hilfe eines Schlüssels ("key") oder Index, kann aber auch sequentiell vorgenommen werden.

Wichtig:
DIRFILEs sind auf dem EUMEL-System standardmäßig nicht implementiert! Deswegen wird auf diesen Dateityp hier nicht weiter eingegangen.

2.8.2 Deklaration und Assoziierung

Dateien müssen in einem ELAN – Programm – wie alle anderen Objekte auch – deklariert werden.

Form:

FILE VAR interner Dateibezeichner

Beispiel:

```
FILE VAR f
```

Dabei ist zu beachten, daß im EUMEL – System alle FILEs als VAR deklariert werden müssen, denn jede Lese/Schreib – Operation verändert einen FILE.

Dateien werden normalerweise vom Betriebsystem eines Rechners aufbewahrt und verwaltet. Somit ist eine Verbindung von einem ELAN – Programm, in dem eine Datei unter einem Namen – wie jedes andere Datenobjekt auch – angesprochen werden soll, und dem Betriebssystem notwendig. Dies erfolgt durch die sogenannte Assoziierungsprozedur. Die Assoziierungsprozedur 'sequential file' hat die Aufgabe, eine in einem Programm deklarierte FILE VAR mit einer bereits vorhandenen oder noch einzurichtenden Datei des EUMEL – Systems zu koppeln.

Form:

sequential file (Betriebsrichtung, Dateiname)

Es gibt folgende Betriebsrichtungen (TRANSPUTDIRECTIONs):

input:

Die Datei kann vom Programm nur gelesen werden. Durch 'input' wird bei der Assoziierung automatisch auf den ersten Satz der Datei positioniert. Ist die zu lesende Datei nicht vorhanden, wird ein Fehler gemeldet.

output:

Die Datei kann vom Programm nur beschrieben werden. Durch 'output' wird bei der Assoziierung automatisch hinter den letzten Satz der Datei positioniert (bei einer leeren Datei also auf den ersten Satz). Ist die Datei vor der Assoziierung nicht vorhanden, wird sie automatisch eingerichtet.

modify:

Im EUMEL – System gibt es noch die Betriebsrichtung 'modify'.
Die Datei kann vom Programm in beliebiger Weise gelesen und beschrieben werden. Im Gegensatz zu den Betriebsrichtungen 'input' und 'output', bei denen ausschließlich ein rein sequentielles Lesen oder Schreiben erlaubt ist, kann bei 'modify' beliebig positioniert, gelöscht, eingefügt und neu geschrieben werden.

Nach erfolgter Assoziierung ist auf den zuletzt bearbeiteten Satz positioniert. Die Datei wird automatisch eingerichtet, wenn sie vor der Assoziierung nicht vorhanden war.

Der zweite Parameter der Assozilerungsprozedur gibt an, unter welchem Namen die Datei in der Task existiert oder eingerichtet werden soll.

Beispiel:

```
FILE VAR meine datei :: sequential file (output, "xyz");
```

Folgendes Beispiel zeigt ein Programm, welches eine Datei liest und auf dem Ausgabemedium ausgibt:

Beispiel:

```
FILE VAR f :: sequential file (input, "datei1");
TEXT VAR satz;
WHILE NOT eof (f) REP
   getline (f, satz);
   putline (satz);
END REP.
```

Eine genau Übersicht der für Dateien existierende Operatoren und Prozeduren finden Sie im Teil 5.3.

2.9 Abstrakte Datentypen im EUMEL – System

2.9.1 Datentyp TASK

Tasks müssen im Rechnersystem eindeutig identifiziert werden; sogar im EUMEL – Rechner – Netz sind Tasks eindeutig identifizierbar. Dazu wird der spezielle Datentyp 'TASK' benutzt, denn die Identifizierung einer Task über den Namen ist nicht eindeutig. Der Benutzer kann ja einen Tasknamen ändern, eine Task löschen und eine neue Task mit gleichem Namen einrichten, die jedoch nicht gleich reagiert. Somit werden Tasks eindeutig über Variablen vom Datentyp TASK identifiziert.

Beispiel:

```
TASK VAR plotter := task ("PLOTTER 1")
```

Die Taskvariable 'plotter' bezeichnet jetzt die Task im System, die augenblicklich den Namen "PLOTTER 1" hat. Die Prozedur 'task' liefert den systeminternen Taskbezeichner.

Nun sind Taskvariablen auch unter Berücksichtigung der Zeit und nicht nur im aktuellen Systemzustand eindeutig. Der Programmierer braucht sich also keine Sorgen darüber zu machen, daß seine Taskvariable irgendwann einmal eine "falsche" Task (nach Löschen von "PLOTTER 1" neu eingerichtete gleichen oder anderen Namens) identifiziert. Wenn die Task "PLOTTER 1" gelöscht worden ist, bezeichnet 'plotter' keine gültige Task mehr.

Unbenannte Tasks haben alle den Pseudonamen " – ". Sie können nur über Taskvariablen angesprochen werden.

```
PROC generate shutup manager:
   TASK VAR son;
   begin ("shutup", PROC shutup manager, son)
END PROC generate shutup manager;

PROC shutup manager:
   disable stop;
   command dialogue (TRUE);
   REP
      break;
      line;
      IF yes ("shutup")
         THEN clear error;
              shutup
      FI
   PER
END PROC shutup manager
```

Ein Taskvariable wird zum Beispiel als Parameter für die Prozedur 'begin' benötigt.

begin

PROC begin (TEXT CONST son name, PROC start,
TASK VAR new task)

Die Prozedur richtet eine Sohntask mit Namen 'son name' (im Beispiel: shutup) ein, die mit der Prozedur 'start' (im Beispiel: shutup manager) gestartet wird. 'new task' (im Beispiel: son) identifiziert den Sohn, falls die Sohntask korrekt eingerichtet wurde.

2.9.2 Datentyp THESAURUS

Ein Thesaurus ist ein Namensverzeichnis, das bis zu 200 Namen beinhalten kann. Dabei muß jeder Name mindestens ein Zeichen und höchstens 100 Zeichen lang sein. Steuerzeichen (code < 32) werden im Namen folgendermaßen umgesetzt:

steuerzeichen wird umgesetzt in *"'" + code(steuerzeichen) + "'"*

Ein Thesaurus ordnet jedem eingetragenen Namen einen Index zwischen 1 und 200 (einschließlich) zu. Diese Indizes bieten dem Anwender die Möglichkeit, Thesauri zur Verwaltung benannter Objekte zu verwenden. (Der Zugriff erfolgt dann über den Index eines Namens in einem Thesaurus). So werden Thesauri u.a. von der Dateiverwaltung benutzt. Sie bilden die Grundlage der ALL – und SOME – Operatoren.

```
Beispiel:
initialisiere;
arbeite thesaurus ab.

initialisiere:
   THESAURUS VAR eine auswahl :: SOME (myself);
   TEXT VAR thesaurus element;
   INT VAR index := 0.

arbeite thesaurus ab:
   REPEAT
      get (eine auswahl, thesaurus element, index);
      IF thesaurus element = ""
         THEN LEAVE arbeite thesaurus ab
      FI;
      fuehre aktionen durch
   PER.

   fuehre aktionen durch:
      edit (thesaurus element);
      lineform (thesaurus element);
      pageform (thesaurus element);
      print (thesaurus element).
```

Dieses Beispiel führt für eine Auswahl der in der Task befindlichen Dateien nacheinander die Kommandos 'edit', 'lineform', 'pageform' und 'print' aus.

Die benutzten Operatoren und Prozeduren leisten folgendes:

SOME

THESAURUS OP SOME (TASK CONST task)
Der Operator bietet das Verzeichnis der in der angegeben Task befindlichen Dateien zum Editieren an. Namen, die nicht gewünscht sind, müssen aus dem Verzeichnis gelöscht werden.

get

PROC get (THESAURUS CONST t, TEXT VAR name, INT VAR index)
Die Prozedur liefert den nächsten Eintrag aus dem angegebenen Thesaurus 't'. 'Nächster' heißt hier, der kleinste vorhandene mit einem Index größer als 'index'. Dabei wird in 'name'der Name und in 'index'der Index des Eintrags geliefert.

2.9.3 Datenräume

Datenräume sind die Grundlage von Dateien im EUMEL – System. Einen Datenraum kann man sich als eine Sammlung von Daten vorstellen (u.U. leer). Man kann einem Datenraum durch ein Programm einen Datentyp "aufprägen". Nach einem solchen "Aufpräge" – Vorgang kann der Datenraum wie ein "normaler" Datentyp behandelt werden.

Standarddateien (FILEs) sind eine besondere Form von Datenräumen. Sie können nur Texte aufnehmen, da sie ja hauptsächlich für die Kommunikation mit dem Menschen (vorwiegend mit Hilfe des Editors bzw. Ein – / Ausgabe) gedacht sind. Will man Zahlen in einen FILE ausgeben, so müssen diese zuvor in Texte umgewandelt werden. Hierfür stehen Standardprozeduren zur Verfügung (z.B. 'put (f, 17)').

Will man aber Dateien zur Kommunikation zwischen Programmen verwenden, die große Zahlenmengen austauschen, verursachen die Umwandlungen von Zahlen in TEXTe und umgekehrt unnötigen Rechenaufwand. Zu diesem Zweck werden im EUMEL – System Datenräume eingesetzt, die es gestatten, beliebige Strukturen (Typen) in Dateien zu speichern. Solche Datenräume kann man weder mit dem Editor noch mit dem Standarddruckprogramm (print) bearbeiten, da diese ja den Typ des in dem Datenraum gespeicherten Objektes nicht kennen.

2.9.3.1 Datentyp DATASPACE

Datenräume können als eigener Datentyp (DATASPACE) in einem Programm behandelt werden. Somit können Datenräume (als Ganzes) ohne Kenntnis eines eventuell (vorher oder später) aufgeprägten Typs benutzt werden.

Als Operationen auf DATASPACE – Objekte sind nur Transporte, Löschen und Zuweisung zugelassen.

```
Beispiel:
DATASPACE VAR ds
```

Für Datenräume ist die Zuweisung definiert. Der Zuweisungsoperator (':=') bewirkt eine Kopie des Datenraums vom rechten auf den linken Operanden.

```
Beispiel:
DATASPACE VAR datenraum := nilspace;
```

Die Prozedur 'nilspace' liefert einen leeren Datenraum. Der Datenraum 'datenraum' ist also eine Kopie des leeren Datenraums.

Die Prozeduren und Operatoren für Datenräume werden im Teil 5.4.7 beschrieben.

2.9.3.2 BOUND – Objekte

Wie bereits erwähnt, kann man einem Datenraum einen Datentyp aufprägen. Dazu werden BOUND – Objekte benutzt. Mit dem Schlüsselwort **BOUND**, welches in der Deklaration vor den Datentyp gestellt wird, teilt man dem ELAN – Compiler mit, daß die Werte eines Datentyps in einem Datenraum gespeichert sind bzw. gespeichert werden sollen.

Beispiel:

```
BOUND ROW 1000 REAL VAR liste
```

Die Ankopplung des BOUND – Objekts an eine Datei erfolgt mit dem Operator **:=**.

Form:

BOUND – Objekt **:=** Datenraum

Beispiel:

```
BOUND ROW 1000 REAL VAR gehaltsliste := new ("Gehälter")
```

Die Prozedur 'new' kreiert dabei einen leeren Datenraum (hier mit dem Namen 'Gehälter'), der mit Hilfe der Zuweisung (hier: Initialisierung) an die Variable 'gehaltsliste' gekoppelt wird.

Nun kann man mit der 'gehaltsliste' arbeiten wie mit allen anderen Feldern auch. Die Daten, die in 'gehaltsliste' gespeichert, werden eigentlich im Datenraum 'Gehälter' abgelegt.

```
Beispiel:
gehaltsliste [5] := 10 000.0;          (* Traumgehalt *)
gehaltsliste [index] INCR 200.0;       (* usw.        *)
```

Man kann auch Prozeduren schreiben, die auf der Gehaltsliste arbeiten.

```
Beispiel:
PROC sort (ROW 1000 REAL VAR liste);
...
END PROC sort;
...
sort (CONCR (gehaltsliste));
...
```

Man beachte, daß der formale Parameter der Prozedur 'sort' nicht mit BOUND spezifiziert werden darf (BOUND wird nur bei der Deklaration des Objekts angegeben). Das ist übrigens ein weiterer wichtiger Vorteil von BOUND – Objekten: man kann alle Prozeduren des EUMEL – Systems auch für BOUND – Objekte verwenden, nur die Datentypen müssen natürlich übereinstimmen.

Häufige Fehler bei der Benutzung von Datenräumen

- Wenn man an ein DATASPACE – Objekt zuweist (z.B.: DATASPACE VAR ds := new ("mein datenraum")), so erhält man, wie bereits erwähnt, eine Kopie des Datenraums in 'ds'. Koppelt man jetzt 'ds' an ein BOUND – Objekt an und führt Änderungen durch, so wirken diese nur auf die Kopie und nicht auf die Quelle. Für Änderungen in der Quelle, also in der vom Datei – Manager verwalteten Datei, ist stets direkt anzukoppeln.

Beispiel:

```
BOUND ROW 10 INT VAR reihe;
INT VAR i;

PROC zeige dsinhalt (TEXT CONST datenraum):
   BOUND ROW 10 INT VAR inhalt := old (datenraum);
   INT VAR j;
   line;
   putline ("Inhalt:" + datenraum);
   FOR j FROM 1 UPTO 10 REP
      put (inhalt (j))
   PER
END PROC zeige dsinhalt;

(*  falsch: es wird auf der Kopie gearbeitet:          *)
DATASPACE VAR ds := new ("Gegenbeispiel: Zahlen 1 bis 10");
reihe := ds;
besetze reihe;
zeige dsinhalt ("Gegenbeispiel: Zahlen 1 bis 10");

(*  richtig: es wird auf dem Datenraum gearbeitet:     *)
reihe := new ("Beispiel: Zahlen 1 bis 10");
besetze reihe;
zeige dsinhalt ("Beispiel: Zahlen 1 bis 10").

besetze reihe:
   FOR i FROM 1 UPTO 10 REP
      reihe (i) := i
   PER.
```

Der Datenraum 'Gegenbeispiel: Zahlen 1 bis 10' wird nicht mit Werten besetzt, sondern die Kopie dieses Datenraums, der unbenannte Datenraum 'ds'. Auf dem direkt angekoppelten Datenraum 'Beispiel: Zahlen 1 bis 10' werden die Werte gespeichert.

- Wenn man ein DATASPACE – Objekt benutzt, ohne den Datei – Manager zu verwenden, so muß man selbst dafür sorgen, daß dieses Objekt nach seiner Benutzung wieder gelöscht wird. Das Löschen geschieht durch die Prozedur 'forget'. Ein automatisches Löschen von DATASPACE – Objekten erfolgt nicht bei Programmende (sonst könnten sie ihre Funktion als Datei nicht erfüllen). Nur durch 'forget' oder beim Löschen einer Task werden alle ihr gehörenden DATASPACE – Objekte gelöscht und der belegte Speicherplatz freigegeben.

- Ferner ist zu beachten, daß vor der Ankopplung an ein BOUND – Objekt das DATASPACE – Objekt initialisiert wird (im Normalfall mit 'nilspace').

```
DATASPACE VAR ds :=  nilspace;
BOUND ROW 1000 REAL VAR real feld := ds;
....
real feld [index] := wert;
....
forget (ds) (* Datenraum löschen,
               damit der Platz wieder verwendet wird *)
```

- Will man auf die Feinstruktur eines BOUND – Objekts zugreifen, so muß man strenggenommen den Konkretisierer benutzen:

  Form:

  CONCR (Ausdruck)

  Der Konkretisierer ermöglicht eine typmäßige Umbetrachtung vom BOUND – Objekt zum Datentyp der Feinstruktur. Ist der Zugriff jedoch eindeutig, so wird 'CONCR' automatisch vom Compiler ergänzt.

```
Beispiel:
BOUND INT VAR i := old ("i-Wert");
INT VAR x;
x := wert.

wert:
   IF x < 0
     THEN 0
     ELSE CONCR (i)
   FI.
```

In diesem Beispiel muß der Konkretisierer benutzt werden, da sonst der Resultattyp des Refinements nicht eindeutig ist (BOUND oder INT?).

2.9.3.3 Definition neuer Dateitypen

Durch die Datenräume und die Datentyp – Definition von ELAN ist es für Programmierer relativ einfach, neue Datei – Datentypen zu definieren. In der Regel reicht der Datentyp FILE für "normale" Anwendungen aus, jedoch kann es manchmal sinnvoll und notwendig sein, neue Datei – Typen für spezielle Aufgaben zu definieren.

In diesem Abschnitt soll an dem Beispiel DIRFILE (welcher zwar im ELAN – Standard definiert, aber nicht im EUMEL – System realisiert ist) gezeigt werden, wie ein neuer Datei – Datentyp definiert wird:

Beispiel:

```
PACKET dirfiles DEFINES DIRFILE, :=, dirfile, getline, ...:

LET maxsize = 1000;

TYPE DIRFILE = BOUND ROW maxsize TEXT;
(* DIRFILE besteht aus TEXTen; Zugriff erfolgt ueber einen
   Schluessel, der den Index auf die Reihung darstellt *)

OP := (DIRFILE VAR dest, DATASPACE CONST space):
   CONCR (dest) := space
END OP :=;

DATASPACE PROC dirfile (TEXT CONST name):
   IF exists (name)
      THEN old (name)
      ELSE new (name)
   FI
END PROC dirfile;

PROC getline (DIRFILE CONST df, INT CONST index,
              TEXT VAR record):
   IF index <= 0
      THEN errorstop ("access before first record")
   ELIF index > maxsize
      THEN errorstop ("access after last record")
      ELSE record := df [index]
   FI
END PROC getline;

PROC putline (DIRFILE CONST df, INT CONST index,
              TEXT VAR record):
   ...
END PROC putline;

...
END PACKET dirfiles;
```

Die Prozedur 'dirfile' ist die Assoziierungsprozedur für DIRFILEs (analog 'sequential file' bei FILEs). 'dirfile' liefert entweder einen bereits vorhandenen Datenraum oder richtet einen neuen ein. Um eine Initialisierung mit der 'dirfile' – Prozedur vornehmen zu können, braucht man auch einen Zuweisungsoperator, der den Datenraum an den DIRFILE – Datentyp koppelt.

Zugriffe auf einen DIRFILE sind nun relativ einfach zu schreiben. Im obigen Beispiel wird nur die Prozedur 'getline' gezeigt.

Nun ist es möglich, Programme zu schreiben, die den DIRFILE – Datentyp benutzen.

```
DIRFILE VAR laeufer :: dirfile ("Nacht von Borgholzhausen");
INT VAR nummer;
TEXT VAR name;

REP
   put ("Startnummer bitte:");
   get (nummer);
   line;
   put ("Name des Laeufers:");
   get (name);
   putline (laeufer, nummer, name);
   line
UNTIL no ("weiter") END REP;
...
```

2.9.4 Datentyp INITFLAG

Im Multi – User – System ist es oft notwendig, Pakete beim Einrichten einer neuen Task in dieser neu zu initialisieren. Das muß z.B. bei der Dateiverwaltung gemacht werden, da die neue Task ja nicht die Dateien des Vaters erbt. Mit Hilfe von INITFLAG – Objekten kann man zu diesem Zweck feststellen, ob ein Paket in dieser Task schon initialisiert wurde.

INITFLAG

TYPE INITFLAG
Erlaubt die Deklaration entsprechender Flaggen.

:=

OP := (INITFLAG VAR flag, BOOL CONST flagtrue)
Erlaubt die Initialisierung von INITFLAGs

initialized

BOOL PROC initialized (INITFLAG VAR flag)
Wenn die Flagge in der Task A auf TRUE oder FALSE gesetzt wurde, dann liefert sie beim ersten Aufruf den entsprechenden Wert, danach immer TRUE (in der Task A!).

Beim Einrichten von Söhnen wird die Flagge in den Sohntasks automatisch auf FALSE gesetzt. So wird erreicht, daß diese Prozedur in den neu eingerichteten Söhnen und Enkeltasks genau beim ersten Aufruf FALSE liefert.

Beispiel:

```
PACKET stack DEFINES push, pop:

INITFLAG VAR in this task := FALSE ;
INT VAR stack pointer ;
ROW 1000 INT VAR stack ;

PROC push (INT CONST value) :

   initialize stack if necessary ;
   ....

END PROC push ;

PROC pop (INT VAR value) :

   initialize stack if necessary ;
   ....

END PROC pop ;.

initialize stack if necessary :
IF NOT initialized (in this task)
   THEN stack pointer := 0

FI .

END PACKET stack
```

TEIL 3: Der Editor

Mit dem EUMEL – Editor steht für den Teil der Programmierung, der aus der Eingabe von Programmtext besteht, dasselbe komfortable Werkzeug zur Verfügung, wie für die Textverarbeitung. Merkmale des EUMEL – Editors sind die einfache Fenstertechnik und die übersichtliche Bedienung durch wenige Funktionstasten.

Eine mit dem Editor erzeugte Textdatei ist maximal 4075 Zeilen lang, die maximale Breite einer Zeile beträgt 16000 Zeichen.

3.1 Ein – und Ausschalten des Editors

Der Editor wird eingeschaltet durch Eingabe von:

```
gib kommando :
edit ("dateiname")
```

Falls eine Datei unter dem eingegebenen Namen existiert, wird ein Fenster auf dieser Datei an der Stelle geöffnet, an der zuletzt ein Zugriff auf diese Datei stattfand.

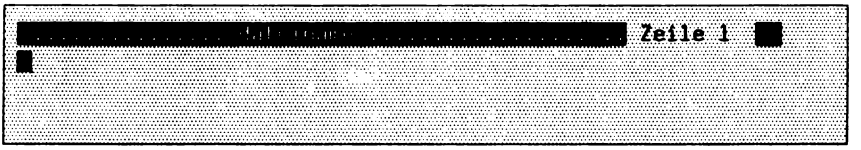
Existiert noch keine Datei unter dem angegebenen Namen in der Task, folgt eine Anfrage, ob eine Datei unter dem eingegebenen Namen neu eingerichtet werden soll:

```
gib kommando :
edit("dateiname")
"dateiname" neu einrichten (j/n) ?
```

Die Abfrage dient der Kontrolle der Schreibweise. Man kann ggf. das Einrichten der Datei ablehnen, den Dateinamen verbessern und das Kommando erneut geben.

Bei korrekter Schreibweise bejahen Sie die Kontrollfrage [1]mit

Es erscheint ein leerer Editorbildschirm. Die oberste Zeile des Bildschirms ist die Titelzeile. In ihr kann nicht geschrieben werden. Sie zeigt jedoch verschiedene nützliche Dinge an: den Namen der Datei, die Nummer der aktuellen Zeile, in der gerade geschrieben wird, Tabulatormarken, Einfügemodus, Lernmodus usw.



Wollen Sie die Schreibarbeit beenden und den Editor ausschalten, so drücken Sie die beiden Tasten

nacheinander. Sie haben damit den Editor verlassen und befinden sich wieder auf Monitor – Ebene.

3.2 Die Funktionstasten

Die Funktionstasten realisieren diejenigen Fähigkeiten des Editor, die über die reine Zeicheneingabe hinausgehen. Wo die Tasten auf Ihrem Gerät liegen, hängt von dem jeweiligen Gerätetyp ab. Die Wirkung der Tasten ist im Weiteren erläutert.

Umschalttaste

Positionierungstasten

Eingabe – / Absatztaste

Kommandotaste

Verstärkertaste

Tabulatortaste

Markiertaste

Löschtaste

Einfügetaste

Supervisortaste

Stoptaste

Weitertaste

Es kann sein, daß Tasten nicht richtig beschriftet sind. Die Installations – anleitung muß dann die Entsprechungen beschreiben. Natürlich können sich weitere Funktionstasten außer den im folgenden beschriebenen auf Ihrer Tastatur befinden. Diese haben standardmäßig jedoch keine besondere Bedeutung für den Editor.

3.3 Die Wirkung der Funktionstasten

Umschalttaste

Wird diese Taste gleichzeitig mit einer anderen betätigt, so wird ein Buchstabe in Großschreibung, bei den übrigen Tasten das obere Zeichen, ausgegeben. So wird z.B. anstelle der "9" das Zeichen ")" ausgegeben.

Positionierungstasten

Positionierung des Cursors um eine Spalten – /Zeilenposition in die jeweilige Richtung.

Eingabetaste / Absatztaste, Carriage Return, kurz: 'CR'

Diese Taste schließt die aktuelle Zeile explizit ab und es wird an den Beginn der nächsten Zeile positioniert. Einrückungen werden beibehalten.

Der EUMEL – Editor ist auf automatischen Wortumbruch voreingestellt, d.h. ein Wort, das über das 77. Zeichen der aktuellen Zeile herausreichen würde, wird automatisch in die nächste Zeile gerückt (siehe 'word wrap' 4.2.5). Die Absatztaste wird also benötigt, um explizite Zeilenwechsel und Einrückungen bei der Textformatierung zu erhalten. Eine Absatzmarke wird durch ein 'blank' hinter dem letzten Zeichen der Zeile erzeugt und ist im Editor an der Inversmarkierung am rechten Bildschirmrand zu erkennen.

Im EUMEL – System werden Kommandos auf einer Kommandozeile, auf der alle Editorfunktionen zur Verfügung stehen, eingegeben. Auf dieser Ebene beendet die Taste also ausdrücklich die Kommandoeingabe, das gegebene Kommando wird anschließend analysiert und ausgeführt.

"Verstärkertaste"; wird als Vorschalttaste bedient.

In Kombination mit anderen Funktionstasten wird deren Wirkung verstärkt.

Steht der Cursor nicht am unteren Bildrand, so wird er dorthin positioniert. Steht er am unteren Bildrand, so wird um einen Bildschirminhalt "weitergeblättert".

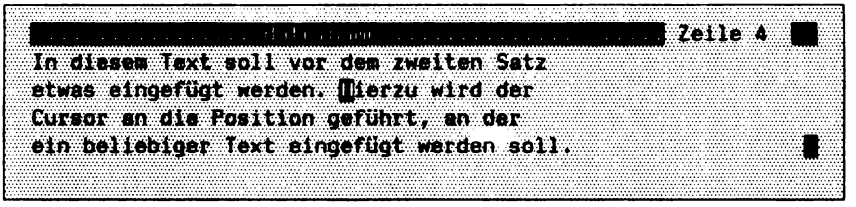
Entsprechend werden auch die Tasten 

mit der HOP – Taste verstärkt.

HOP RUBIN

Einfügen von Textpassagen. Die HOP – Taste in Verbindung mit RUBIN und RUBOUT wird zum 'verstärkten' Löschen und Einfügen verwendet.

Ab der aktuellen Position des Cursors 'verschwindet' der restliche Text. Es kann wie bei der anfänglichen Texteingabe fortgefahren werden. Die Anzeige 'REST' in der Titelzeile erinnert daran, daß noch ein Resttext existiert. Dieser erscheint nach einem neuerlichen Betätigen der beiden Tasten HOP RUBIN wieder auf dem Bildschirm (die Anzeige 'REST' verschwindet dann wieder).



Nach Betätigen der Taste HOP und RUBIN sieht der Bildschirm wie folgt aus:

Nun kann beliebig viel Text eingefügt werden. Nochmaliges Betätigen von HOP und RUBIN führt den Text – Rest wieder bündig heran.

HOP RUBOUT

Löscht die Zeile ab Cursor – Position bis Zeilenende.

Nach Betätigen der Tasten HOP und RUBOUT sieht der Bildschirm wie folgt aus:

Steht der Cursor am Zeilenanfang, wird nach HOP RUBOUT dementsprechend die ganze Zeile gelöscht und die Lücke durch Nachrücken der Folgezeilen geschlossen (HOP RUBOUT betätigen).

TAB

Tabulatortaste

Mit der Tabulatortaste werden die eingestellten Tabulatorpositionen angesprungen. Jeder Tastendruck läßt den Cursor auf die nächste eingestellte Tabulatorposition springen.

Voreingestellte Tabulatorpositionen sind die beiden Schreibgrenzen, Textanfang in der Zeile und Ende der Zeile.

Weitere Tabulatorpositionen können durch Positionierung auf die gewünschte Spalte und HOP TAB gesetzt werden. Sie können gelöscht werden, indem sie mit TAB angesprungen und mit HOP TAB ausgeschaltet werden.

Die gesamte eingestellte Tabulalation kann durch ESC TAB ein-/ und ausgeschaltet werden.

Die eingestellten Tabulatorpositionen erkennen Sie an den Tabulatorzeichen (Dachzeichen) in der obersten Bildschirmzeile.

MARK

Ein – bzw. Ausschalten der Markierung.

Bei Betätigung dieser Taste wird in einen speziellen Markierzustand geschaltet. Alles, was Sie jetzt schreiben bzw. durch Bewegen des Cursors in Richtung Dateiende kennzeichnen, steht als *markierter* Bereich für die Bearbeitung zur Verfügung. Zur besseren Sichtbarkeit wird der markierte Bereich invers zum übrigen Text dargestellt.

Wird der Cursor in eine Richtung bewegt, wird das gesamte Textstück zwischen Einschaltpunkt der Markierung und aktueller Cursorposition markiert. Rückwärtsbewegungen des Cursors verkürzen den markierten Bereich wieder.

Durch erneutes Betätigen der MARK – Taste schalten Sie den Markier – Zustand wieder aus.

Mit weiteren Kommandos kann der Bereich nun bearbeitet werden:

ESC d — Markierten Abschnitt in 'Scratch' – Datei kopieren.

ESC p — Markierten Abschnitt herauskopieren.

ESC RUBOUT — Markierten Abschnitt löschen.

Der mit ESC p oder d kopierte Bereich kann beliebig oft in derselben oder einer anderen Datei ein/angefügt werden.

Der mit ESC RUBOUT gelöschte Abschnitt kann genau einmal durch ESC RUBIN an anderer Stelle derselben Datei eingefügt werden.

(vgl. ESC – Taste, Operationen auf Markierungen, 3 – 15)

RUBIN

Ein – bzw. Ausschalten des Einfügemodus.

Das Betätigen der Taste schaltet in den Einfügemodus. Der Zustand wird durch das Wort "RUBIN" im linken Drittel der Titelzeile der Datei angezeigt. Vor dem Zeichen, auf dem der Cursor steht, wird eingefügt. Nochmaliges Betätigen der Taste schaltet den Einfügemodus aus.

RUBOUT

Löschtaste

Das Zeichen, auf dem der Cursor steht, wird gelöscht. Wenn der Cursor, wie bei fortlaufender Eingabe üblich, hinter dem letzten Zeichen einer Zeile steht, wird das letzte Zeichen gelöscht.

3.4 ESC Kommandos

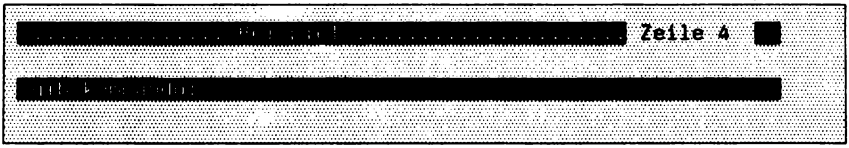
ESC

Kommandotaste

Mit der ESC – Taste in Kombination mit einer Folgetaste werden vordefinierte Aktionen ausgelöst. Es gibt Aktionen, die vorprogrammiert zur Verfügung stehen, und Sie selbst können weitere hinzufügen.

Der Kommandodialog wird eingeschaltet durch:

ESC ESC



Der Kommandodialog ermöglicht die Eingabe von beliebigen Kommandos ohne den Editor verlassen zu müssen. Insbesondere Such – und Kopieroperationen stellen auch für den Programmierer nützliches Werkzeug dar (siehe 3.5).

Auf der Kommandozeile kann jedes Kommando gegeben werden. Die Kommandozeile kann wie eine normale Textzeile editiert werden. Nach CR verschwindet die Kommandozeile und das Kommando wird ausgeführt.

Falls ein Fehler auftritt erfolgt eine entsprechende Fehlermeldung in der Kopfzeile und die Kommandozeile erscheint erneut.

Um ein weiteres Editor – Fenster zu 'öffnen', betätigt man im Editor

Betätigt man ESC e ungefähr in der Mitte des Bildschirms, hat man das Fenster auf die neue Datei in der unteren Hälfte des Bildschirms und die 'alte' Datei in der oberen Bildschirmhälfte. Zunächst wird der Dateiname erfragt. Nach dessen Eingabe und CR wird ein Fenster auf eröffnet. Die obere linke Ecke des Fensters befindet sich an der aktuellen Cursor – Position. Dabei darf sich der Cursor nicht zu sehr am rechten oder unteren Rand befinden, weil das Fenster sonst zu klein würde. In diesem 'Fenster' kann man dann genauso arbeiten wie im 'normalen' Editor.

Mit der Tastenfolge

wechselt man von einem Fenster (zyklisch) in das benachbarte. Es gibt eine Hierarchie zwischen den Fenstern in der Reihenfolge, in der eines im anderen eingerichtet worden ist. Gibt man

in einem Fenster, so verschwindet dieses und alle darin eingeschachtelten Fenster, und man befindet sich im übergeordneten Fenster.

Durch

schreibt man einen markierten Teil in eine 'Scratch' – Datei (nicht editierbarer Zwischenspeicher); durch ESC p wird ein markierter Text aus der Ursprungsdatei entfernt und in die 'Scratch' – Datei geschrieben. Im Gegensatz dazu wird er durch ESC d kopiert. Durch

fügt man ihn in eine andere (oder dieselbe) Datei ein. Im Unterschied zu ESC RUBIN wird die temporäre Datei dadurch nicht entleert.

Die für ESC p, bzw. ESC d benutzte 'Scratch' – Datei, die nicht editierbar ist, ist nicht mit dem sogenannten Notizbuch zu verwechseln. Das Notizbuch ist eine Datei, in der alle Editorfunktionen benutzt werden können, auf die jedoch ohne Angabe eines Dateinamens durch

ab der aktuellen Cursorposition ein Fenster eröffnet wird. Das Notizbuch nimmt insbesondere Fehlermeldungen und Meldungen bei der Übersetzung von Programmen auf.

erlaubt vom äußeren Fenster aus alle eingeschachtelten Fenster zu verlassen.

Vorbelegte Tasten

ESC q	Verlassen des Editors bzw. der eingeschachtelten Fenster.
ESC e	Weiteres Editorfenster einschalten.
ESC n	Notizbuch 'anzeigen'.
ESC v	Dateifenster auf ganzen Bildschirm vergrößern bzw. Bildschirm rekonstruieren (eingeschachteltes Fenster verlassen).
ESC w	Dateiwechsel beim Fensterediter.
ESC f	Nochmalige Ausführung des letzten Kommandos.
ESC b	Das Fenster wird auf den linken Rand der aktuellen (ggf. verschobenen) Zeile gesetzt.
ESC →	Zum nächsten Wortanfang.
ESC ←	Zum vorherigen Wortanfang.
ESC 1	Zum Anfang der Datei.
ESC 9	Zum Ende der Datei.

Operationen auf Markierungen

ESC RUBOUT	Markiertes "vorsichtig" löschen.
ESC RUBIN	Mit ESC RUBOUT vorsichtig Gelöschtes einfügen.
ESC p	Markiertes löschen und in die Notiz – Datei schreiben. Kann mit ESC g an anderer Stelle reproduziert werden.
ESC d	Duplizieren: Markiertes in die Notiz – Datei kopieren, anschließend die Markierung abschalten. Kann mit ESC g beliebig oft reproduziert werden.
ESC g	Mit ESC p gelöschten oder mit ESC d duplizierten Text an aktuelle Cursor – Stelle schreiben, d.h. Notiz – Datei an aktueller Stelle einfügen.

Zeichen schreiben

Diese Tasten sind standardmäßig so vorbelegt wie hier aufgeführt, sie können aber von Benutzern und in Anwenderprogrammen geändert werden.

ESC a	Schreibt ein ä.
ESC A	Schreibt ein Ä.
ESC o	Schreibt ein ö.
ESC O	Schreibt ein Ö.
ESC u	Schreibt ein ü.
ESC U	Schreibt ein Ü.
ESC s	Schreibt ein ß.
ESC (	Schreibt eine [.
ESC)	Schreibt eine].
ESC <	Schreibt eine {.
ESC >	Schreibt eine }.
ESC #	Schreibt ein #, das auch gedruckt werden kann.
ESC -	Schreibt einen (geschützten) Trennstrich, siehe Textverarbeitung.
ESC k	Schreibt ein (geschütztes) "k", siehe Textverarbeitung.
ESC blank	Schreibt ein (geschütztes) Leerzeichen, siehe Textverarbeitung.

Kommando auf Taste legen

ESC ESC	Kommandodialog einschalten
ESC ! taste	Im Kommandodialog: Geschriebenes Kommando auf Taste legen.
ESC ? taste	Im Kommandodialog: Auf 'taste' gelegtes Kommando zum Editieren anzeigen.
ESC k	Im Kommandodialog: Das zuletzt editierte Kommando (einzellige ELAN – Programm) anzeigen.

Der Lernmodus

Der Lernmodus ermöglicht beliebige Tastensequenzen zu speichern und auf eine Taste 't' zu legen. Durch ESC t wird die gesamte Sequenz ausgeführt.

Nicht belegt werden können die vom System vorbelegten Tasten (3 – 14).

Beispielsweise könnte es für einen Programmierer sinnvoll sein die Tastenfolge 'THEN' 'CR' '→' '→' '→' '→' auf die Taste T zu legen. Durch ESC T wird 'THEN' in die aktuelle Zeile geschrieben und der Cursor mit passender Einrückung in die Folgezeile gesetzt.

ESC HOP Lernen einschalten.

ESC HOP taste Lernen ausschalten und Lernsequenz auf 'taste'legen.

ESC HOP HOP Gelerntes vergessen. Bedingung ist, daß man die Lernsequenz in der Task löscht, in der man sie hat lernen lassen.

A C H T U N G :

Der Lernmodus bleibt eingeschaltet, auch wenn der Editor beendet wird. Dann werden die folgenden Monitor – Kommandos usw. usf. 'gelernt'. Durch unsinniges 'Lernen' lassen sich schlimmstenfalls beliebige Verwüstungen anrichten.

Der Lernmodus wird in der Editor – Kopfzeile angezeigt. Falls der Editor beendet wird, ohne den Lernmodus auszuschalten, erfolgt eine Warnung auf Monitor – Ebene.

Um den Lernmodus zu beenden drücken Sie:

ESC HOP HOP

Dadurch wird der Lernmodus ausgeschaltet und nichts gelernt, die Gefahr ist gebannt.

SUPERVISOR – Taste

Betätigen Sie diese Taste im Editor, dann unterbrechen Sie Ihre Editierarbeit und erhalten die Meldung

```
                    EUMEL Version 1.8/M

          gib supervisor kommando:



   ESC ? --> help
   ESC b --> begin("")              ESC h --> halt
   ESC c --> continue("")           ESC s --> storage info
   ESC q --> break                  ESC t --> task info
```

Wollen Sie nun im Editor fortfahren bzw. haben Sie irrtümlich die SV – Taste betätigt, dann geben Sie das Kommando

```
gib supervisor kommando :
continue ("meine task")
```

(falls Ihre Task, in der Sie arbeiteten, wirklich "meine task" hieß!)

Um Ihren in Bearbeitung befindlichen Text wieder vollständig auf dem Bildschirm zu sehen, betätigen die die Tasten

Sie sind wieder an der Stelle, an der Sie den Text mit der SV – Taste verlassen haben, und können normal weiterarbeiten.

Achtung: Die SV – Taste kann, je nach Terminal, durch das Betätigen von zwei Tasten gleichzeitig realisiert sein (oft 'CTRL b'). Beachten Sie die Beschreibung Ihrer Tastatur!

Für die Programmierung ist die Tastenfolge SV ESC h von Bedeutung, da hierdurch der Fehler 'halt vom Terminal' erzeugt wird. Dadurch können unerwünscht laufende Programme abgebrochen werden.

Unterbrechen einer Ausgabe (oft auch als CTRL a realisiert).

Haben Sie diese Taste aus Versehen betätigt, erkennen Sie dies daran, daß der Editor nicht "reagiert". Betätigen Sie die WEITER – Taste (oft auch CTRL c).

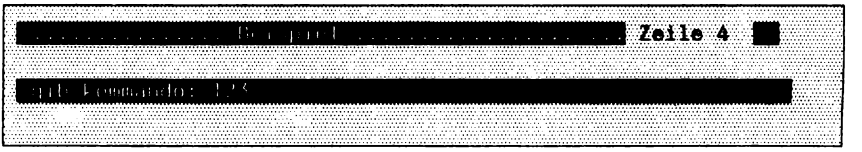
Unterbrochene Ausgabe fortsetzen.

Ein mit der STOP – Taste angehaltene Ausgabe können Sie durch Betätigen der WEITER – Taste fortsetzen.

VORSICHT: Die STOP – Taste unterbricht nur die Ausgabe auf den Bildschirm. Zeichen, die während des STOP eingegeben werden, werden gespeichert und nach 'WEITER' ausgegeben!

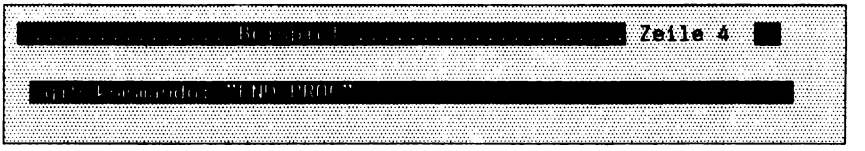
3.5 Positionieren, Suchen, Ersetzen im Kommandodialog

Um das Editorfenster auf eine bestimmte Zeile zu positionieren wird einfach diese Zeilennummer angegeben.

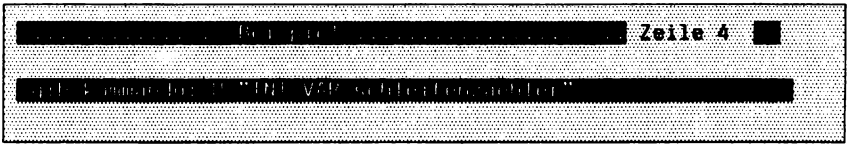


Falls die Zeilenzahl der Datei geringer als die angegebene Zeilennummer ist, wird auf die letzte Zeile positioniert.

Um das Editorfenster auf ein bestimmtes Textstück zu positionieren, wird der gesuchte Text, ggf. mit Suchrichtung angegeben.

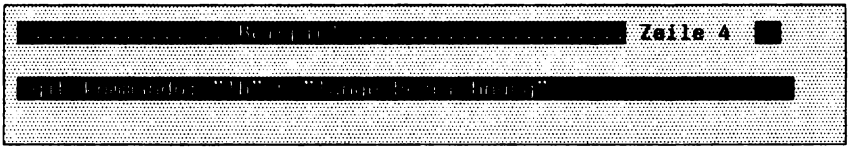


Die Suchrichtung kann durch 'D' (down) oder 'U' (up) zusätzlich spezifiziert werden.



Um beliebige Texte durch andere zu ersetzen, dienen die Operatoren 'C' (change) bzw. 'CA' (change all).

Bei Ausführung dieses Kommandos wird zunächst nach unten in der editierten Datei nach dem zu ersetzenden Text gesucht. Wenn der Text gefunden wird, wird er durch den hinter dem Operator stehenden Text ersetzt.



Bei Anwendung von 'CA' wird jedes Auftreten des gesuchten Textes ab der Cursorposition durch den Ersatztext ersetzt, bis das Dateiende errreicht ist.

Weitere Erklärungen zum Suchen und Ersetzen in 5.5.

Weitere Hilfen

Textabschnitt an anderer Stelle der Datei einsetzen:

- Abschnitt markieren und mit ESC d zwischenspeichern.

- Zweites Editorfenster auf die Datei mit ESC e öffnen.

- Nach ESC ESC Zeilennummer oder Suchbegriff angeben und mit ESC g Abschnitt an der gewünschte Stelle einsetzen.

Textabschnitt schnell herauskopieren und sichern:

- Gewünschten Abschnitt markieren

- ESC ESC PUT("dateiname") CR
  Der Abschnitt wird in die Datei 'dateiname' geschrieben. Falls die Frage 'dateiname' löschen (j/n) verneint wird, wird der Abschnitt, an das Ende der Datei angefügt. Dadurch können Textabschnitte schnell gesammelt werden.

Komplette Datei in die editierte Datei einfügen:

- ESC ESC GET("dateiname") CR
  Der komplette Inhalt von 'dateiname' wird an die aktuelle Position geschrieben.

Breitere Zeile erzeugen:

- ESC ESC limit(123) CR
  Die Zeilenbreite wird auf 123 Zeichen geändert. Der maximal zulässige Wert ist 16000. Dieser Wert bezieht sich auf den Zeilenumbruch. Bei Zeilenbreite > 77 wird nur die aktuelle Zeile verschoben. Um für den ganzen Bildschirm die rechte Seite der Datei zu sehen, kann die linken Spalte des Bildschirmfenster neu gesetzt werden:

  ESC ESC margin(60) CR

  Die Normaleinstellung wird durch 'limit(77)' und 'margin(1)' wiederhergestellt.

TEIL 4: Kommandosprache

In Teil 4 sind diejenigen Kommandos beschrieben, die erfahrungsgemäß eher der Handhabung der Arbeitsumgebung zuzurechnen sind. Es ist den Verfassern bewußt, daß Auswahl und Zusammenstellung recht willkürlich sind, weil eine klare Abgrenzung zum Teil 5, welcher die Kommandos, die dem Thema: 'Programmierung' zugeordnet werden, nicht möglich ist.

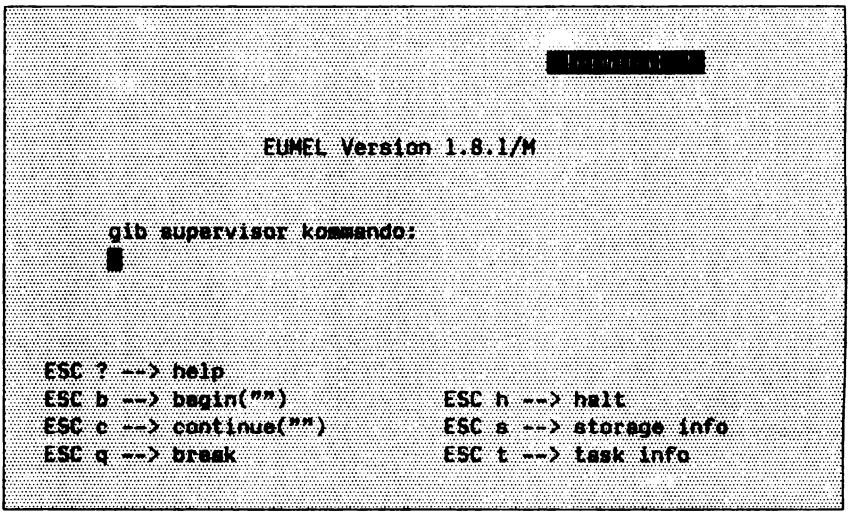
Der Teil 4 ist in die Themen:

– 4.1.	Supervisor – Kommandos
– 4.2.1	Hilfs – und Informationsprozeduren
– 4.2.2	Thesaurus
– 4.2.3	Tasks
– 4.2.4	Handhabung von Dateien
– 4.2.5	Editor
– 4.2.6	Dateitransfer
– 4.2.7	Passwortschutz
– 4.2.8	Archiv

gegliedert. Insbesondere zu 4.2.4 ist anzumerken, daß nur Kommandos, die ganze Dateien betreffen hier erläutert sind. Kommandos, die Dateiinhalte betreffen (Suchen, Ersetzen etc.) sind in 3.5, bzw. 5.3 beschrieben.

4.1 Supervisor

Es gibt genau sieben vom Supervisor akzeptierte Kommandos. Diese Kommandos können gegeben werden wenn nach dem Einschalten des Geräts oder dem Abkoppeln einer Task die SV – Taste gedrückt wurde und die sogenannte EUMEL – Tapete erscheint.



Desweiteren kann SV in einer Task gedrückt werden, um durch ESC h einen Programmabbruch einzuleiten.

Im Gegensatz zu den im weiteren beschriebenen, durch ELAN Prozeduren realisierten Kommandos, sind diese Supervisor – Kommandos nicht als Prozeduren im System und mithin nicht durch 'help (...)' anzeigbar.

begin

PROC begin (TEXT CONST taskname)
Richtet eine neue Task als Sohn von PUBLIC ein.

PROC begin (TEXT CONST taskname, vatertask)
Richtet eine neue Task als Sohn der Task 'vatertask' ein, falls die Vater – Task eine Manager – Task ist. Falls diese Task keinen Managerstatus besitzt, passiert nichts! In diesem Falle muß das Kommando durch SV abgebrochen werden.

FEHLER : "taskname" existiert bereits
"vatertask" gibt es nicht

continue

PROC continue (TEXT CONST taskname)
Eine existierende Task wird an das Terminal des Benutzers angekoppelt.

FEHLER : "taskname" gibt es nicht

Falls 'begin' oder 'continue' trotz korrekter Voraussetzungen kein Resultat zeigen, 'hängt' die betroffene Task. Beim 'begin' Kommando kann das der Fall sein, falls die Vater – Task nicht durch 'break' abgekoppelt wurde, sondern mit SV verlassen wurde. In diesem Fall muß das Kommando durch SV abgebrochen werden, die Vater – Task angekoppelt und mit ESC q korrekt abgekoppelt werden.

break

PROC break
Das Terminal wird vom Rechner abgekoppelt.

halt

PROC halt
Das laufende Programm der dem Terminal aktuell zugeordneten Task wird abgebrochen.

Falls in der an das Terminal gekoppelten Task ein laufendes Programm abgebrochen werden soll, muß zunächst durch SV der Supervisor aufgerufen werden. Durch das Supervisor – Kommando 'halt' wird der Fehler 'halt from terminal' induziert. Das Programm wird wie durch jeden anderen Fehler abgebrochen, falls nicht 'disable stop' gesetzt wurde!

storage info

PROC storage info
Informationsprozedur über den belegten und den verfügbaren Hintergrund – Speicher des gesamten Systems in KByte[1].

Das Terminal wird unmittelbar abgekoppelt!

task info

PROC task info
Informiert über alle Tasknamen im System unter gleichzeitiger Angabe der Vater/Sohn – Beziehungen durch Einrückungen.

help

PROC help
Kurzbeschreibung der SV – Kommandos.

1) Bei der derzeit aktuellen '+' Version EUMEL 1.8.1/M+ sind die beiden Angaben mit 4 zu multiplizieren !

4.2 Monitor

Unter dem Stichwort Monitor – Kommandos sind an dieser Stelle Kommandos beschrieben, die ständig zur Handhabung der Arbeitsumgebung benutzt werden. Gleichwohl sei sofort darauf hingewiesen, daß jedes ELAN Programm dem Monitor zur Ausführung übergeben werden kann. Es gibt also keine speziellen Monitor – Kommandos, sondern nur eine Reihe von Prozeduren (= Kommandos), die in dieser Umgebung erfahrungsgemäß besonders häufig benutzt werden.

4.2.1 Hilfs – und Informationsprozeduren

– Pakete, Prozeduren Parameter	:	packets, bulletin , help
– Tasksystem zeigen	:	task info , task status
– Speicherplatz zeigen	:	storage , storage info

4.2.2 Thesaurus

– besondere Thesauri	:	ALL , all , SOME , remainder
– Verknüpfung		+ , – , /

4.2.3 Taskoperationen

– besondere Tasknamen	:	archive , brother , father , myself printer , public , son , supervisor
– Terminal abkoppeln	:	break
– Task löschen	:	end
– Manager – Task	:	global manager , free global manager
– Umbenennen der Task	:	rename myself

4.2.4 Handhabung von Dateien

copy , edit , forget , list , rename , show

4.2.5 Editor

– Editieren	:	edit , editget , show
– Tastenbelegung	:	kommando auf taste (legen) , lernsequenz auf taste (legen) , std tastenbelegung , taste enthält kommando , word wrap

4.2.6 Transfer

– Datei holen	:	fetch , fetchall
– Datei senden	:	save , saveall
– Drucken	:	print
– Datei löschen	:	erase

4.2.7 Passwortschutz

– 'begin' absichern	:	begin password
– 'continue' absichern	:	task password
– Dateien absichern	:	enter password
– Systemzweig sichern	:	family password

4.2.8 Das Archiv

– Reservieren/freigeben	:	archive , release
– Formatieren	:	format
– Löschen	:	clear
– Kontrollesen	:	check

4.2.1 Hilfsprozeduren

Die drei Prozeduren listen ihre Ausgabe jeweils in eine temporäre Datei, die mit 'show' (s. 4.2.5) gezeigt wird.

packets

PROC packets
Auflisten der Namen aller insertierten Pakete in der Task.

bulletin

PROC bulletin (TEXT CONST paket name)
Listen aller in der DEFINES – Liste des Pakets mit dem Namen "paket name" enthaltenen Prozeduren.

FEHLER : ... ist kein Paketname

PROC bulletin
Es wird eine Liste aller bisher insertierten Objekte erstellt. Diese Liste ist paketweise sortiert. 'bulletin' zeigt also eine Liste aller Prozeduren an, die in der Task benutzt werden können.

help

PROC help (TEXT CONST name)
Listen aller Prozeduren / Operatoren mit dem Namen "name". Der Name des Packets in dessen Schnittstelle die Prozedur steht wird mit ausgegeben.

Falls es kein Objekt des erfragten Namens gibt, erfolgt die Ausgabe:

unbekannt "name".

Beispiel:

```
gib kommando :
help("save")
```

liefert:

```
PACKET nameset:

     save.......... (THESAURUS CONST, TASK CONST)
     save.......... (THESAURUS CONST)

PACKET globalmanager:

     save.......... (DATASPACE CONST, TEXT CONST, TASK CONST)
     save.......... (TEXT CONST, TASK CONST)
     save.......... (TEXT CONST)
     save..........
```

Desweiteren kann auch nach Prozedurnamen gesucht werden, die nur annähernd bekannt sind, indem ein Suchmuster spezifiziert wird. Das Suchmuster besteht aus dem bekannten Teil des Namens und dem Operator '*', der vor und/oder nach dem Suchbegriff gesetzt werden kann. '*' bezeichnet eine beliebige (auch leere) Zeichenkette.

Beispiel: Gesucht werden die verschiedenen 'info' Prozeduren:

```
gib kommando :
help("*info*")
```

```
taskinfo..... (INT CONST, INT CONST)
taskinfo..... (INT CONST, FILE VAR)
taskinfo..... (INT CONST)
taskinfo.....
editinfo..... (FILE VAR, INT CONST)
editinfo..... (FILE CONST)  --> INT
storageinfo....
```

Dieser Stern darf nicht mit dem 'joker' des 'Pattern Matching' verwechselt werden. In der 'help' Prozedur darf '*' nicht in den Suchbegriff eingesetzt werden, sondern nur an Wortanfang und – Ende gesetzt werden.

Informationsprozeduren

storage

INT PROC storage (TASK CONST task)
Informationsprozedur über den logisch belegten Hintergrund – Speicher der Task. (Angabe in KByte, bzw. 4KB Einheiten bei der '+' – Version)

```
gib kommando :
put(storage(myself))
1234

gib kommando :
```

storage info

PROC storage info
Informationsprozedur über den belegten und den verfügbaren Hintergrund – Speicher des gesamten Systems. Die Ausgabe erfolgt in KByte, bei der aktuellen '+' – Version in 4 KByte Einheiten.

```
gib kommando :
storage info
1234K von 12000K

gib kommando :
```

task info

PROC task info
Informiert über alle Tasknamen im System unter gleichzeitiger Angabe der Vater/Sohn – Beziehungen (Angabe durch Einrückungen).

PROC task info (INT CONST art)
Informiert über alle Tasks im System. Mit 'art' kann man die Art der Zusatz – Information auswählen.

art = 1: entspricht 'task info' ohne Parameter, d.h. es gibt nur die Tasknamen unter Angabe der Vater/Sohn – Beziehungen aus.

art = 2: gibt die Tasknamen aus. Zusätzlich erhalten Sie Informationen über die verbrauchte CPU – Zeit der Task, die Priorität, den Kanal, an dem die Task angekoppelt ist, und den eigentlichen Taskstatus. Hierbei bedeuten:

0	– busy –	Task ist aktiv.
1	i/o	Task wartet auf Beendigung des Outputs oder auf Eingabe.
2	wait	Task wartet auf Sendung von einer anderen Task.
4	busy – blocked	Task ist rechenwillig, aber blockiert[1].
5	i/o – blocked	Task wartet auf I/O, ist aber blockiert.
6	wait – blocked	Task wartet auf Sendung, ist aber blockiert. Achtung: Die Task wird beim Eintreffen einer Sendung automatisch entblockiert.
> 6	dead	

art = 3: wie 2, aber zusätzlich wird der belegte Speicher angezeigt. (Achtung: Prozedur ist zeitaufwendig!).

1) Eine Blockierung kann von 'Scheduler' veranlaßt werden (siehe Systemhandbuch)

```
gib kommando :
task info(2)
```

liefert:

```
15.05.87   10:39                          CPU     PRIO CHAN STATUS
SUPERVISOR..........................  0000:19:47    0    -   wait
  -.........  ......................  0000:07:54    0    -   wait
  SYSUR.............................  0000:34:02    0    -   wait
     shutup dialog.....  ...........  0000:05:26    0    -   i/o
     configurator......................  0000:04:17    0    -   wait
     OPERATOR..........................  0000:00:14    0    -   i/o
     ARCHIVE...........................  0000:10:33    0   31   wait
     net...............................  0006:41:56    0    -   wait
        net timer.....................  0000:02:48    2    -   i/o
        net port......................  0000:40:23    0    7   wait
     PRINTER...........................  0000:05:59    0    -   wait
        -..............................  0000:00:11    0    -   wait
UR.....................................  0000:02:11    0    -   wait
  PUBLIC.............  ...............  0002:02:03    0    -   wait
     task1.............................  0000:41:50    0    -   -busy-
     task2.............................  0000:03:10    0    -   i/o
     task3.............................  0000:57:28    0    1   -busy-
```

PROC task info (INT CONST art, FILE VAR infodatei)
Wie oben, die Ausgabe wird jedoch in die Datei 'infodatei' geschrieben.

```
FILE VAR info := sequential file(output,"infodatei") ;
taskinfo(3, info);
```

PROC task info (INT CONST art, stationsnr)
Ermöglicht im Netzbetrieb 'task info' über die Station mit der Nummer 'stationsnr'.

```
gib kommando :
taskinfo(1,12) ;
```

task status

PROC task status
Informationsprozedur über den Zustand der eigenen Task. Informiert über
- Name der Task, Datum und Uhrzeit;
- verbrauchte CPU – Zeit;
- belegten Speicherplatz;
- Kanal, an den die Task angekoppelt ist;
- Zustand der Task (rechnend u.a.m.);
- Priorität.

PROC task status (TASK CONST t)
Wie obige Prozedur, aber über die Task mit dem internen Tasknamen 't'.

```
 gib kommando :
 task status (public)

15.05.87 10:30   TASK: PUBLIC

Speicher: 1234K
CPU Zeit: 0011.12:23
Zustand : wait, (Prio 0), Kanal -
```

4.2.2 Thesaurus

Ein Thesaurus ist ein Namensverzeichnis, das bis zu 200 Namen beinhalten kann. Dabei muß jeder Namen mindestens ein Zeichen und darf höchstens 100 Zeichen lang sein. Steuerzeichen (code < 32) in Namen werden umgesetzt (siehe 2.9.2).

Thesauri werden unter anderem von der Dateiverwaltung benutzt, um das Dateiverzeichnis einer Task zu führen.

Man kann einen Thesaurus selbst erstellen, indem eine Datei z.B. mit Namen von Dateien gefüllt wird. Diese Datei kann dann als Thesaurus für weitere Aktionen dienen.

– Thesaurus liefern	ALL , all , SOME , remainder
– Auswählen	LIKE
– Verknüpfen	+ , – , /

ACHTUNG : Bei der Verwendung von Thesaurus Operationen in Verbindung mit 'fetch', 'save' etc. ist zu beachten, daß mit 'SOME', 'ALL' und 'all' zunächst nur eine Auswahl aus einer Liste getroffen wird. Zusätzlich muß das Ziel oder die Quelle des Dateitransfers vereinbart werden.

Ein beliebter Fehler ist z.B.: 'fetch (ALL archive)'.

Hier ist nicht weiter spezifiziert, von wo Dateien geholt werden sollen – also werden sie von 'father' geholt! (s. 4.2.5)

Falls die Dateien vom Archiv geholt werden sollen, ist das Archiv als Quelle zu benennen:

Also : 'fetch (ALL archive, archive)' = Hole alle Dateien, die in dem Thesaurus von 'archive' sind von der Task 'archive'.

ALL

THESAURUS OP ALL (TASK CONST task)
Liefert einen Thesaurus, der alle Dateinamen der angegebenen Task enthält.

THESAURUS OP ALL (TEXT CONST dateiname)
Liefert einen Thesaurus, der die in der angegebenen Datei vorhandenen Namen (jede Zeile ein Name) enthält.

all

THESAURUS PROC all
Liefert einen Thesaurus, der alle Dateinamen der eigenen Task enthält. Entspricht 'ALL myself'.

SOME

THESAURUS OP SOME (THESAURUS CONST thesaurus)
Bietet den angegebenen Thesaurus zum editieren an. Dort können nicht erwünschte Namen gestrichen werden.

THESAURUS OP SOME (TASK CONST task)
Aufruf von: SOME ALL task.

THESAURUS OP SOME (TEXT CONST dateiname)
Aufruf von: SOME ALL dateiname.

remainder

PROC remainder
Liefert nach einem 'errorstop' die noch nicht bearbeiteten Dateien.

```
gib kommando :
save all (archive)

'"....." kann nicht geschrieben werden (Archiv voll)'
```

Nachdem man eine neue Floppy ins Archivlaufwerk gelegt hat, kann man mit

```
gib kommando :
save (remainder, archive)
```

den Rest der Dateien auf die nächste Floppy sichern.

LIKE

THESAURUS OP LIKE (THESAURUS CONST thesaurus, TEXT CONST muster)
Alle im Thesaurus enthaltenen Dateien, die dem 'muster' entsprechen sind im Ergebnisthesaurus enthalten.

(Die Syntax von 'muster' ist bei der Beschreibung des Pattern – Matching (5.4) beschrieben)

```
gib kommando :
print (all LIKE "*.p")
```

Alle Dateien, deren Name mit '.p' endet, werden gedruckt.

+

THESAURUS OP + (THESAURUS CONST links, rechts)
Liefert die Vereinigungsmenge von 'links' und 'rechts'.
Achtung: Die Vereinigungsmenge enthält keine Namen mehrfach.

THESAURUS OP + (THESAURUS CONST links, TEXT CONST rechts)
Fügt dem Thesaurus 'rechts' zu, wenn 'rechts' noch nicht im Thesaurus enthalten ist.

-

THESAURUS OP – (THESAURUS CONST links, rechts)
Liefert die Differenzmenge. Achtung: Die Differenzmenge enthält keine Namen mehrfach.

THESAURUS OP – (THESAURUS CONST links, TEXT CONST rechts)
Nimmt den Namen 'rechts' aus dem Thesaurus.

```
gib kommando :
fetch(ALL father - ALL myself)
```

/

THESAURUS OP / (THESAURUS CONST links, rechts)
Liefert die Schnittmenge
Achtung: Die Schnittmenge enthält keine Namen mehrfach.

4.2.3 Tasks

Zur Identifizierung von Tasks dienen sogenannte 'interne Taskbezeichner'. Ein solcher Taskbezeichner wird beim Einrichten einer neuen Task vergeben. Interne Taskbezeichner sind auch unter Berücksichtigung der Zeit eindeutig.

Der Zugriff auf interne Taskbezeichner erfolgt über Prozeduren und Operatoren, die auf Objekte des Datentyps TASK (siehe 2.9.1) angewandt werden.

Zusätzlich zum internen Tasknamen, der nicht auszugeben ist, haben Tasks meistens einen Namen[1)].

Aus Benutzersicht können benannte Tasks innerhalb eines Rechners vollständig und eindeutig über ihren Namen identifiziert werden.

– Task liefern		/ , task , niltask
– Verwandtschaften		brother , father , myself , son
– Ausgezeichnete Tasks	:	archive , printer , public , supervisor
– Namen liefern		name
– Tasknamen ändern		rename myself
– Reservieren bes. Tasks		reserve

1) Unbenannte Tasks haben den Pseudonamen " – ".

/

TASK OP / (TEXT CONST taskname)
Liefert die Task des angegebenen Namens, falls sie existiert. Der eigene Katalog wird automatisch aktualisiert

(identisch mit der PROC task (TEXT CONST taskname).

FEHLER : "taskname" gibt es nicht

TASK OP / (INT CONST station number, TEXT CONST name)
Liefert im Netzbetrieb die Task des angegebenen Namen von der Station mit der angegebenen Nummer.

niltask

TASK CONST niltask
Bezeichner für "keine Task". So liefern die Prozeduren 'son', 'brother' und 'father' als Resultat 'niltask', wenn keine Sohn –, Bruder – oder Vatertask existiert.

task

TASK PROC task (TEXT CONST taskname)
Liefert die Task des angegebenen Namens, falls sie existiert. Der eigene Katalog wird automatisch aktualisiert.

FEHLER : "taskname" gibt es nicht

TASK PROC task (INT CONST channel number)
Liefert den Namen der Task, die an dem angegebenen Kanal hängt.

brother

TASK PROC brother (TASK CONST task)
Liefert den nächsten Bruder von 'task'. Falls kein Bruder existiert, wird 'niltask' geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

father

TASK PROC father
Liefert die eigene Vatertask.

TASK PROC father (TASK CONST task)
Liefert den Vater von 'task'. Existiert kein Vater (z.B. bei UR), wird niltask geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

myself

TASK PROC myself
Liefert eigenen Task – Bezeichner.

son

TASK PROC son (TASK CONST task)
Liefert den ersten Sohn von 'task'. Falls keiner im Katalog vermerkt ist, wird 'niltask' geliefert. Aktualisiert den eigenen Katalog nicht automatisch!

archive

TASK PROC archive
Liefert den internen Taskbezeichner der aktuellen Task mit Namen ARCHIVE. Diese Prozedur dient zum schnellen und bequemen Ansprechen der Archivtask.

printer

TASK PROC printer
Liefert den internen Taskbezeichner der aktuellen Task mit Namen PRINTER. Diese Prozedur dient zum schnellen und bequemen Ansprechen des Druckspoolers.

public

TASK PROC public
Liefert den internen Taskbezeichner der Task PUBLIC.

supervisor

TASK PROC supervisor
Liefert den internen Taskbezeichner des Supervisors.

name

TEXT PROC name (TASK CONST task)
Liefert den Namen von 'task'. Die Task muß noch im System existieren, sonst ist der Name nicht mehr bekannt. Falls die 'task' noch nicht im eigenen Katalog enthalten ist, wird er aktualisiert.

rename myself

PROC rename myself (TEXT CONST neuer name)
Name der eigenen Task wird in 'neuer name' geändert. Wirkt wie Löschung und Wiedereinrichten der Task in Bezug auf alle TASK VAR's die sich auf diese Task beziehen.

FEHLER : Task existiert bereits
Name unzulässig
= > anderen Namen wählen

reserve

PROC reserve (TASK CONST task)
Reservieren einer Task für den ausschließlichen Dialog mit der Task, in der das Kommando gegeben wurde.

PROC reserve (TEXT CONST message, TASK CONST task)
Wie 'reserve (TASK CONST task)' mit Übergabe einer 'message'.

Die reservierte Task muß ein spezieller Manager, (z.B. /"DOS" aus dem Werkzeug MS – DOS – DAT) sein !

4.2.4 Handhabung von Dateien

copy

PROC copy (TEXT CONST quelle, ziel)
Kopiert die Datei 'quelle' in eine neue Datei mit dem Namen 'ziel' in der Benutzer – Task.

FEHLER : "ziel" existiert bereits
"quelle" gibt es nicht
zu viele Dateien

forget

PROC forget (TEXT CONST dateiname)
Löschen einer Datei mit dem Namen 'dateiname' in der Benutzer – Task.

FEHLER : "datei" gibt es nicht

PROC forget (THESAURUS CONST thesaurus)
Löscht die im 'thesaurus' enthaltenen Dateien in der Benutzer – Task.

Im Dialog erfolgt vor dem Löschen einer Datei standardmäßig die Abfrage:

```
   gib kommando :
   forget("einedatei")
"einedatei" löschen(j/n) ?
```

PROC list
Listet alle Dateien der Benutzer – Task mit Namen und Datum des letzten Zugriffs auf dem Terminal auf.

PROC list (TASK CONST task)
Listet alle Dateien der angegebenen 'task' mit Namen und Datum der letzten Änderung auf dem Terminal auf. Die Task muß Manager sein.

PROC list (FILE VAR liste)
Listet alle Dateinamen in die Datei 'liste', die mit 'output'(s. 5.3.5) assoziiert sein muß.

PROC list (FILE VAR liste, TASK CONST manager)
Listet alle Dateien der Task 'manager' mit Namen und Datum der letzten Änderung in die Datei 'liste'.

```
gib kommando :
FILE VAR f:= sequential file (output,"list");list(f,archive)
```

rename

PROC rename (TEXT CONST altername, neuername)
Umbenennen einer Datei von 'altername' in 'neuername'.

FEHLER : "neuername" gibt es bereits
"altername" gibt es nicht

4.2.5 Editor – Prozeduren

edit

PROC edit (TEXT CONST dateiname)
Ruft den Editor mit 'dateiname' auf.

PROC edit
a) Im Monitor:
Ruft den Editor mit den zuletzt verwandten Dateinamen auf.

b) Im Editor:
Der Dateiname wird erfragt.

Für jedes 'edit' gilt:
Wurde 'edit' zum ersten Mal aufgerufen, nimmt das Fenster den gesamten Bildschirm ein. Bei erneutem 'edit' – Aufruf wird ein Fenster nach rechts unten ab der aktuellen Cursor – Position eröffnet.

PROC edit (THESAURUS CONST t)
Editieren aller in dem Thesaurus 't' enthaltenen Dateien nacheinander.

Weitere 'edit – Prozeduren', die z.B. Variation der Fenstergröße etc. zulassen, sind in 5.4.6 beschrieben.

editget

PROC editget (TEXT VAR editsatz)
Ausgabe einer (Kommando)zeile, in der Editorfunktionen zur Verfügung stehen siehe Teil 5.5.1.4.

show

PROC show (TEXT CONST dateiname)
Die Datei wird am Bildschirm gezeigt. Positionierung und Suchen funktionieren wie in 'edit', Aktionen die Änderungen in der Datei bewirken würden, werden nicht angenommen.

PROC show
'show' auf der zuletzt bearbeiteten Datei.

kommando auf taste legen

PROC kommando auf taste legen (TEXT CONST taste, elan programm)
Die Taste 'taste' wird mit dem angegebenen ELAN – Programm belegt. Durch ESC taste wird das Programm direkt ausgeführt.

```
gib kommando :
kommando auf taste legen ("a","fetch (SOME archive,archive)")
```

kommando auf taste

TEXT PROC kommando auf taste (TEXT CONST taste)
Falls 'taste' mit einem ELAN – Programm belegt ist, liefert die Prozedur den Programmtext, andernfalls den leeren Text niltext.

```
gib kommando :
put (kommando auf taste("f"))
```

taste enthaelt kommando

BOOL PROC taste enthaelt kommando (TEXT CONST taste)
Liefert TRUE falls 'taste' mit einem ELAN – Programm belegt ist.

lernsequenz auf taste legen

PROC lernsequenz auf taste legen (TEXT CONST taste, sequenz)
'taste' wird mit der Zeichenfolge 'sequenz' belegt. Durch ESC taste wird die Zeichenfolge an der aktuellen Position ausgegeben.

Als Zeichenfolge sind natürlich auch einzelne Zeichen und EUMEL – Codes zulässig.

Die vom System vorbelegten Tasten sind in 3.4 'Zeichen schreiben' aufgelistet.

```
gib kommando :
lernsequenz auf taste legen ("x","gib kommando :"13""2""2"")
```

lernsequenz auf taste

TEXT PROC lernsequenz auf taste (TEXT CONST taste)
Liefert die auf 'taste' gelegte Zeichenfolge.

std tastenbelegung

PROC std tastenbelegung
Die Standard – Tastenbelegung (s.3.4) wird (wieder) hergestellt.

word wrap

PROC word wrap (BOOL CONST b)
Der automatische Zeilenumbruch wird durch 'word wrap (FALSE)' aus – und durch 'word wrap (TRUE)' eingeschaltet. Wird diese Prozedur während des Editierens aufgerufen, gilt die Einstellung für die aktuelle Textdatei. Wird die Prozedur als Monitor – Kommando gegeben, so gilt die Eingabe als Voreinstellung für neue Dateien.

4.2.6 Dateitransfer

Unter diesem Abschnitt sind diejenigen Prozeduren beschrieben, die der simplen Kommunikation mit Manager – Tasks dienen: Holen oder Senden einer Dateikopie, Löschen in der Manager – Task.

ACHTUNG : Für alle Prozeduren gilt: falls die Manager – Task nicht existiert, wird eine Fehlermeldung erzeugt, existiert eine Task des angegebenen Namens, die aber nicht Managertask ist, so terminieren die Prozeduren nicht!

fetch

PROC fetch (TEXT CONST dateiname, TASK CONST manager)
Kopiert die Datei 'dateiname' aus der Task 'manager'

PROC fetch (THESAURUS CONST th, TASK CONST manager)
Kopiert alle Dateien, deren Namen im Thesaurus th enthalten sind, aus der Task 'manager'.

```
gib kommando :
fetch(ALL(12/"PUBLIC"), 12/"PUBLIC")
```

Mit diesem Kommando werden (in einem EUMEL Netz) alle Dateien der Task 'PUBLIC' des Rechners mit der Stationsnummer 12 in diesem Netz kopiert.

```
gib kommando :
fetch(SOME archive , archive)
```

Bietet den Thesaurus von 'ARCHIVE' an, nach Auswahl werden alle Dateien deren Namen nicht gelöscht wurden, von der Diskette kopiert.

PROC fetch (TEXT CONST dateiname)
Kopiert die Datei 'dateiname' aus der Task 'father'

PROC fetch (THESAURUS CONST th)
Kopiert alle Dateien, deren Namen in 'th' sind aus der Task 'father'.

fetchall

PROC fetchall
entspricht: fetch (ALL father, father)

PROC fetchall (TASK CONST manager)
entspricht: fetch(ALL manager, manager)

save

PROC save (TEXT CONST dateiname, TASK CONST manager)
Kopiert die Datei 'dateiname' in die Task 'manager'

PROC save (THESAURUS CONST th, TASK CONST manager)
Kopiert alle Dateien, deren Namen im Thesaurus th enthalten sind, in die Task 'manager'.

```
gib kommando :
save(all, (12/"PUBLIC"))
```

Mit diesem Kommando werden (in einem EUMEL Netz) alle Dateien der eigenen Task in die Task 'PUBLIC' des Rechners mit der Stationsnummer 12 in diesem Netz kopiert.

```
gib kommando :
save(SOME myself, manager)
```

Bietet den eigenen Thesaurus an, nach Auswahl werden alle Dateien deren Namen nicht gelöscht wurden, zur Task 'manager' kopiert.

PROC save (TEXT CONST dateiname)
Kopiert die Datei 'dateiname' in die Task 'father'

PROC save (THESAURUS CONST th)
Kopiert alle Dateien, deren Namen in 'th' enthalten sind, in die Task 'father'.

PROC save
Kopiert die zuletzt bearbeitete Datei in die Task 'father'

saveall

PROC saveall
entspricht: save (all, father)

PROC saveall (TASK CONST manager)
entspricht: save (ALL myself, manager)

erase

PROC erase (TEXT CONST dateiname, TASK CONST manager)
Löscht die Datei 'dateiname' aus der Task 'manager'

PROC erase (THESAURUS CONST th, TASK CONST manager)
Löscht alle Dateien, deren Namen im Thesaurus th enthalten sind, aus der Task 'manager'.

PROC erase (TEXT CONST dateiname)
Löscht die Datei 'dateiname' aus der Task 'father'

PROC erase (THESAURUS CONST th)
Löscht alle Dateien, deren Namen in 'th' sind, aus der Task 'father'

PROC erase
Löscht die zuletzt bearbeitete Datei aus der Task 'father'

print

Das Kommando 'print' beinhaltet den Auftrag an die Task 'PRINTER' die enthaltene(n) Datei(en) auszudrucken.

Voraussetzung ist natürlich, daß die Druckersoftware ordnungsgemäß benutzt wurde, um 'PRINTER' einzurichten. Siehe dazu Systemhandbuch Teil 6.

PROC print (TEXT CONST dateiname)
Kopiert die Datei 'dateiname' in die Task 'PRINTER'.

PROC print (THESAURUS CONST th)
Kopiert alle Dateien, deren Namen im Thesaurus 'th' enthalten sind, in die Task 'PRINTER'.

PROC print
Kopiert die zuletzt bearbeitete Datei in die Task 'PRINTER'.

4.2.7 Passwortschutz

Der Passwortschutz im EUMEL – System ist in verschiedener Ausprägung möglich. Einfachste Möglichkeit ist der Schutz einer Task durch ein Passwort. Falls diese Task nicht Manager ist, können alle Daten und Programme, die nur in dieser Task zur Verfügung stehen, auch nur vom Besitzer der Task benutzt werden.

Ähnlich kann auch von einer Manager – Task aus der gesamte Zweig unterhalb dieser Task mit einem Passwort geschützt werden: beispielsweise kann es empfehlenswert sein, den Systemzweig komplett zu schützen, indem in SYSUR ein entsprechendes Passwort vereinbart wird.

Ein Umgehen des Passwortschutzes bei Manager – Tasks (durch Einrichten einer Sohn – Task und 'fetchall') wird durch ein 'begin password' verhindert.

Auch einzelne Dateien lassen sich schützen, indem Lese/Schreibpasswörter für den Dateitransfer vereinbart werden.

Generell gilt für die Verwendung von Passworten:

- Passworte, die zu naheliegend gewählt sind (Vorname des Lebenspartners o.ä.) sind meistens sinnlos, falls wirklich Datenschutz bezweckt ist.

- Passworte, die so raffiniert sind, daß sogar ihr Schöpfer sie vergißt, führen zu 100%igem Datenverlust, da die betroffene Task oder Datei nur noch gelöscht werden kann.

- Die Vereinbarung von "–" als Passwort bewirkt, daß die entsprechende Aktion nicht mehr durchgeführt werden kann. Wird z.B. '–' als 'task password' eingegeben, so kann die Task nie wieder an ein Terminal gekoppelt werden.

- Passwörter können geändert werden, indem das entsprechende Kommando noch einmal mit dem neuen Passwort gegeben wird.

begin password

PROC begin password (TEXT CONST passwort)

Auf Supervisor – Ebene wird vor Einrichten einer neuen Task als Sohn der Task in der das 'begin password' gegeben wurde, dieses erfragt.

Das Password vererbt sich auf die hinzukommenden Sohn – Tasks.

```
SYSUR
maintenance :
begin password ("alles dicht")
```

bewirkt:

```
                                        Terminal 2

                EUMEL Version 1.8.1/M


     gib supervisor kommando:
     begin ("sabotage","SYSUR")
           Passwort:

ESC ? --> help
ESC b --> begin("")              ESC h --> halt
ESC c --> continue("")           ESC s --> storage info
ESC q --> break                  ESC t --> task info
```

enter password

PROC enter password (TEXT CONST datei, schreibpass, lesepass)

Hiermit können ausgewählte Dateien einer Manager – Task geschützt werden. Die angegebene Datei wird mit Schreib – und Lesepassword versehen. Die Passwörter werden in der eigenen Task nicht berücksichtigt.

Bei einem lesenden Zugriff (fetch) von irgendeiner Task aus auf die entsprechende Datei in der Manager – Task muß das Lesepasswort, bei schreibendem Zugriff (save/erase) das Schreibpasswort vereinbart sein.

```
maintenance :
enter password ("wichtige datei","sicher","heit")
```

PROC enter password (TEXT CONST password)
Passwort für den Dateitransfer einstellen. Falls zwei verschiedene Passwörter für Lesen und Schreiben vereinbart werden sollen, so sind sie als ein Text durch "/" getrennt einzugeben.

```
gib kommando :
enter password ("sicher/heit")

gib kommando :
save(SOME all)
```

family password

PROC family password (TEXT CONST geheim)
Einstellen eines Passworts für den Zweig des Systems , der unterhalb der (Manager) Task liegt, in der das 'family password' eingegeben wurde. Dabei erhalten alle Tasks, die kein Password oder dasselbe wie diese Manager – Task haben, das 'family password'. Tasks in dem Zweig, die ein eigenes anderes besitzen, behalten dieses.

```
PUBLIC

  Task1 ""

  Task2 family password("fingerweg")
     Task21 geheim
     Task22 ""

  Task3 ""
     Task31 ""
```

bewirkt:

```
PUBLIC

  Task1 ""

  Task2 fingerweg
     Task21 geheim
     Task22 fingerweg

  Task3 ""
     Task31 ""
```

task password

PROC task password (TEXT CONST geheim)
Einstellen eines Passworts für die Task in der es gegeben wird. Ist eine Task mit einem Passwort geschützt, so wird durch den Supervisor nach dem 'continue' – Kommando das Passwort angefragt (Entsprechend dem 'begin password'). Nur nach Eingabe des richtigen Passworts gelangt man in die gewünschte Task. Das Passwort kann durch nochmaligen Aufruf von 'task password' geändert werden, z.B. wenn es in regelmäßigen Abständen geändert werden muß, um personenbezogene Daten zu schützen.

4.2.8 Das Archiv

Mit dem Terminus 'Archiv' wird beim EUMEL – System ein Diskettenlaufwerk bezeichnet, das nur Datensicherungsaufgaben dient. Falls ein Rechner eins von zwei vorhandenen Diskettenlaufwerk als Arbeitsspeicher benutzt, so wird dieses als Hintergrund bezeichnet. Falls Sie einen derartigen Rechner benutzen, können Sie der Installationsanleitung entnehmen, welches Laufwerk welcher Aufgabe zugeordnet ist.

Das Archiv übernimmt im EUMEL – System die Verwaltung der langfristigen Datenhaltung. Das Archiv sollen Sie benutzen, um:

- Sicherungskopien wichtiger Dateien außerhalb des Rechners zu besitzen;
- nicht benötigte Dateien außerhalb einer Task zu halten (Speicherplatzersparnis!);
- Dateien auf andere Rechner zu übertragen.

Das Archiv wird im EUMEL – System durch die Task 'ARCHIVE', die das Diskettenlaufwerk des Rechners verwaltet, realisiert.

– reservieren	archive
– freigeben	release
– löschen	clear , format
– prüfen	check

archive

PROC archive (TEXT CONST archivname)
Reservierung der Task ARCHIVE für den exklusiven Dialog mit der aufrufenden Task. 'archivname' wird bei allen folgenden Archivoperationen mit dem der Diskette zugewiesenen (und hoffentlich auf dem Aufkleber vermerkten) Namen abgeglichen.

release

PROC release (TASK CONST archive)
Nach diesem Kommando kann die Task 'ARCHIVE' mit ihren Leistungen von einer anderen Task in Anspruch genommen werden. Falls dieses Kommando nicht gegeben wird, aber seit 5 Minuten kein Dialog mit 'archive' stattfand, kann eine andere Task durch die Anforderung 'archive("diskettenname")' das Archiv reservieren. Durch diese Maßnahme wird verhindert, daß ein vergeßlicher Benutzer bei einem System mit mehreren Benutzern das Archiv blockiert.

clear

PROC clear (TASK CONST archive)
Löschen des Disketten – Inhaltsverzeichnisses und Zuweisung des in der Reservierung eingegebenen Namens.

```
gib kommando :
archive("name"); clear (archive)
```

Durch die Ausführung des Kommandos erhält die eingelegte Diskette den in der Reservierung angegebenen Namen. **Das Inhaltsverzeichnis, das sich auf der Diskette befindet, wird gelöscht. Damit sind die Daten, die sich eventuell auf dieser Diskette befanden, nicht mehr auffindbar.** Die Diskette entspricht einer neu formatierten Diskette[1)].

Man kann also eine beschriebene Diskette nicht umbenennen, ohne die darauf befindlichen Daten zu löschen.

Eine Neuformatierung ist demnach bei Wiederverwendung der Diskette nicht notwendig.

[1)] Das Kommando 'format' enthält implizit 'clear'.

format

PROC format (TASK CONST archive)
Formatieren einer Diskette. Vor der erstmaligen Benutzung einer Archivdiskette muß diese formatiert, d.h. in Spuren und Sektoren für die Positionierung des Schreib-/Lesekopfes des Diskettenlaufwerks eingeteilt werden, um überhaupt ein Beschreiben der Diskette zu ermöglichen. Die Einteilung ist geräteabhängig, häufige Formate sind:

40 Spuren zu je 9 Sektoren (360 K)
80 Spuren zu je 9 Sektoren (720 K).

Die Erstbenutzung einer Archivdiskette erfordert nach der Reservierung des Archivs das Kommando:

```
gib kommando :
archive("diskname");

gib kommando :
format (archive);
```

Erst nach einer Kontrollabfrage:

```
gib kommando:
format (archive)

Archiv "diskname" formatieren ? (j/n)
```

wird tatsächlich formatiert und die Diskette steht mit dem Namen "diskname" für Archivoperationen zur Verfügung.

PROC format (INT CONST code, TASK CONST archive)
Bei einigen Rechnern ist es möglich, die Formatierung zu variieren. Falls beim Formatieren auf einem solchen Rechner ein anderes als das Standardformat erzeugt werden soll, so ist die Codierung des gewünschten Formats mitanzugeben.

Beispiel: Für ein Gerät mit 5,25 Zoll Disketten wäre z.B. einstellbar:
code 0 : Standardformat
code 1 : 2D , 40 Spuren , 9 Sektoren
code 2 : 2D , 80 Spuren , 9 Sektoren
code 3 : HD , 80 Spuren ,15 Sektoren

'format (archive)' erzeugt ebenso wie 'format (0,archive)' eine standardformatierte Diskette, 'format (3,archive)' erzeugt eine High Density Formatierung (HD Floppy benutzen!).

ACHTUNG: Wird eine bereits beschriebene Diskette noch einmal formatiert, so sind alle Daten, die auf der Diskette waren, verloren.

Die Umformatierung einer Diskette (z.B. von 720K auf 360K) auf unterschiedlichen Laufwerken kann zu Problemen führen.

check

PROC check (TEXT CONST dateiname, TASK CONST task)
Überprüft, ob die Datei 'dateiname' auf dem Archiv lesbar ist.

PROC check (THESAURUS CONST t, TASK CONST task)
Überprüft, ob die in dem Thesaurus 't' enthaltenen Dateien auf dem Archiv lesbar sind.

Mit diesem Kommando kann nach dem Beschreiben einer Diskette überprüft werden, ob die Datei(en) lesbar sind. Hierdurch können also verschmutzte oder beschädigte Disketten erkannt werden.

```
gib kommando :
save (all , archive)

gib kommando :
check (ALL archive, archive)
```

Beispiel:

```
gib kommando :
archive ("neu")

gib kommando :
format (archive)
```

liefert zunächst die Kontollfrage:

```
gib kommando :
format (archive)

Archiv "neu" formatieren ? (j/n)
```

Nach Eingabe 'j'

```
gib kommando :
saveall (archive)

gib kommando :
archive("alt") (* nächste Diskette *)

gib kommando :
fetch(SOME archive ,archive)
```

Der Thesaurus des Archivs wird angezeigt:

```
                    all 160 K belegt von 220 K
01.02.87   25 K  "handbuch teil 1"
01.03.87   23 K  "handbuch teil 2"
01.04.87   20 K  "handbuch teil 3"
01.05.87   32 K  "handbuch teil 4"
```

Zum Abschluß Archiv freigeben!

```
gib kommando :
release(archive)
```

Fehlermeldungen des Archivs

Versucht man, eine Datei vom Archiv zu holen, kann es vorkommen, daß das Archiv – System

```
gib kommando :
fetch ("datei", archive)
Lese-Fehler (Archiv)
```

meldet und den Lese – Vorgang abbricht. Dies kann auftreten, wenn die Floppy beschädigt oder aus anderen Gründen nicht lesbar ist (z.B. nicht justierte Disketten – Geräte). In einem solchen Fall vermerkt das Archiv – System intern, daß die Datei nicht korrekt gelesen werden kann. Das sieht man z.B. bei 'list (archive)'. Dort ist der betreffende Datei – Name mit dem Zusatz 'mit Lese – Fehler' gekennzeichnet. Um diese Datei trotzdem zu lesen, muß man sie unter ihrem Dateinamen mit dem Zusatz 'mit Lese – Fehler' lesen.

```
gib kommando:
fetch ("datei mit Lese-Fehler", archive)
```

Die Datei wird in diesem Fall trotz Lese – Fehler (Informationsverlust!) vom Archiv gelesen.

Weitere Fehlermeldungen des Archivs:

FEHLER : Lesen unmöglich (Archiv)
Die Archiv – Diskette ist nicht eingelegt oder die Tür des Laufwerks ist nicht geschlossen.
= > Diskette einlegen bzw. Tür schließen.

FEHLER : Schreiben unmöglich (Archiv)
Die Diskette ist schreibgeschützt.
= > falls wirklich gewünscht, Schreibschutz entfernen.

FEHLER : Archiv nicht angemeldet
Das Archiv wurde nicht angemeldet
= > 'archive ("name")' geben.

FEHLER : Lese – Fehler (Archiv)
Siehe Lesen unmöglich

FEHLER : Schreibfehler (Archiv)
Die Diskette kann nicht (mehr) beschrieben werden.
= > Andere Diskette verwenden.

FEHLER : Speicherengpass
Im System ist nicht mehr genügend Platz, um eine Datei vom Archiv zu laden.
= > ggf. Dateien löschen.

FEHLER : RERUN bei Archiv – Zugriff Das System wurde bei einer Archiv – Operation durch Ausschalten bzw. Reset unterbrochen.

FEHLER : "dateiname" gibt es nicht
Die Datei "dateiname" gibt es nicht auf dem Archiv.
= > mit 'list(archive)' Archiv prüfen.

FEHLER : Archiv heißt ...
Die eingelegte Diskette hat einen anderen als den eingegebenen Archivnamen.
= > Kommando 'archive' mit korrektem Namen geben.

FEHLER : Archiv wird von Task ... benutzt
Das Archiv wurde von einem anderen Benutzer reserviert.
= > Abwarten.

FEHLER : "dateiname" kann nicht geschrieben werden (Archiv voll)
Die Datei ist zu groß für die eingelegte Diskette.
= > Andere Diskette für diese Datei nehmen.

FEHLER : Archiv inkonsistent
Die eingelegte Diskette hat nicht die Struktur einer Archiv – Diskette.
= > 'format (archive)' vergessen.

FEHLER : save/erase wegen Lese – Fehler verboten
Bei Archiven mit Lese – Fehler sind Schreiboperationen verboten, weil ein Erfolg nicht garantiert werden kann.

TEIL 5: Programmierung

5.1 Der ELAN – Compiler

Der ELAN – Compiler des EUMEL – Systems dient zweierlei Aufgaben: zum einen der Übersetzung von ELAN – Programmen, zum anderen der Verwaltung der taskeigenen Modulbibliothek.

Diese Moduln, in ELAN Pakete (siehe 2.4.3.4ff.) genannt, stellen als vorübersetzte, und damit abrufbereite[1] Prozeduren den Kommandovorrat einer Task dar.

Der Codebereich einer Task liegt in ihrem Standarddatenraum (ds4). Die Größe dieses Codebereiches beträgt 256K. Der Inhalt besteht zunächst aus den von der Vatertask ererbten (durch Kopie des ds4 dieser Task) Moduln, im weiteren allen in dieser Task neu hinzu insertierten Packeten.

ACHTUNG: Durch ständiges Neuinsertieren eines Packets kann der Codebereich der betroffenen Task zum Überlaufen gebracht werden!

Jedes Kommando im EUMEL – System ist der Aufruf einer, in der Schnittstelle eines bereits insertierten Packetes stehenden, Prozedur.

Kommandos für den ELAN – Compiler:

– Übersetzen	:	do , insert , run , runagain
– Protokollieren	:	check , checkon/off , prot , protoff , warnings on/off

1) Die von anderen Systemen her gewohnten Phasen 'Binden' und 'Laden' sind durch das EUMEL – ELAN – Compiler – Konzept unnötig.

do

PROC do (TEXT CONST program)
Übersetzen und Ausführen von 'program' von einem Programm aus. 'program' muß ein ausführbares ELAN Programm sein.

```
Beispiel
PACKET reo DEFINES reorganize all:

PROC reorganize all(THESAURUS CONST thes):
do (PROC (TEXT CONST) reorganize ,thes)
   (* Die Prozedur 'reorganize' (siehe 5-52), die einen*)
   (* Dateinamen als Parameter verlangt, wird auf alle *)
   (* Dateien des Thesaurus 'thes' angewandt.          *)
END PROC reorganize all;
END PACKET reo;
```

insert

PROC insert (TEXT CONST dateiname)
Insertieren eines oder mehrerer PACKETs aus der Datei 'dateiname'. Der Programmtext muß sich in einer Datei befinden.

PROC insert
Insertieren eines oder mehrerer PACKETs. Der Dateiname ist der zuletzt benutzte Dateiname.

PROC insert (THESAURUS CONST t)
Insertieren aller PACKETs, die in den Dateien des Thesaurus 't' enthalten sind.

run

PROC run
Übersetzen und Ausführen eines ELAN – Programms. Der Programmtext muß sich in einer Datei befinden. Der Dateiname ist der zuletzt benutzte Dateiname.

PROC run (TEXT CONST dateiname)
Wie oben. Der Programmtext wird aus der Datei mit dem Namen 'dateiname' geholt.

runagain

PROC runagain
Nochmaliges Ausführen des zuletzt mit 'run' übersetzten ELAN – Programms. Wurde in der letzten Übersetzung ein Fehler gefunden, erfolgt die Meldung:

FEHLER : "run again nicht möglich"

check

BOOL PROC check
Informationsprozedur, die TRUE liefert, wenn 'check' eingeschaltet ist.

PROC check on
Einschalten der Generierung von Zeilennummern durch den ELAN – Compiler. Der bei der Übersetzung erzeugte Code wird ca. 25% umfangreicher!
Voreinstellung im 'PUBLIC' – Zweig: 'check on'.

PROC check off
Ausschalten der Generierung von Zeilennummern durch den ELAN – Compiler.
Voreinstellung im 'SYSUR' – Zweig: 'check off.

prot

BOOL PROC prot
Informationsprozedur, die TRUE liefert, gdw. 'prot' eingeschaltet ist.

PROC prot (TEXT CONST dateiname)
Einschalten des Compilerlistings auf dem Bildschirm. Das Listing wird gleichzeitig in die Datei 'dateiname' geschrieben.

PROC prot off
Ausschalten des Listings.

warnings

BOOL PROC warnings
Informationsprozedur, die TRUE liefert gdw. 'warnings' eingeschaltet ist.

PROC warnings on
Warnungen werden wie Fehlermeldungen ins Notizbuch ausgegeben.

PROC warnings off
Warnungen werden nicht mit in das Notizbuch ausgegeben.

5.1.1 Fehlermeldungen des ELAN – Compilers

erfolgen stets in der Form:

COMPILER ERROR: <zahl>

wobei <zahl> folgende Werte annehmen kann:

<zahl> Bedeutung und eventuelle Abhilfe:

101 Überlauf der Namenstabelle
Die Anzahl der Namen aller sichtbaren Pakete ist zu groß oder es wurden die Anführungstriche eines TEXT – Denoters vergessen.
=> Keine Abhilfe.

102 Überlauf der Symboltabelle
Die Anzahl der deklarierten Objekte ist zu groß.
=> Programm in Pakete unterteilen.

103 Überlauf des Zwischencodebereiches
=> Programm in Pakete unterteilen.

104 Überlauf der Permanenttabelle
Zu viele Pakete insertiert.
=> Keine (neue Task beginnen).

106 Paketdatenadresse zu groß
Im Paket wird zuviel Platz (> 64K) von globalen Datenobjekten und Denotern eingenommen.
=> Keine Abhilfe.

107 Lokale Datenadresse zu groß
Im Paket wird zuviel Platz (> 32K) von lokalen Datenobjekten belegt.
=> Keine Abhilfe.

204 Überlauf des Compilerstack
=> Keine Abhilfe.

301 Modulnummern – Überlauf
Zu viele sichtbare Pakete, Prozeduren und Operatoren (> 2048).
=> Keine Abhilfe.

303
siehe 304

304 Zu viele Ansprungadressen
In dem gerade übersetzten Modul (Prozedur, Operator oder Paketrumpf) werden vom Compiler zu viele Marken benötigt (mehr als 2000). Marken werden z.B. für die Codegenerierung von Auswahl (IF ...) und Wiederholung (REP ...) gebraucht. Insbesondere bei SELECT – Anweisungen werden 'casemax – casemin + 2' Marken benötigt, wobei 'casemax' der INT – Wert des maximalen, 'casemin' der des minimalen CASE – Wertes ist. Dieser Fehler ist somit fast immer auf zu viele und/oder zu weit gespannte SELECT – Anweisungen zurückzuführen.
=> SELECT – Anweisungen über mehrere Prozeduren verteilen oder Spannweiten verringern.

305 Codeüberlauf
Der insgesamt erzeugte sichtbare Code ist zu umfangreich (> 256K).
=> Keine Abhilfe.

306 Paketdatenadresse zu groß
Insgesamt zu viele Datenobjekte in den Paketen (> 128K).
=> Keine Abhilfe.

307 Temporäre Datenadresse zu groß
Zu viele (lokale) Datenobjekte in einer Prozedur (> 32K).
=> Prozedur in mehrere unterteilen, so daß die Datenobjekte sich über mehrere Prozeduren verteilen.

308 Modulcode – Überlauf
Ein Modul (Prozedur, Operator oder Paket – Initialisierungsteil) ist zu groß (> 7.5 KB Code).
=> In mehrere Prozeduren oder Pakete zerlegen.

309 Zuviele Paketdaten
(Insgesamt mehr als 128K Paketdaten)
=> Keine Abhilfe

5.2 Standardtypen

5.2.1 Bool

Der Wertebereich für Datenobjekte vom Typ BOOL besteht aus den Werten TRUE und FALSE.

AND

BOOL OP AND (BOOL CONST a, b)
Logisches UND, liefert TRUE gdw. a und b TRUE sind.

CAND

BOOL OP CAND
Bedingtes logisches UND, entspricht: 'IF a THEN b ELSE false FI'. Der zweite Operand wird nicht ausgewertet, falls er für das Ergebnis nicht relevant ist.

COR

BOOL OP COR
Bedingtes logisches ODER, entspricht: 'IF a THEN true ELSE b FI'. Der zweite Operand wird nicht ausgewertet, falls er für das Ergebnis nicht relevant ist.

false

BOOL CONST false

NOT

BOOL OP NOT (BOOL CONST a)
Logische Negation.

OR

BOOL OP OR (BOOL CONST a, b)
Logisches ODER, liefert TRUE gdw. a und/oder b TRUE ist.

true

BOOL CONST true

XOR

BOOL OP XOR (BOOL CONST a, b)
Exklusives ODER, liefert TRUE gdw. entweder a oder b TRUE ist.

5.2.2 Integer – Arithmetik

Ein Datenobjekt vom Typ INT belegt im Speicher 2 Bytes. Zulässige INT – Werte sind die ganzen Zahlen von –32768 bis +32767 einschließlich.

Falls größere ganze Zahlen benötigt werden, muß das Packet 'LONGINT', welches sich auf dem Archive 'std.zusatz' befindet, nachinsertiert werden (siehe 6.1.2).

Operationen für Integers:

– Vergleich		= , <> , < , <= , > , >=
– Verknüpfung		+ , – , * , ** , DECR , DIV , INCR
– Sonstiges	:	abs , ABS , initialize random , max , maxint , min , minint , MOD , random , sign , SIGN , text

INT OP := (INT VAR a, INT CONST b)
Zuweisung.

BOOL OP = (INT CONST a, b)
Vergleich.

BOOL OP <> (INT CONST a, b)
Vergleich auf Ungleichheit.

BOOL OP < (INT CONST a, b)
Vergleich auf kleiner.

BOOL OP <= (INT CONST a, b)
Vergleich auf kleiner gleich.

BOOL OP > (INT CONST a, b)
Vergleich auf größer.

BOOL OP >= (INT CONST a, b)
Vergleich auf größer gleich.

INT OP + (INT CONST a)
Monadischer Operator (Vorzeichen, ohne Wirkung).

INT OP + (INT CONST a, b)
Addition.

INT OP – (INT CONST a)
Vorzeichen – Umkehrung.

INT OP – (INT CONST a, b)
Subtraktion.

INT OP * (INT CONST a, b)
Multiplikation.

INT OP ** (INT CONST arg, exp)
Exponentiation mit 'exp' > = 0

DECR

OP DECR (INT VAR links, INT CONST rechts)
Wirkt wie links := links – rechts

DIV

INT OP DIV (INT CONST a, b)
INT – Division.

FEHLER :
– DIV durch 0

INCR

OP INCR (INT VAR links, INT CONST rechts)
Wirkt wie links := links + rechts

abs

INT PROC abs (INT CONST argument)
Absolutbetrag eines INT – Wertes.

INT OP ABS (INT CONST argument)
Absolutbetrag eines INT – Wertes.

initialize random

PROC initialize random (INT CONST wert)
Initialisieren der 'random' – Prozedur, um nicht reproduzierbare Zufallszahlen zu bekommen. Diese 'initialize random' – Prozedur gilt für den "INT – Random Generator".

max

INT PROC max (INT CONST links, rechts)
Liefert den Größten der beiden INT – Werte.

maxint

INT CONST maxint
Größter INT – Wert im EUMEL – System (32 767).

min

INT PROC min (INT CONST links, rechts)
Liefert den Kleinsten der beiden INT – Werte.

```
min ( 3.0, 2.0) ==>  2.0
min (-2.0, 3.0) ==> -2.0
```

minint

INT CONST minint
Kleinster INT – Wert im EUMEL – System (– 32768).

MOD

INT OP MOD (INT CONST links, rechts)
Liefert den Rest einer INT – Division.

```
 3 MOD 2 ==> 1
-3 MOD 2 ==> 1
```

FEHLER :
– DIV durch 0

random

INT PROC random (INT CONST lower bound, upper bound)
Pseudo – Zufallszahlen – Generator im Intervall 'upper bound' und 'lower bound' einschließlich. Es handelt sich hier um den "INT Random Generator".

real

REAL PROC real (INT CONST a)
Konvertierungsprozedur.

sign

INT PROC sign (INT CONST argument)
Feststellen des Vorzeichens eines INT – Wertes. Folgende Werte werden geliefert:

```
argument > 0      ==>  1
argument = 0      ==>  0
argument < 0      ==> -1
```

INT OP SIGN (INT CONST argument)
Feststellen des Vorzeichens eines INT – Wertes.

text

TEXT PROC text (INT CONST zahl)
Konvertierung des INT Wertes 'zahl' in den kürzest möglichen Text. Das Vorzeichen bleibt erhalten.

TEXT PROC text (INT CONST zahl, länge)
Konvertierung des INT Wertes 'zahl' in einen Text der Länge 'länge'. Das Vorzeichen bleibt erhalten. Falls der Text kürzer als 'länge' ist, wird er links (vorne) mit Leerzeichen aufgefüllt, falls er länger ist wird 'länge' mal "*" ausgegeben.

```
out ("X:"); out(text(12345,7)) ; line;
out ("Y:"); out(text(12345,3)) ;
    (* ergibt *)
X:  12345
Y:***
```

5.2.3 Real – Arithmetik

Für den Datentyp REAL gibt es außer den üblichen Verknüpfungs – und Vergleichsoperationen noch eine Anzahl mathematischer Prozeduren und Operationen. Teilweise stehen diese in mehr als einer Version zur Verfügung.

Jedes Datenobjekt vom Typ REAL belegt im Speicher 8 Byte.

REALs haben eine 13 – stellige Mantisse, die im Rechner dezimal geführt wird. (Das heißt, bei Konversionen zwischen interner und TEXT – Darstellung treten keine Rundungsfehler auf.) Der Wertebereich wird durch folgende Eckwerte abgegrenzt:

9.9999999999999e + 126	größter REAL – Wert
0.000000000001	kleinster positiver REAL – Wert mit x + 1.0 > 1.0
9.9999999999999e – 126	kleinster positiver REAL – Wert > 0.0
– 9.9999999999999e – 126	größter negativer REAL – Wert
– 9.9999999999999e + 126	kleinster REAL – Wert

– Vergleiche : = , <> , < , <= , > , >=

– Verknüpfungen : + , – , * , / , ** , DECR , INCR

– Diverse : abs , arctan , arctand , cos , cosd , decimal exponent , e , exp , floor , frac , initialize random , int , ln , log2 , log10 , max , maxreal , min , MOD , pi , random , round , sign , SIGN , sin , sind , smallreal , sqrt , tan , tand , text

REAL OP := (REAL VAR a, REAL CONST b)
Zuweisung.

BOOL OP = (REAL CONST a, b)
Vergleich.

BOOL OP <> (REAL CONST a, b)
Vergleich auf Ungleichheit.

BOOL OP < (REAL CONST a, b)
Vergleich auf kleiner.

BOOL OP <= (REAL CONST a, b)
Vergleich auf kleiner gleich.

BOOL OP > (REAL CONST a, b)
Vergleich auf größer.

BOOL OP >= (REAL CONST a, b)
Vergleich auf größer gleich.

REAL OP + (REAL CONST a)
Monadischer Operator (Vorzeichen, ohne Wirkung).

REAL OP + (REAL CONST a, b)
Addition.

REAL OP – (REAL CONST a)
Vorzeichen – Umkehrung.

REAL OP – (REAL CONST a, b)
Subtraktion.

REAL OP * (REAL CONST a, b)
Multiplikation.

REAL OP / (REAL CONST a, b)
Division.

FEHLER :
– Division durch 0

**

REAL OP ** (REAL CONST arg, exp)
Exponentiation.

REAL OP ** (REAL CONST arg, INT CONST exp)
Exponentiation.

DECR

OP DECR (REAL VAR links, REAL CONST rechts)
Wirkt wie links := links – rechts

INCR

OP INCR (REAL VAR links, REAL CONST rechts)
Wirkt wie links := links + rechts

abs

REAL PROC abs (REAL CONST wert)
Absolutbetrag eines REAL – Wertes.

REAL OP ABS (REAL CONST wert)
Absolutbetrag eines REAL – Wertes.

arctan

REAL PROC arctan (REAL CONST x)
Arcus Tangens – Funktion. Liefert einen Wert in Radiant.

arctand

REAL PROC arctand (REAL CONST x)
Arcus Tangens – Funktion. Liefert einen Wert in Grad.

cos

REAL PROC cos (REAL CONST x)
Cosinus – Funktion. 'x' muß in Radiant angegeben werden.

cosd

REAL PROC cosd (REAL CONST x)
Cosinus – Funktion. 'x' muß in Winkelgrad angegeben werden.

decimal exponent

INT PROC decimal exponent (REAL CONST mantisse)
Liefert aus einem REAL – Wert den dezimalen Exponenten als INT – Wert.

e

REAL PROC e
Eulersche Zahl (2.718282).

exp

REAL PROC exp (REAL CONST z)
Exponentialfunktion.

floor

REAL PROC floor (REAL CONST real)
Schneidet die Nachkommastellen des REAL – Wertes 'real' ab.

frac

REAL PROC frac (REAL CONST z)
Liefert die Stellen eines REAL – Wertes hinter dem Dezimalpunkt.

initialize random

PROC initialize random (REAL CONST z)
Initialisieren der 'random' – Prozedur mit verschiedenen Werten für 'z', um nicht reproduzierbare Zufallszahlen zu bekommen. Diese Prozedur gilt für den 'REAL – Random Generator'.

int

INT PROC int (REAL CONST a)
Konvertierungsprozedur. Die Nachkommastellen werden abgeschnitten.
Bsp: int (3.9) => 3

ln

REAL PROC ln (REAL CONST x)
Natürlicher Logarithmus.

FEHLER :
– ln mit negativer Zahl
  Nur echt positive Argumente sind zulässig.

log2

REAL PROC log2 (REAL CONST z)
Logarithmus zur Basis 2.

FEHLER :
– log2 mit negativer zahl
  Nur echt positive Argumente sind zulässig.

log10

REAL PROC log10 (REAL CONST x)
Logarithmus zur Basis 10.

FEHLER :
– log10 mit negativer zahl
  Nur echt positive Argumente sind zulässig.

max

REAL PROC max (REAL CONST links, rechts)
Liefert den Größten der beiden REAL – Werte.

maxreal

REAL CONST maxreal
Größter REAL – Wert im EUMEL – System (9.999999999999e126).

min

REAL PROC min (REAL CONST links, rechts)
Liefert den Kleinsten der beiden REAL – Werte.

MOD

REAL OP MOD (REAL CONST links, rechts)
Modulo – Funktion für REALs (liefert den Rest). Beispiele:

```
5.0 MOD 2.0 ==> 1.0
4.5 MOD 4.0 ==> 0.5
```

pi

REAL CONST pi
Die Zahl pi (3.141593).

random

REAL PROC random
Pseudo – Zufallszahlen – Generator im Intervall 0 und 1. Es handelt sich hier um den "REAL Random Generator".

round

REAL PROC round (REAL CONST real, INT CONST digits)
Runden eines REAL – Wertes auf 'digits' Stellen. Für positive Werte wird auf Nachkommastellen gerundet. Beispiel:

```
round (3.14159, 3)
```

liefert '3.142'. Für negative 'digits' – Werte wird auf Vorkommastellen gerundet.

```
round (123.456, -2)
```

liefert '100.0'. Abweichung vom Standard: Es wird mit 'digits' – Ziffern gerundet.

sign

INT PROC sign (REAL CONST argument)
Feststellen des Vorzeichens eines REAL – Wertes.

INT OP SIGN (REAL CONST argument)
Feststellen des Vorzeichens eines REAL – Wertes.

sin

REAL PROC sin (REAL CONST x)
Sinus – Funktion. 'x' muß in Radiant (Bogenmaß) angegeben werden.

sind

REAL PROC sind (REAL CONST x)
Sinus – Funktion. 'x' muß im Winkelgrad angegeben werden.

smallreal

REAL PROC smallreal
Kleinster darstellbarer REAL – Wert im EUMEL – System für den

```
1.0 - smallreal <> 1.0
1.0 + smallreal <> 1.0
```

gilt (1.0E – 12).

sqrt

REAL PROC sqrt (REAL CONST z)
Wurzel – Funktion.

FEHLER :
– sqrt von negativer Zahl
  Das Argument muß größer gleich 0.0 sein.

tan

REAL PROC tan (REAL CONST x)
Tangens – Funktion. 'x' muß in Radiant angegeben werden.

tand

REAL PROC tand (REAL CONST x)
Tangens – Funktion. 'x' muß in Winkelgrad angegeben werden.

text

TEXT PROC text (REAL CONST real)
Konvertierung eines REAL – Wertes in einen TEXT. Ggf. wird der TEXT in Exponenten – Darstellung geliefert.

TEXT PROC text (REAL CONST real, laenge)
Konvertierung eines REAL – Wertes in einen TEXT. Der TEXT wird in Exponenten – Darstellung geliefert. Um diese Darstellung zu ermöglichen ist der Wert 'laenge' größer oder gleich 8 anzugeben.

TEXT PROC text (REAL CONST real, INT CONST laenge, fracs)
Konvertierung eines REAL – Wertes in einen TEXT. Dabei gibt 'laenge' die Länge des Resultats einschließlich des Dezimalpunktes und 'fracs' die Anzahl der Dezimalstellen an. Kann der REAL – Wert nicht wie gewünscht dargestellt werden, wird

```
laenge * "*"
```

geliefert.

5.2.4 Text

Jedes Datenobjekt vom Typ TEXT besteht aus einem festen Teil von 16 Bytes und möglicherweise aus einem flexiblen Teil auf dem *Heap*. Im festen Teil werden Texte bis zur Länge von 13 Zeichen untergebracht. Wenn eine TEXT – Variable einen Wert mit mehr als 13 Zeichen Länge annimmt, werden alle Zeichen auf dem Heap untergebracht. Genauer ergibt sich folgendes Bild:

kurzer Text (Länge < = 13):

Heap – Link	2 Bytes
Textlänge	1 Byte
Text	13 Bytes

langer Text (Länge > 13):

Heap – Link	2 Bytes
255	1 Byte
Länge	2 Bytes
ungenutzt	11 Bytes

Wenn eine Variable einmal Platz auf dem Heap bekommen hat, behält sie diesen vorbeugend auch dann, wenn sie wieder einen kurzen Text als Wert erhält. So muß wahrscheinlich kein neuer Platz auf dem Heap zugewiesen werden, wenn sie wieder länger wird. Das gilt allerdings nur bis zur nächsten Garbage Collection auf den TEXT – Heap, denn dabei werden alle Heap – Container minimal gemacht bzw. gelöscht, wenn sie nicht mehr benötigt werden. Der Platz auf dem Heap wird in Vielfachen von 16 Bytes vergeben. In Fremddatenräumen wird in jedem Container neben dem eigentlichen Text auch die Containerlänge untergebracht.

Beispiele:	TEXT – Länge	Speicherbedarf (Byte)
	0	16
	13	16
	14	32
	15	48
	30	48
	31	64
	46	64
	47	80
	62	80

Die Heapgröße eines Fremddatenraums berechnet sich als:

1024 * 1024 = 1048056 – stat Bytes

'stat' ist dabei die statische Größe der Datenstruktur, die dem Datenraum aufgeprägt wurde. Bei einem BOUND ROW 1000 TEXT ergibt sich also eine Heapgröße von

1048056 – (1000 * 16) = 1032056 Bytes.

heap size

INT PROC heap size
Informationsprozedur für die Größe (in KB) des TEXT – Heaps.

TEXT – Operationen:

– Vergleich		= , <> , < , <= , > , >= LEXEQUAL , LEXGREATER , LEXGREATEREQUAL
– Verkettung		+ , * , CAT
– Veränderung	:	change , change all , code , compress , delete char , insert char , length , LENGTH , max text length , pos , real , replace , SUB , subtext , text

Der EUMEL – Zeichensatz

Das EUMEL System definiert einen Zeichensatz, der gewährleistet, daß gleiche Textzeichen auf allen Maschinen gleich codiert werden.

Die interne Darstellung wird durch die folgende EUMEL – Codetabelle beschrieben. Der Zeichensatz beruht auf dem ASCII – Zeichensatz mit Erweiterungen. Der in der Tabelle freie Bereich (z.B code(127) bis code(213)) ist nicht einheitlich verfügbar und wird deshalb nicht beschrieben. Die Codierung bildet mithin auch Grundlage für Vergleiche und Sortierungen.

Die Korrekte Darstellung dieser Zeichen auf Bildschirm, Drucker etc. setzt natürlich eine korrekte Konfiguration der Geräte voraus. Die Anpassung eines Geräts an diesen Zeichensatz ist im EUMEL – Systemhandbuch in Teil 2 beschrieben.

	0	1	2	3	4	5	6	7	8	9
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9		;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~			
13										
.										
.										
20										
21					Ä	Ö	Ü	ä	ö	ü
22	k	-	#	SP						
23										
24										
25		ß								

TEXT OP := (TEXT VAR a, TEXT CONST b)
Zuweisung.

BOOL OP = (TEXT CONST links, rechts)
Vergleich von zwei Texten auf Gleichheit (Texte mit ungleichen Längen sind immer ungleich).

BOOL OP <> (TEXT CONST links, rechts)
Vergleich von zwei Texten auf Ungleichheit (Texte mit ungleichen Längen sind stets ungleich).

BOOL OP < (TEXT CONST links, rechts)
Vergleich zweier Texte auf kleiner ('links' kommt lexikographisch vor 'rechts').

BOOL OP <= (TEXT CONST links, rechts)
Vergleich von zwei Texten auf kleiner gleich ('links' kommt lexikographisch vor oder ist gleich 'rechts').

BOOL OP > (TEXT CONST links, rechts)
Vergleich zweier Texte auf größer ('links' kommt lexikographisch nach 'rechts').

BOOL OP >= (TEXT CONST links, rechts)
Vergleich zweier Texte auf größer gleich ('links' kommt lexikographisch nach oder ist gleich 'rechts').

LEXEQUAL

BOOL OP LEXEQUAL (TEXT CONST links, rechts)
Prüfung auf lexikalische Gleichheit.

LEXGREATER

BOOL OP LEXGREATER (TEXT CONST links, rechts)
Prüfung ob der Text 'links' lexikalisch größer als 'rechts' ist.

LEXGREATEREQUAL

BOOL OP LEXGREATEREQUAL (TEXT CONST links, rechts)
Prüfung ob der Text 'links' lexikalisch größer oder gleich dem Text 'rechts' ist.

Die drei Operatoren prüfen nach folgenden Regeln:

- Buchstaben haben die aufsteigende Reihenfolge 'A' bis 'Z'. Dabei werden kleine und große Buchstaben gleich behandelt.

- Umlaute werden wie üblich ausgeschrieben. (Ä = Ae usw.)
  (ß = ss)

- Alle Sonderzeichen (auch Ziffern) außer ' '(Leerzeichen) und '-' werden ignoriert, diese beiden Zeichen werden gleich behandelt.

TEXT OP + (TEXT CONST links, rechts)
Verkettung der Texte 'links' und 'rechts' in dieser Reihenfolge. Die Länge des Resultats ergibt sich aus der Addition der Längen der Operanden.

TEXT OP * (INT CONST faktor, TEXT CONST quelle)
'faktor' fache Erstellung von 'quelle' und Verkettung. Dabei muß

```
times >= 0
```

sein, sonst wird 'niltext' geliefert.

OP CAT (TEXT VAR links, TEXT CONST rechts)
hat die gleiche Wirkung wie

```
links := links + rechts
```

Hinweis: Der Operator 'CAT' hat eine geringere Heap – Belastung als die Operation mit expliziter Zuweisung.

change

PROC change (TEXT VAR senke, TEXT CONST alt, neu)
Ersetzung des (Teil-) TEXTes 'alt' in 'senke' durch 'neu' bei dem erstmaligen Auftreten. Ist 'alt' nicht in 'senke' vorhanden, so wird keine Meldung abgesetzt (Abweichung vom Standard). Die Länge von 'senke' kann sich dabei verändern.
Beispiel:

```
TEXT VAR mein text :: "EUMEL-Benutzerhandbuch";
change (mein text, "Ben", "N");
                    (* EUMEL-Nutzerhandbuch *)
```

PROC change (TEXT VAR senke, INT CONST von, bis, TEXT CONST neu)
Der TEXT 'neu' wird in den TEXT 'senke' anstatt des TEXTes, der zwischen 'von' und 'bis' steht, eingesetzt. Die Länge von 'senke' kann sich dabei verändern.
Beispiel:

```
TEXT VAR mein text :: "EUMEL-Benutzerhandbuch";
change (mein text, 7, 9, "N");   (* wie oben *)
```

change all

PROC change all (TEXT VAR senke, TEXT CONST alt, neu)
Der Teiltext 'alt' wird durch 'neu' in 'senke' ersetzt. Im Unterschied zur 'change'-Prozedur findet die Ersetzung nicht nur bei dem erstmaligen Auftreten von 'alt' statt, sondern so oft, wie 'alt' in 'senke' vorhanden ist. Beispiel:

```
TEXT VAR x :: "Das ist ein Satz";
change all (x, " ", "");      (* DasisteinSatz *)
```

code

TEXT PROC code (INT CONST code)
Wandelt einen INT – Wert 'code' in ein Zeichen um. 'code' muß

```
0 <= code <= 255
```

sein.

INT PROC code (TEXT CONST text)
Wandelt ein Zeichen 'text' in einen INT – Wert um. Ist

```
LENGTH text <> 1
```

dann wird der Wert – 1 geliefert (also bei mehr als ein Zeichen oder niltext).
(Codetabelle auf Seite 5 – 29)

compress

TEXT PROC compress (TEXT CONST text)
Liefert den TEXT 'text' ohne führende und nachfolgende Leerzeichen.

delete char

PROC delete char (TEXT VAR string, INT CONST delete pos)
Löscht ein Zeichen aus dem Text 'string' an der Position 'delete pos'. Für

```
delete pos <= 0
```

oder

```
delete pos > LENGTH string
```

wird keine Aktion vorgenommen.

insert char

PROC insert char (TEXT VAR string, TEXT CONST char,INT CONST insert pos)
Fügt ein Zeichen 'char' in den Text 'string' an der Position 'insert pos' ein. Für

```
insert pos > LENGTH string + 1
```

wird keine Aktion vorgenommen. Daher ist es möglich, mit dieser Prozedur auch am Ende eines Textes (Position: LENGTH string + 1) ein Zeichen anzufügen.

length

INT PROC length (TEXT CONST text)
Anzahl von Zeichen ("Länge") von 'text' einschließlich Leerzeichen.

LENGTH

INT OP LENGTH (TEXT CONST text)
Anzahl von Zeichen ("Länge") von 'text' einschließlich Leerzeichen.

max text length

INT CONST max text length
Maximale Anzahl von Zeichen in einem TEXT (32 000).

pos

INT PROC pos (TEXT CONST quelle, pattern)
Liefert die erste Position des ersten Zeichens von 'pattern' in 'quelle', falls 'pattern' gefunden wird. Wird 'pattern' nicht gefunden oder ist 'pattern' niltext, so wird der Wert '0' geliefert. Beispiel:

```
TEXT VAR t1 :: "abcdefghijk...xyz",
         t2 :: "cd";
... pos (t1, t2) ... (* liefert 3 *)
... pos (t2, t1) ... (* liefert 0 *)
```

INT PROC pos (TEXT CONST quelle, pattern, INT CONST von)
Wie obige Prozedur, jedoch wird erst ab der Position 'von' ab gesucht. Dabei gilt folgende Einschränkung:

```
length (pattern) < 255
```

INT PROC pos (TEXT CONST quelle, low char, high char, INT CONST von
Liefert die Position des ersten Zeichens 'x' in 'quelle' ab der Position 'von', so daß

```
low char <= x <= high char
```

'low char' und 'high char' müssen TEXTe der Länge 1 sein. Wird kein Zeichen in 'quelle' in dem Bereich zwischen 'low char' und 'high char' gefunden, wird der Wert '0' geliefert. Beispiel:

```
(*Suche nach dem ersten Zeichen <> blank nach einer Leerspalte*)
 TEXT VAR zeile :: "BlaBla      Hier gehts weiter";
 INT VAR pos erstes blank :: pos (zeile, " "),
 ende leerspalte  ::
       pos (zeile, ""33"",""254"", pos erstes blank);
```

real

REAL PROC real (TEXT CONST text)
Konvertierung eines TEXTes 'text' in einen REAL – Wert. Achtung: Zur Zeit werden keine Überprüfungen vorgenommen, d.h. in dem TEXT muß ein REAL – Wert stehen.

replace

PROC replace (TEXT VAR senke, INT CONST position, TEXT CONST quelle)
Ersetzung eines Teiltextes in 'senke' durch 'quelle' an der Position 'position' in 'senke'. Es muß gelten

```
1 <= position <= LENGTH senke
```

d.h. 'position' muß innerhalb von 'senke' liegen und 'quelle' muß von der Position 'position' ab in 'senke' einsetzbar sein. Dabei bleibt die Länge von 'senke' unverändert.

SUB

TEXT OP SUB (TEXT CONST text, INT CONST pos)
Liefert ein Zeichen aus 'text' an der Position 'pos'. Entspricht

```
subtext (text, pos, pos)
```

Anmerkung: Effizienter als obiger Prozedur – Aufruf. Für

```
pos <= 0
pos > LENGTH text
```

wird niltext geliefert.

subtext

TEXT PROC subtext (TEXT CONST quelle, INT CONST von)
Teiltext von 'quelle', der bei der Position 'von' anfängt. Die Länge des Resultats ergibt sich also zu

```
LENGTH quelle - von + 1
```

d.h. von der Position 'von' bis zum Ende von 'quelle'. 'von' muß innerhalb von 'quelle' liegen. Ist von < 1, dann wird 'quelle' geliefert. Falls von > LENGTH quelle ist, wird niltext geliefert.

TEXT PROC subtext (TEXT CONST quelle, INT CONST von, bis)
Teiltext von 'quelle' von der Position 'von' bis einschließlich der Position 'bis'. Die Länge des Resultats ist also

```
bis -  von + 1
```

Dabei muß gelten

```
1 <= von <= bis <= LENGTH quelle
```

d.h. die Positionen 'von' und 'bis' müssen in dieser Reihenfolge innerhalb von 'quelle' liegen. Ist

```
bis >= LENGTH quelle
```

wird 'subtext (quelle, von)' ausgeführt. Für die Bedingungen für 'von' siehe vorstehende Beschreibung von 'subtext'.

text

TEXT PROC text (TEXT CONST quelle, INT CONST laenge)
Teiltext aus 'quelle' mit der Länge 'laenge', beginnend bei der Position 1 von 'quelle'. Es muß gelten

```
1 <= laenge <= LENGTH quelle
```

d.h. der gewünschte Teiltext muß aus 'quelle' ausblendbar sein.
Wenn gilt:

```
laenge > LENGTH quelle
```

wird der zu liefernde TEXT mit der an 'laenge' fehlenden Zeichen mit Leerzeichen aufgefüllt.

TEXT PROC text (TEXT CONST quelle, INT CONST laenge, von)
Teiltext aus 'quelle' mit der Länge 'laenge', beginnend an der Position 'von' in dem TEXT 'quelle'. Entspricht

```
text (subtext (quelle, von, LENGTH quelle),laenge)
```

Es muß

```
laenge >= 0
1 <= von <= LENGTH quelle
```

gelten, d.h. 'von' muß eine Position angeben, die innerhalb von 'quelle' liegt. Für

```
laenge > LENGTH quelle - von + 1
```

also wenn die angegebene Länge 'laenge' größer ist als der auszublendende Text, wird das Resultat rechts mit Leerzeichen aufgefüllt. Wenn

```
laenge < LENGTH quelle - von + 1
```

d.h. wenn die angegebene Länge kleiner ist als der Teiltext von 'von' bis zum letzten Zeichen von 'quelle', wird das Resultat mit der Länge

```
LENGTH quelle - von + 1
```

geliefert.

5.3 Der Datentyp FILE (Textdateien)

Der Datentyp FILE definiert Dateien von sequentieller Struktur, die Texte enthalten. Ein Objekt vom Datentyp FILE ist charakterisiert durch:

1) seine Betriebsrichtung (TRANSPUTDIRECTION)
   input = nur lesender Zugriff
   output = nur schreibender Zugriff
   modify = lesender und schreibender Zugriff.
2) seinen Namen.

Betriebsrichtung und Name werden in der Assoziierungsprozedur 'sequential file' (siehe Kap 2.8.2) festgelegt.

```
Beispiel
TEXT VAR name := ausgabe ;

FILE VAR f := sequential file(output,name) ;
```

Das Festlegen einer Betriebsrichtung impliziert eine Kontrolle der Benutzung der betreffenden Datei, hilft somit Programmierfehler zu vermeiden.

ACHTUNG : Alle Prozeduren, die auf FILEs zugreifen, verlangen Objekte vom Typ FILE VAR, da die Lese/Schreiboperationen als ändernd betrachtet werden (müssen).

5.3.1 Assoziierung

sequential file

FILE PROC sequential file
(TRANSPUTDIRECTION CONST mode, DATASPACE VAR ds)

Assoziierung einer sequentiellen Datei mit dem Dataspace 'ds' und der Betriebsrichtung 'mode' (vergl. 'modify', 'input' bzw. 'output'). Diese Prozedur dient zur Assoziierung eines temporären Datenraums in der Benutzer-Task, der nach der Beendigung des Programmlaufs nicht mehr zugriffsfähig ist (weil der Name des Datenraums nicht mehr ansprechbar ist). Somit muß der Datenraum explizit vom Programm gelöscht werden.

FILE PROC sequential file
(TRANSPUTDIRECTION CONST mode,TEXT CONST name)

Assoziierung einer sequentiellen Datei mit dem Namen 'name' und der Betriebsrichtung 'mode' (vergl. 'input' bzw. 'output'). Existiert der FILE bereits, dann wird mit 'input' auf den Anfang des FILEs, bei 'output' hinter den letzten Satz der datei positioniert. Existiert dagegen der FILE noch nicht und ist die TRANSPUTDIRECTION 'output' oder 'modify', wird ein neuer FILE eingerichtet.

FEHLER : "name" gibt es nicht"
Es wurde versucht, einen nicht vorhandenen FILE mit 'input' zu assoziieren.

input

PROC input (FILE VAR f)
Ändern der Verarbeitungsart von 'modify' oder 'output' in 'input'. Dabei wird auf den ersten Satz der Datei positioniert.

TRANSPUTDIRECTION CONST input
Assoziierung in Zusammenhang mit der Prozedur 'sequential file' einer sequentiellen Datei mit der 'TRANSPUTDIRECTION' 'input' (nur lesen).

output

PROC output (FILE VAR file)
Ändern der Verarbeitungsart von 'input' oder 'modify' in 'output'. Dabei wird hinter den letzten Satz der Datei positioniert.

TRANSPUTDIRECTION CONST output
In Verbindung mit der Prozedur 'sequential file' kann eine Datei assoziiert werden mit der Betriebsrichtung 'output' (nur schreiben).

modify

PROC modify (FILE VAR f)
Ändern der Betriebsrichtung von 'input' oder 'output' in die Betriebsrichtung 'modify'.

TRANSPUTDIRECTION CONST modify
Diese Betriebsrichtung erlaubt das Vorwärts – und Rückwärts – Positionieren und das beliebige Einfügen und Löschen von Sätzen. 'modify' wird für die Assoziierungsprozedur 'sequential file' benötigt.

5.3.2 Informationsprozeduren

eof

BOOL PROC eof (FILE CONST file)
Informationsprozedur auf das Ende eines FILEs. Liefert den Wert TRUE, sofern hinter den letzten Satz eines FILEs positioniert wurde.

line no

INT PROC line no (FILE CONST file)
Liefert die aktuelle Zeilennummer.

lines

PROC lines (FILE VAR f)
Liefert die Anzahl der Zeilen der Datei 'f'.

headline

TEXT PROC headline (FILE CONST f)
Liefert den Inhalt der Kopfzeile der Datei 'f'.

PROC headline (FILE VAR f, TEXT CONST ueberschrift)
Setzt 'ueberschrift' in die Kopfzeile der Datei 'f'.

5.3.3 Betriebsrichtung INPUT

In der Betriebsrichtung 'input' sind nur Leseoperationen auf der Datei zugelassen. Die Assoziierungsprozedur 'sequential file' bewirkt:

1) Falls die Eingabedatei noch nicht existiert, wird eine Fehlermeldung ausgegeben.

2) Falls es eine Datei des Namens gibt, wird auf das erste Zeichen des ersten Satzes positioniert.

get

PROC get (FILE VAR f, INT VAR number)
Lesen des nächsten Wortes aus der Datei 'f' und Konvertierung des Wortes zu einem Integer – Objekt.

PROC get (FILE VAR f, REAL VAR number)
Lesen des nächsten Wortes aus der Datei 'f' und Konvertierung des Wortes zu einem Real – Objekt.

PROC get (FILE VAR f, TEXT VAR text)
Lesen des nächsten Wortes aus der Datei 'f'.

PROC get (FILE VAR f, TEXT VAR text, TEXT CONST delimiter)
Lesen eines TEXT – Wertes 'text' von der Datei 'f', bis das Zeichen 'delimiter' angetroffen wird. Ein eventueller Zeilenwechsel in der Datei wird dabei übergangen.

PROC get (FILE VAR f, TEXT VAR text, INT CONST maxlength)
Lesen eines TEXT – Wertes 'text' von der Datei 'f' mit 'maxlength' Zeichen. Ein eventueller Zeilenwechsel in der Datei wird dabei nicht übergangen.

getline

PROC get line (FILE VAR file, TEXT VAR record)
Lesen der nächsten Zeile aus der sequentiellen Datei 'file'.
Mögliche Fehler bei Betriebsrichtung 'input':

"Datei zu"
Die Datei 'file' ist gegenwärtig nicht assoziiert.

"Leseversuch nach Dateiende"
Es wurde versucht, über die letzte Zeile einer Datei zu lesen.

"Leseversuch auf output file"
Es wurde versucht, von einem mit 'output' assoziierten FILE zu lesen.

"Unzulässiger Zugriff auf modify – FILE"

5.3.4 Betriebsrichtung OUTPUT

In der Betriebsrichtung 'output' sind nur Schreiboperationen auf der Datei zugelassen. Die Assoziierungsprozedur 'sequential file' bewirkt:

1) Falls die Ausgabedatei noch nicht existiert, wird sie angelegt und auf den ersten Satz positioniert.

2) Falls es bereits eine Datei des Namens gibt, wird hinter den letzten Satz positioniert, die Datei wird also fortgeschrieben.

put

PROC put (FILE VAR f, INT CONST number)
Ausgabe eines INT – Wertes 'number' in die Datei 'f'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

PROC put (FILE VAR f, REAL CONST number)
Ausgabe eines REAL – Wertes 'number' in die Datei 'f'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

PROC put (FILE VAR f, TEXT CONST text)
Ausgabe eines TEXT – Wertes 'text' in die Datei 'f'. Dabei wird ein Leerzeichen an die Ausgabe angefügt.

putline

PROC putline (FILE VAR file, TEXT CONST record)
Ausgabe eines TEXTes 'record' in die Datei 'file'. Danach wird auf die nächste Zeile positioniert. 'file' muß mit 'output' assoziiert sein.

write

PROC write (FILE VAR f, TEXT CONST text)
Schreibt 'text' in die Datei 'f' (analog 'put (f, text)'), aber ohne Trennblank.

line

PROC line (FILE VAR file)
Positionierung auf die nächste Zeile der Datei 'file'. Wird versucht, über das Ende eines mit 'input' assoziierten FILEs zu positionieren, wird keine Aktion vorgenommen.

PROC line (FILE VAR file, INT CONST lines)
Positionierung mit 'lines' Zeilen Vorschub in der Datei 'file'.

FEHLER: "Datei zu!"
Die Datei 'file' ist gegenwärtig nicht assoziiert.

"Schreibversuch auf input – File"
Es wurde versucht, auf einen mit 'input' assoziierten FILE zu schreiben.

Bei Textdateien, die mit dem Editor weiterbearbeitet werden sollen, ist also zu beachten: Eine Ausgabe mit 'put' setzt ein 'blank' hinter die Ausgabe. Falls dieses Leerzeichen das letzte Zeichen in der Zeile ist, wird eine Absatzmarke in der Zeile gesetzt. Wird mit 'write' oder 'putline' ausgegeben, steht kein Leerzeichen und somit keine Absatzmarke am Zeilenende.

5.3.5 Betriebsrichtung MODIFY

In der Betriebsrichtung 'modify' sind Lese– und Schreiboperationen auf der Datei zugelassen. Desweiteren ist beliebiges Positionieren in der Datei erlaubt. Neue Sätze können an beliebiger Stelle in die Datei eingefügt werden, die sequentielle Struktur der Datei bleibt erhalten. Die Assoziierungsprozedur 'sequential file' bewirkt:

1) Falls die Ausgabedatei noch nicht existiert, wird sie angelegt.

2) Falls es bereits eine Datei des Namens gibt, ist undefiniert wo positioniert ist. Die erste Positionierung muß explizit vorgenommen werden!

col

PROC col (FILE VAR f, INT CONST position)
Positionierung auf die Spalte 'position' innerhalb der aktuellen Zeile.

INT PROC col (FILE CONST f)
Liefert die aktuelle Position innerhalb der aktuellen Zeile.

down

PROC down (FILE VAR f)
Positionieren um eine Zeile vorwärts.

PROC down (FILE VAR f, INT CONST number)
Positionieren um 'number' Zeilen vorwärts.

to line

PROC to line (FILE VAR f, INT CONST number)
Positionierung auf die Zeile 'number'.

PROC up (FILE VAR f)
Positionieren um eine Zeile rückwärts.

PROC up (FILE VAR f, INT CONST number)
Positionieren um 'number' Zeilen rückwärts.

delete record

PROC delete record (FILE VAR file)
Der aktuelle Satz der Datei 'file' wird gelöscht. Der folgende Satz wird der aktuelle Satz.

insert record

PROC insert record (FILE VAR file)
Es wird ein leerer Satz in die Datei 'file' vor die aktuelle Position eingefügt. Dieser Satz kann anschließend mit 'write record' beschrieben werden (d.h. der neue Satz ist jetzt der aktuelle Satz).

read record

PROC read record (FILE CONST file, TEXT VAR record)
Liest den aktuellen Satz der Datei 'file' in den TEXT 'record'. Die Position wird dabei nicht verändert.

write record

PROC write record (FILE VAR file, TEXT CONST record)
Schreibt einen Satz in die Datei 'file' an die aktuelle Position. Dieser Satz muß bereits vorhanden sein, d.h. mit 'write record' kann keine leere Datei beschrieben werden, sondern es wird der Satz an der aktuellen Position überschrieben. Die Position in der Datei wird nicht verändert.

5.3.6 FILE – Ausschnitte

Ähnlich den Editorfunktionen 'ESC RUBOUT' und 'ESC RUBIN', die erlauben ganze Abschnitte einer Datei zu löschen und das Gelöschte an anderer Stelle wiedereinzufügen, gibt es die Möglichkeit per Programm solche Segmente
eines 'modify – FILEs' zu verschieben.

clear removed

PROC clear removed (FILE VAR f)
Das mit 'remove' entfernte Segment wird gelöscht und nicht an anderer Stelle eingefügt.

reinsert

PROC reinsert (FILE VAR f)
Das mit 'remove' entfernte Segment wird vor die aktuelle Zeile wiedereingefügt.

remove

PROC remove (FILE VAR f, INT CONST size)
Löscht 'size' Zeilen vor der aktuellen Position aus 'f'. Das Segment wird in einen internen Puffer geschrieben.

reorganize

PROC reorganize (TEXT CONST datei)
Reorganisation von 'datei'. Die durch Löschen und Einfügen aus vielen Segmenten bestehende Datei wird zu einem Segment zusammengefügt, die aktuelle Position ist danach das erste Zeichen der ersten Zeile.

Durch diese Prozedur kann ggf. Speicherplatz gespart werden.

PROC reorganize
Reorganisation der zuletzt bearbeiteten Datei.

segments

PROC segments (FILE VAR f)
Liefert die Anzahl der Segmente von 'f'. Eine große Anzahl von Segmenten kann langsamere Zugriffe zur Folge haben.

5.4 Suchen und Ersetzen in Textdateien

Such- und Ersetzungsprozeduren können sowohl interaktiv beim Editieren (siehe dazu 3.3), als auch in Prozeduren, die auf FILEs (siehe 5.3) arbeiten, angewandt werden.

Die dazu dienenden Prozeduren sind im Paket 'pattern match' enthalten. Mit 'Pattern Matching' (Muster treffen) wird ein Verfahren bezeichnet Gleichheit von Objekten anhand von Regeln, denen diese Objekte genügen, zu überprüfen.

Da oft nach Texten gesucht werden muß, deren genaue Ausprägung nicht bekannt ist, oder deren Auftreten nur in einem bestimmten Zusammenhang interessiert, gibt es die Möglichkeit feststehende Textelemente mit Elementen ungewisser Ausprägung zu kombinieren, also Textmuster zu erzeugen.

Um einen Text zu suchen, muß die Suchrichtung und der gesuchte Text oder ein Muster, welches diesen Text beschreibt, angegeben werden.

– Aufbauen von Textmustern	:	+ , – , OR , any , bound , notion
– Suchen nach Textmustern	:	down , downety , up , uppety
– Treffer registrieren		LIKE , UNLIKE , at , pattern found
– Treffer herausnehmen		** , match , matchend , matchpos , somefix , word
– Ändern in Dateien	:	change

Nach einem erfolgreichen Suchvorgang ist stets auf das erste Zeichen der zu suchenden Zeichenkette positioniert.

Eine besondere Funktion kommt dem 'joker' zu: Dieses Symbol (Defaultwert: '*') steht für eine beliebige Zeichenkette beliebiger Länge. Insbesondere bei Ersetzungsaktionen in denen dieses Zeichen zur Musterbeschreibung verwendet wird, ist daher Vorsicht geboten und sorgfältig zu testen.

5.4.1 Aufbau von Textmustern

TEXT OP + (TEXT CONST links, rechts)
Verkettung der Texte 'links' und 'rechts' zu 'linksrechts'. Falls das Ergebnis länger als die maximal zulässige Textlänge ist, ist es undefiniert.

Wenn 'muster1' einen beliebigen Text finden sollte, (Siehe: PROC any) wird das Ende des von 'muster1' erkannten Textes durch den Anfang des von 'muster2' erkannten Textes im Nachhinein definiert.

TEXT OP – (TEXT CONST alphabet)
Der Operator liefert das zu 'alphabet' komplementäre Alphabet, also alle Zeichen gemäß der EUMEL Codetabelle (5.2.4), die nicht in 'alphabet' enthalten sind. Sinnvoll im Zusammenhang mit der Textprozedur 'any'.

TEXT OP OR (TEXT CONST links, rechts)
Liefert die Alternative von 'links' und 'rechts'. Die Reihenfolge spielt beim Suchen keine Rolle.

Die Textprozedur 'any' liefert einen unbekannten Text unbestimmter Länge. Dieser Text sollte entweder durch festen Text sinnvoll eingegrenzt werden, oder direkt eingeschränkt werden.

any

Die Textprozedur 'any' liefert einen unbekannten Text unbestimmter Länge. Dieser Text sollte entweder durch festen Text sinnvoll eingegrenzt werden, oder direkt eingeschränkt werden.

TEXT PROC any
Beschreibt einen beliebigen Text.

TEXT PROC any (INT CONST laenge)
Beschreibt einen beliebigen Text der angebenen Länge.

TEXT PROC any (TEXT CONST alphabet)
Beschreibt einen beliebigen Text, der nur aus Zeichen besteht, die in 'alphabet' enthalten sind.

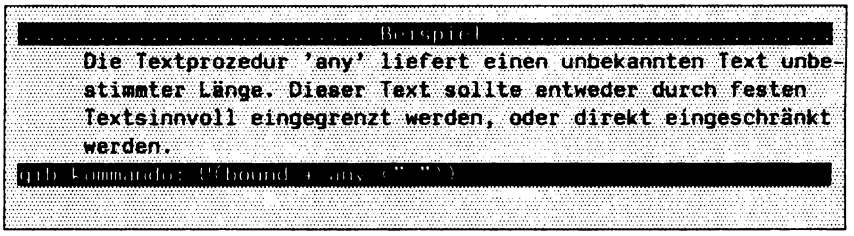
TEXT PROC any (INT CONST laenge, TEXT CONST alphabet)
Beschreibt einen Text der vorgegebenen Länge, der nur aus den in 'alphabet' vorgegebenen Zeichen besteht.

Beispiel

```
Die Textprozedur 'any' liefert einen unbekannten Text unbe-
stimmter Länge. Dieser Text sollte entweder durch festen Text
sinnvoll eingegrenzt werden, oder direkt eingeschränkt werden.
gib Kommando: D"D" OR "d" + any (2,"abcrs")
Sucht nach bestimmten Artikeln: 'der', 'die', 'das'  etc.
```

bound

TEXT PROC bound
Bezeichnet ein Muster der Länge null, das nur am Zeilenanfang oder am Zeilenende gefunden wird. Ein Präfix 'bound' fordert, daß das gesuchte Muster in der ersten Spalte beginnen muß, ein Postfix 'bound' fordert, daß das Muster mit dem Zeilenende abschließt.



liefert Treffer bei eingerückten Zeilen.

notion

PROC notion (TEXT CONST suchwort)
Mit dieser Prozedur kann ein Wort spezifiziert werden, nach dem gesucht werden soll. Bei der Suche nach 'suchwort' wird nur dann ein Treffer geliefert, wenn 'suchwort' als Wort, also begrenzt von ' ' (blank), '.' , ',' oder anderen Sonderzeichen ist.

PROC notion (TEXT CONST suchwort, INT CONST reg)
Wie oben, der Treffer wird im Register 'reg' gespeichert.

5.4.2 Suche nach Textmustern

down

PROC down (FILE VAR f, TEXT CONST muster)
Suche nach 'muster' in der Datei 'f' in Richtung Dateiende. Wird 'muster' gefunden, ist die Position das erste Zeichen von 'muster'. Andernfalls steht man hinter dem letzten Zeichen der Datei.

Achtung: 'down' sucht vom nächsten Zeichen rechts ab, so daß wiederholtes Suchen keine Endlosschleife ergibt.

PROC down (FILE VAR f, TEXT CONST muster, INT CONST number)
Wie obiges 'down', es wird aber maximal nur 'number' – Zeilen weit nach 'muster' gesucht.

downety

PROC downety (FILE VAR f, TEXT CONST muster)
Suche nach 'muster' in der Datei 'f' in Richtung Dateiende. Wird 'muster' gefunden, ist die Position das erste Zeichen von 'muster'. Andernfalls steht man auf dem letzten Zeichen der Datei.

Achtung: 'downety' sucht (im Gegensatz zu 'down') vom aktuellen Zeichen an. Daher muß explizit vorwärts positioniert werden.

PROC downety (FILE VAR f, TEXT CONST muster, INT CONST number)
Wie obiges 'downety', aber maximal nur 'number' – Zeilen weit.

up

PROC up (FILE VAR f, TEXT CONST muster)
Suche nach 'muster' in der Datei 'f' in Richtung Dateianfang. Wird 'muster' gefunden, ist die Position das erste Zeichen von 'muster'. Andernfalls steht man auf dem ersten Zeichen der Datei.

Achtung: 'up' sucht vom nächsten Zeichen links ab, so daß wiederholtes Suchen keine Endlosschleife ergibt.

PROC up (FILE VAR f, TEXT CONST muster, INT CONST number)
Wie obiges 'up', aber maximal nur 'number' – Zeilen weit.

uppety

PROC uppety (FILE VAR f, TEXT CONST muster)
Suche nach 'muster' in der Datei 'f' in Richtung Dateianfang. Wird 'muster' gefunden, ist die Position das erste Zeichen von 'muster'. Andernfalls steht man auf dem ersten Zeichen der Datei.

Achtung: 'uppety' sucht (im Gegensatz zu 'up') vom aktuellen Zeichen.

PROC uppety (FILE VAR f, TEXT CONST muster, INT CONST number)
Wie obiges 'uppety', aber maximal nur 'number' – Zeilen weit.

5.4.3 Treffer registrieren

LIKE

BOOL OP LIKE (TEXT CONST text , muster)
Liefert TRUE, falls der Text 'text' 'muster' entspricht. In 'muster' kann das Spezialzeichen '*' verwandt werden, das abkürzend für die Konkatenation mit 'any' steht.

Daraus folgt, daß das Suchen oder Ersetzen des Zeichens '*' nur durch any (1,"*") zu bewerkstelligen ist.

```
                    Beispiel
#Druckdateien aus Thesaurus löschen#
gib Kommando:"rm *.p" ( "" )
16.04.87        "Handbuch teil1"
04.05.87        "Handbuch teil1.p"
16.04.87        "Handbuch teil2"
06.05.87        "Handbuch teil2.p"
```

aber:

```
                    Beispiel
#Vordere Kommentarklammern löschen #
gib Kommando:"" + ...( 1,"*") ( "" )
lernsequenz auf taste legen("a" , "archive") ;
(* lernsequenz auf taste legen("(" , ""91"") ; *)
(* lernsequenz auf taste legen(")" , ""93"") ; *)
   kommando auf taste legen("p" , "print(""""")"8""8""11"") .
```

UNLIKE

BOOL OP UNLIKE (TEXT CONST text , muster)
Wirkt wie: '(NOT text LIKE muster)'

5.4.4 Treffer herausnehmen

Mit Hilfe der 'Register' ist es möglich identifizierte Texte zwischenzuspeichern und in weiteren Aktionen weiterzuverwenden.

******

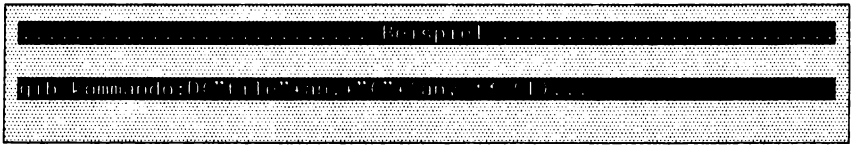
TEXT OP ** (TEXT CONST muster, INT CONST register)
Der als 'muster' erkannte Text wird einem 'Register' mit der Nummer 'register' zugeordnet. Es können 256 Register (1 bis 256) benutzt werden.

match

TEXT PROC match (INT CONST nr)
Liefert den Text der dem Register 'nr' zuletzt zugeordnet wurde.

matchpos

INT PROC matchpos (INT CONST nummer)
Liefert die Spaltennummer, auf der das dem Register 'nummer' zugeordnete Muster in der Zeile beginnt.



5.4.5 Ändern in Dateien

change

PROC change (FILE VAR datei, INT CONST von, bis , TEXT CONST neuertext)
In der Datei wird in der aktuellen Zeile in den Ausschnitt zwischen 'von' und 'bis' der Text 'neuertext' eingesetzt.

entspricht:

```
FILE VAR file := sequential file (modify, name)
TEXT VAR zeile;
    .
    .
  read record (file ,zeile);
  change (zeile, von, bis ,"neuertext");
  write record (file, zeile);
    .
```

5.4.6 Editor – Prozeduren

edit

edit (TEXT CONST datei)
Editieren der Datei 'datei'. Das Editorfenster ist maximal groß (von 1,1 bis max,max). Der Standard – Kommandointerpreter ist gültig, so daß Eingaben, die mit ESC beginnen, interpretiert werden, wie in 3.4 'Vorbelegte Tasten' beschrieben.

edit
Wie oben, editiert wird die Datei mit dem zuletzt benutzten Namen.

edit (THESAURUS CONST thes)
Wie oben, editiert werden alle Dateien, deren Namen im Thesaurus 'thes' enthalten sind.

edit (TEXT CONST datei, INT CONST von x, von y, bis x, bis y)
Editieren der Datei 'datei'. Das Editorfenster hat die linke obere Ecke bei 'von x, von y' und die rechte untere Ecke bei 'bis x, bis y'.

edit (FILE VAR f)
Editieren der als 'sequential file' assoziierten Textdatei 'f'.

edit (FILE VAR, INT CONST von x, von y, bis x, bis y)
Editieren der als 'sequential file' assoziierten Textdatei in einem Fenster mit der linken, oberen Ecke 'von x, von y' und der rechten, unteren Ecke 'bis x, bis y'.

edit (FILE VAR f, TEXT CONST res,
PROC (TEXT CONST) kdo interpreter)
Editieren der als 'sequential file' assoziierten Textdatei 'f'. In 'res' werden reservierte Zeichen übergeben, die von der Prozedur 'kdo interpreter' als Kommandos interpretiert werden, wenn sie als ESC – Sequenz eingegeben werden.
Beispiel : ESC ●

editget

editget (TEXT VAR ausgabe)
Aufruf des Zeileneditor. An der aktuellen Cursorposition wird eine Zeile ausgegeben in der 'ausgabe' steht. Für diese Zeile stehen alle Editiermöglichkeiten zur Verfügung, 'ausgabe' kann also beliebig überschrieben, ergänzt etc. werden. Die Eingabe wird durch CR abgeschlossen. Im Gegensatz zur Prozedur 'get' ist auch eine leere Eingabe möglich.

editget (TEXT VAR ausgabe, INT CONST zeile, INT CONST scroll,
TEXT CONST sep, TEXT CONST res, TEXT VAR exit)
Wie oben, die Zeilenlänge ist jedoch auf 'zeile' Zeichen begrenzt. Die Eingabe wird durch CR oder durch eine Cursorbewegung über die Position 'zeile' hinaus abgeschlossen.

Die Angabe 'scroll' setzt die Breite des Zeilenfensters fest, wird diese Breite überschritten, so wird 'ausgabe' gerollt.

In 'sep' (Separator) können Zeichen festgesetzt werden, mit denen die Eingabe beendet wird (zusätzlich zu CR !).

In 'res' (reservierte Tasten) können Tasten festgelegt werden, die in Verbindung mit ESC die Eingabe beenden.

Wurde der Zeileneditor durch einen Separator verlassen, so steht in 'exit' dieses Zeichen. Falls der Zeileneditor durch eine reservierte Taste verlassen, so enthält 'exit' 'ESC' und die Taste.

editget (TEXT VAR ausgabe, INT CONST zeile, INT CONST scroll)
Bedeutung der Parameter siehe oben.

editget (TEXT VAR ausgabe, TEXT CONST sep, TEXT CONST res,
TEXT VAR exit)
Bedeutung der Parameter siehe oben.

editget (TEXT VAR ausgabe, INT CONST zeile, TEXT VAR exit)
Bedeutung der Parameter siehe oben.

5.4.7 Sortierung von Textdateien

Für die Sortierung von Textdateien gibt es zwei Sortierprogramme:

- Sortierung nach ASCII : sort
- Sortierung nach deutschem Alphabet : lexsort

sort

PROC sort (TEXT CONST datei)
Diese Prozedur sortiert die Datei 'datei' zeilenweise gemäß der von der EUMEL Codetabelle (siehe 5.2.4) vorgegebenen Reihenfolge. Zur Sortierung werden die Zeilen vom ersten Zeichen der Zeile beginnend, zeichenweise verglichen und dementsprechend sortiert.

PROC sort (TEXT CONST datei, INT CONST position)
Sortierkriterien wie oben, jedoch wird bei Vergleich und Sortierung der Satz erst ab der Position 'position' beachtet. Sortiert wird der ganze Satz!

lex sort

PROC lex sort (TEXT CONST datei)
Zeilenweise Sortierung nach lexikographischer Reihenfolge gemäß DIN 5007. Zu den Vergleichen werden die Operatoren LEXEQUAL, LEXGRATER, LEXGRATEREQUAL benutzt (siehe 5.2.4).

PROC lex sort (TEXT CONST datei, INT CONST position)
Lexikalische Sortierung durch Vergleich ab Position 'position'.

5.4.8 Prozeduren auf Datenräumen

Neben den Textdateien gibt es im EUMEL – System den Typ Datenraum, der Objekte jeglichen Typs aufnehmen kann und direkten Zugriff auf die Objekte gewährt (siehe 2.9.2).

Für Objekte von Type Datenraum (nicht für die in Datenräumen enthaltenen Objekte!) existieren folgende Standardprozeduren:

:=

OP := (DATASPACE VAR ds1, DATASPACE CONST ds2)
Der Datenraum 'ds1' wird als Kopie von 'ds2' angelegt. Es handelt sich zunächst um eine logische Kopie, eine physische Kopie wird erst nach einem Schreibzugriff auf 'ds1' oder 'ds2' nötig.

new

DATASPACE PROC new (TEXT CONST dsname)
Liefert einen neuen Datenraum namens 'dsname'.

```
DATASPACE VAR ds := new ("datenraum")
(* ergibt zwei Datenräume 'ds' und 'datenraum'! *)
```

nilspace

DATASPACE PROC nilspace
Der 'nilspace' ist ein leerer Datenraum, der ausschließlich als Quelle zum Kopieren bei der Initialisierung Verwendung finden darf.

old

DATASPACE PROC old (TEXT CONST dsname)
Liefert einen bereits existierenden Datenraum (oder auch eine Datei) mit dem Namen 'dsname'.

FEHLER : "dsname" gibt es nicht

type

PROC type (DATASPACE CONST ds, INT CONST typ)
Der Datenraum 'ds' erhält den frei wählbaren Schlüssel 'typ'. Es muß eine positive Zahl gewählt werden. Der Datenraum muß zum Zeitpunkt der Typzuweisung an ein BOUND Objekt gekoppelt (gewesen) sein.

INT PROC type (DATASPACE CONST ds)
Liefert den Schlüssel des Datenraums 'ds'. Falls 'ds' nie an ein BOUND Objekt gekoppelt war, liefert die Prozedur einen Wert < 0, sonst 0 (keine Zuweisung erfolgt) oder die zugewiesene Typnummer.

dataspaces

INT PROC dataspaces (TASK CONST task)
Liefert die Anzahl der Datenräume der Task 'task'.

INT PROC dataspaces
Anzahl der Datenräume von 'myself'.

ds pages

INT PROC ds pages (DATASPACE CONST ds)
Liefert die Anzahl der durch 'ds' belegten Seiten (je 512 Byte).

storage

INT PROC storage (DATASPACE CONST ds)
Liefert den von 'ds' belegten Speicherplatz in KB.

copy

PROC copy (DATASPACE CONST ds, TEXT CONST datei)
Eine neue Datei mit dem Namen 'datei' wird angelegt. Der Inhalt der Datei ist eine Kopie des Inhalts des Datenraumes 'ds'.

forget

PROC forget (DATASPACE CONST ds)
Der Datenraum 'ds' wird gelöscht[1].

fetch

PROC fetch (DATASPACE CONST ziel, TEXT CONST datei,
TASK CONST manager)
Aus der Task 'manager' wird der Datenraum der Datei 'datei' in den eigenen Datenraum 'ziel' kopiert.

save

PROC save (DATASPACE CONST quelle, TEXT CONST datei,
TASK CONST manager)
Der eigene Datenraum 'quelle' wird in die Datei 'datei' in der Task 'manager' kopiert.

---

1) Durch diese Prozedur steht nicht unmittelbar mehr freier Speicherplatz zur Verfügung. Die physische Räumung von Speicherplatz erfolgt durch die 'Müllabfuhr' bei einem Fixpunkt.

5.5 Eingabe/Ausgabe

– Eingabesteuerzeichen	:	HOP , ← → ↑ ↓ , TAB , RUBIN , RUBOUT CR , MARK , ESC
– Ausgabesteuerzeichen		HOME , ← → ↑ ↓ , CL EOP , CL EOL CPOS , BELL , CR , ENDMARK , BEGINMARK
– Positionierung	:	cursor , get cursor , line , page
– Eingabe	:	get , getline , inchar , incharety
– Ausgabe	:	cout , out , out subtext , put , putline , TIMESOUT , write
– Kontrolle	:	online , pause , sysin , sysout
– Zeitmessung	:	clock , date , day , hour , pause , time time of day

5.5.1 E/A auf Bildschirm

Steuerzeichen und Standardprozeduren zur Ein – Ausgabe am Bildschirm werden zur Steuerung des Dialogverhaltens von Prozeduren benutzt.

5.5.1.1 Eingabesteuerzeichen

Eingabesteuerzeichen werden durch die Funktionstasten (s. 3.2) erzeugt. Die Wirkung der Tasten ist ebenfalls an dieser Stelle beschrieben.

Durch die Tasten werden folgende Codes an Programme gegeben:

Codierung	Bezeichnung
HOP	1
RECHTS	2
OBEN	3
LINKS	8
TAB	9
UNTEN	10
RUBIN	11
RUBOUT	12
CR	13
MARK	16
ESC	27

5.5.1.2 Ausgabesteuerzeichen

Die Ausgabe dieser Zeichen bewirkt folgendes Verhalten der Bildschirmausgabe.

Code	Name	Wirkung
0	NUL	keine Wirkung
1	HOME	Cursor in die linke obere Ecke setzen (Position 0,0!)
2	RECHTS	Cursor eine Stelle nach rechts setzen
3	OBEN	Cursor eine Zeile höher setzen
4	CL EOP	Rest der Seite löschen
5	CL EOL	Rest der Zeile löschen
6	CPOS	Cursor setzen, nächstes Ausgabezeichen bestimmt die y – Position, das darauf folgende die x – Position.
7	BELL	akustisches Signal
8	LINKS	Cursor eine Stelle nach links setzen
10	UNTEN	Cursor eine Stelle nach unten setzen
13	CR	Cursor an den Anfang der nächsten Zeile setzen
14	ENDMARK	Ende des markierten Bereichs
15	BEGINMARK	Anfang des markierten Bereichs

Beispiel

```
TEXT VAR ausgabe := ("7"15"V O R S I C H T"14"7");
     out(ausgabe);
```

5.5.1.3 Positionierung

cursor

PROC cursor (INT CONST column, row)
Positioniert den Cursor auf dem Bildschirm, wobei 'column' die Spalte und 'row' die Zeile angibt. Die zulässigen Bereiche von 'column' und 'row' sind geräteabhängig.

get cursor

PROC get cursor (INT VAR x, y)
Erfragung der aktuellen Cursor – Position. Die Koordinaten des Cursors werden in 'x' und 'y' geliefert. Die aktuelle Cursor – Position ist nach Ausgabe von 'HOME' (Code = 1) oder einer Positionierung des Cursors mit der Prozedur 'cursor' stets definiert. Die Prozedur 'get cursor' liefert jedoch undefinierte Werte, wenn über den rechten Rand einer Zeile hinausgeschrieben wurde (die Wirkung einer solchen Operation hängt von der Hardware eines Terminals ab).

line

PROC line
Es wird zum Anfang einer neuen Zeile positioniert.

PROC line (INT CONST number)
Es werden 'number' Zeilenwechsel vorgenommen.

page

PROC page
Es wird zum Anfang einer neuen Seite positioniert (hier: linke obere Ecke (Position 1,1 !) des Bildschirms, wobei der Bildschirm gelöscht wird).

5.5.1.4 Eingabe

Grundlegende Prozeduren

Die folgenden Prozeduren dienen ausschließlich der Eingabe vom Terminal.

editget

Siehe 5.4.6

getchar

PROC getchar (TEXT VAR zeichen)
Liest genau ein Zeichen von der Tastatur und schreibt es in die Variable 'zeichen'.

inchar

PROC inchar (TEXT VAR character)
Wartet solange, bis ein Zeichen von der Tastatur eingegeben wird, und schreibt dieses Zeichen in die Variable 'character'.

incharety

TEXT PROC incharety
Versucht, ein Zeichen von der Tastatur zu lesen. Wurde kein Zeichen eingegeben, wird niltext geliefert.

TEXT PROC incharety (INT CONST time limit)
Versucht, ein Zeichen vom Bildschirm zu lesen. Dabei wird maximal eine 'time limit' lange Zeit auf das Zeichen gewartet (gemessen in Zehntel – Sekunden).

Umleitbare Eingabeprozeduren

Die folgenden Eingabeprozeduren lesen ebenfalls vom Terminal, die Eingabequelle kann jedoch durch die Prozedur 'sysin' umgestellt werden. Falls in 'sysin' eine Datei angegeben wird wird die Eingabe statt vom Terminal aus dieser Datei gelesen.

sysin

PROC sysin (TEXT CONST file name)
Eingabe-Routinen lesen nicht mehr vom Benutzer-Terminal, sondern aus der Datei 'file name'.

TEXT PROC sysin
Liefert den Namen der eingestellten 'sysin'-Datei. "" bezeichnet das Benutzer-Terminal.

get

PROC get (INT VAR number)
Einlesen eines INT-Wertes vom Bildschirm. Der einzulesende INT-Wert kann bei der Eingabe vom Terminal editiert werden.

PROC get (REAL VAR value)
Einlesen eines REAL-Wertes vom Bildschirm. Der einzulesende REAL-Wert kann bei der Eingabe vom Terminal editiert werden.

PROC get (TEXT VAR word)
Liest einen Text in die Variable 'word' mit maximal 255 Zeichen. Es werden solange Zeichen vom Terminal gelesen, bis ein Leerzeichen oder CR eingegeben wird. Dabei werden führende Leerzeichen übergeben. Der einzulesende Text kann bei der Eingabe editiert werden. Eine leere Eingabe ist nicht erlaubt.

PROC get (TEXT VAR word, INT CONST laenge)
Liest einen Text vom Bildschirm mit der Länge 'laenge' oder bis CR eingegeben wird. Der einzulesende Wert kann bei der Eingabe editiert werden.

PROC get (TEXT VAR word, TEXT CONST separator)
Liest einen Text vom Bildschirm, bis ein Zeichen 'separator' angetroffen oder CR eingegeben wird. Der einzulesende Text kann bei der Eingabe editiert werden.

getline

PROC get line (TEXT VAR line)
Das System wartet auf eine Zeile vom Bildschirm (max. 255 Zeichen). CR beendet die Eingabe.

5.5.1.5 Ausgabe

Grundlegende Prozeduren

Die folgenden Prozeduren dienen ausschließlich der Ausgabe auf das Terminal.

cout

PROC cout (INT CONST number)
Schreibt 'number' an die aktuelle Cursor – Position auf den Bildschirm. Anschließend wird an diese Position wieder zurück positioniert. 'number' muß > 0 sein. Paßt 'number' nicht mehr auf die Zeile, so ist die Wirkung von 'cout' nicht definiert. 'cout' gibt den Wert von 'number' nur aus, wenn genügend freie Kanal – Kapazität für diese Ausgabe vorhanden ist. Das hat zur Folge, daß Programme nicht auf die Beendigung einer Ausgabe von 'number' warten müssen und ggf. Ausgaben überschlagen werden.

out

PROC out (TEXT CONST text)
Ausgabe eines Textes auf dem Bildschirm. Im Unterschied zu 'put' wird kein Blank an den ausgegebenen Text angefügt.

out subtext

PROC out subtext (TEXT CONST source, INT CONST from)
Ausgabe eines Teiltextes von 'source' von der Position 'from' bis Textende. Es wird keine Aktion vorgenommen für

```
from > LENGTH source
```

PROC out subtext (TEXT CONST source, INT CONST from, to)
Ausgabe eines Teiltextes von 'source' von der Position 'from' bis zur Position 'to'. Für

```
to > LENGTH source
```

wird out subtext (source, from) ausgeführt.

PROC out text (TEXT CONST source, INT CONST from, to)
Ausgabe eines Teiltextes von 'source' von der Position 'from' bis zur Position 'to'. Für

```
to > LENGTH source
```

wird für die fehlenden Zeichen Blanks ausgegeben.

TIMESOUT

OP TIMESOUT (INT CONST times, TEXT CONST text)
Ausgabe eines TEXTes 'text' 'times'mal. An die Ausgabe wird im Gegensatz zu 'put' kein Leerzeichen angefügt. Es wird kein Text ausgegeben für

```
times < 1
```

Umleitbare Ausgabeprozeduren

Die folgenden Ausgabeprozeduren schreiben ebenfalls auf das Terminal, die Ausgabe kann jedoch durch die Prozedur 'sysout' umgeleitet werden. Falls in 'sysout' eine Datei angegeben wird wird die Ausgabe statt zum
Terminal in die angegebene Datei geleitet.

sysout

PROC sysout (TEXT CONST file name)
Ausgabe – Routinen gehen nicht mehr zum Benutzer – Terminal, sondern in die Datei 'file name'.

TEXT PROC sysout
Liefert den Namen der eingestellten 'sysout' – Datei. "" bezeichnet das Benutzer – Terminal.

line

line
Positionierung auf den Anfang einer neuen Ausgabezeile.

line (INT CONST faktor)
Nächste Ausgabezeile um 'faktor' Zeilen weiter positionieren.

put

PROC put (INT CONST number)
Ausgabe eines INT – Wertes auf dem Bildschirm. Anschließend wird ein Leerzeichen ausgegeben.

PROC put (REAL CONST real)
Ausgabe eines REAL – Wertes auf dem Bildschirm. Anschließend wird ein Leerzeichen ausgegeben.

PROC put (TEXT CONST text)
Ausgabe eines Textes auf dem Bildschirm. Nach der Ausgabe von 'text' wird ein Blank ausgegeben, um nachfolgenden Ausgaben auf der gleichen Zeile voneinander zu trennen. Hardwareabhängig sind die Aktionen, wenn eine Ausgabe über eine Zeilengrenze (hier: Bildschirmzeile) vorgenommen wird. Meist wird die Ausgabe auf der nächsten Zeile fortgesetzt.

putline

PROC putline (TEXT CONST text)
Ausgabe von 'text' auf dem Bildschirm. Nach der Ausgabe wird auf den Anfang der nächsten Zeile positioniert. Gibt man TEXTe nur mit 'putline' aus, so ist gesichert, daß jede Ausgabe auf einer neuen Zeile beginnt. Hardwareabhängig sind die Aktionen, wenn eine Ausgabe über eine Zeilengrenze (hier: Bildschirmzeile) vorgenommen wird. Meist wird die Ausgabe auf der nächsten Zeile fortgesetzt.

write

PROC write (TEXT CONST text)
Gibt 'text' ohne Trennblank aus ('put' mit Trennblank).

5.5.1.6 Kontrolle

online

BOOL PROC online
Liefert TRUE, wenn die Task mit einem Terminal gekoppelt ist.

pause

PROC pause (INT CONST time limit)
Wartet 'time limit' in Zehntel – Sekunden. Bei negativen Werten ist die Wirkung nicht definiert. Die Wartezeit wird nicht nur durch das Erreichen der Grenze abgebrochen, sondern auch durch die Eingabe eines beliebigen Zeichens.

PROC pause
Wartet bis zur Eingabe eines beliebigen Zeichens.

5.5.2 Zeitmessung

clock

REAL PROC clock (INT CONST index)
Datum und Uhrzeit werden vom EUMEL – System für alle Tasks geführt. Neben einer Uhr ('Realzeituhr'), die das Datum und die aktuelle Uhrzeit enthält, wird eine Uhr für die von der Task verbrauchte CPU – Zeit geführt ('CPU – Zeituhr'). Beide Zeiten werden vom System als REALs realisiert. Die Prozedur 'clock' liefert die aktuellen Werte dieser Uhren. Bei 'index = 0' wird die akkumulierte CPU – Zeit der Task, bei 'index = 1' der Wert der Realzeituhr geliefert.

Mit den REAL – Werten der Uhren kann ohne weiteres gerechnet werden, jedoch sind nur Werte > 0 definiert. Die REAL – Werte der Realzeituhr beginnen beim 1.1.1900 um 0 Uhr. Es sind nur Werte für dieses Jahrhundert zugelassen. Werte der Realzeituhr in lesbarer Form kann man durch die Konvertierungsprozeduren 'date' (vergl. 5 – 81) (für den aktuellen Tag) und 'time of day' (Uhrzeit, vergl. 5 – 82) erhalten.

Um die benötigte CPU – Zeit eines Programms zu berechnen, muß man die CPU – Zeituhr zweimal abfragen. Um solche Zeiten in lesbarer Form zu erhalten, kann man die Konvertierungsprozedur 'time' (vergl. 5 – 82) verwenden. Beispiel:

Beispiel

```
REAL CONST anfang := clock (0);
     berechnungen;
REAL CONST ende   := clock (0);
  put ("benoetigte CPU-Zeit in Sek:");
  put (time (ende - anfang))
```

date

TEXT PROC date (REAL CONST time)
Konvertierungsprozedur für das Datum, welches sich aus dem Aufruf der Prozedur 'clock (1)' ergibt. Das Datum wird in der Form 'tt.mm.jj' geliefert. Beispiel:

```
put (date (clock (1)))  (* z.B.: 24.12.87 *)
```

REAL PROC date (TEXT CONST datum)
Konvertierungsprozedur für ein Datum in der Form 'tt.mm.jj'. Liefert einen REAL – Wert, wie ihn die Prozedur 'clock (1)' liefern würde. Beispiel:

```
put (date ("24.12.87")) (* 6.273539e10 *)
```

TEXT PROC date
Liefert das Tagesdatum. Wirkt wie 'date (clock (1))', ist jedoch erheblich schneller.

day

REAL CONST day
Liefert die Anzahl der Sekunden eines Tages (86 400.0).

hour

REAL CONST hour
Liefert die Anzahl der Sekunden einer Stunde (3600.0).

pause

PROC pause (INT CONST time limit)
Wartet 'time limit' in Zehntel – Sekunden. Bei negativen Werten ist die Wirkung nicht definiert. Die Wartezeit wird nicht nur durch das Erreichen der Grenze abgebrochen, sondern auch durch die Eingabe eines beliebigen Zeichens.

time

TEXT PROC time (REAL CONST time)
Konvertierungsprozedur für die Zeiten der CPU – Zeituhr. Liefert die Zeiten in der Form 'hh:mm:ss.s'. Vergl. dazu 'clock'.

TEXT PROC time (REAL CONST value, INT CONST laenge)
Konvertiert die Zeit in externe Darstellung. Für die 'laenge' – Werte ergibt sich:

```
laenge = 10              (*   hh:mm:ss.s *)
laenge = 12              (* hhhh:mm:ss.s *)
```

REAL PROC time (TEXT CONST time)
Konvertierungsprozedur für Texte der CPU – Zeituhr in REAL – Werte.

time of day

TEXT PROC time of day (REAL CONST time)
Konvertierungsprozedur für REALs, wie sie die Realzeituhr
liefert. Es wird die Tageszeit in der Form 'hh:mm' geliefert. Beispiel:

```
put (time of day (clock (1)))  (* z.B.: 17:25 *)
```

TEXT PROC time of day
Liefert die aktuelle Tageszeit. Entspricht

```
time of day (clock (1))
```

5.6 Scanner

Der Scanner kann benutzt werden, um festzustellen, welche Art von Symbolen in einem TEXT enthalten sind. Die Repräsentation der Symbole müssen dabei der ELAN – Syntax entsprechen. Folgende Symbole kann der Scanner erkennen:

- "tags", d.h. Namen,
- "bolds", d.h. Schlüsselworte,
- "number", d.h. INT oder REAL Zahlen,
- Operatoren,
- "delimiter", d.h. Begrenzer wie z.B. ";",
- und das Ende des Scan – Textes.

Der Scanner überliest Kommentare und Leerzeichen zwischen den Symbolen. Der (erste) zu verarbeitende Text muß mit der Prozedur

```
scan
```

in den Scanner "hineingesteckt" werden. Mit der Prozedur

```
next symbol
```

wird das jeweils nächste Symbol des TEXTes geholt. Am Ende wird "end of scan" und als Symbol 'niltext' geliefert. Falls innerhalb eines TEXT – Denoters oder eines Kommentars "end of scan" auftritt, wird "within text" bzw. "within comment" gemeldet. Der Scan – Prozeß kann dann mit dem nächsten zu scannenden TEXT (der nächsten Zeile) fortgesetzt werden. Dafür wird nicht die Prozedur 'scan', sondern

```
continue scan
```

verwandt. Sie setzt im letzten Scan – Zustand (z.B. Kommentar oder TEXT – Denoter) wieder auf, so daß auch Folgen von TEXTen (Zeilen) wie z.B. Dateien leicht gescannt werden können.

Mit den Prozeduren

```
scan              (* meldet eine Datei zum scannen an *)
next symbol       (* holt die Symbole                  *)
```

kann man auch eine Datei nach ELAN – Symbolen untersuchen.

```
FILE VAR f :: ...
...
scan (f);          (* beginnt das Scanning in
                      der nächsten Zeile      *)
TEXT VAR symbol;
INT VAR type;
REP
  next symbol (f, symbol, type);
  verarbeite symbol
UNTIL type >= 7 END REP.
```

Scanner – Kommandos

continue scan

PROC continue scan (TEXT CONST scan text)
Das Scanning soll mit 'scan text' fortgesetzt werden. Falls der Scan – Vorgang beim vorigen 'scan text' innerhalb eines TEXT – Denoters oder eines Kommentars abgebrochen wurde, wird er jetzt entsprechend mit dem nächsten 'next symbol' fortgesetzt. Der erste Teil – Scan einer Folge muß aber stets mit 'scan' eingeleitet werden!

next symbol

PROC next symbol (TEXT VAR symbol, INT VAR type)
Holt das nächste Symbol. In "symbol" steht der TEXT des Symbols, so z.B. die Ziffern eines INT – Denoters. Bei TEXT – Denotern werden die führenden und abschließenden Anführungsstriche abgeschnitten. Leerzeichen oder Kommentare spielen in "tags" oder "numbers" keine Rolle. Zwischen Symbolen spielen Leerzeichen oder Kommentare keine Rolle. In "type" steht eine Kennzeichnung für den Typ des Symbols:

tag	= 1,
bold	= 2,
number	= 3,
text	= 4,
operator	= 5,
delimiter	= 6,
end of file	= 7,
within comment	= 8,
within text	= 9.

Wird Scan – Ende innerhalb eines Kommentars gefunden, so wird 'niltext' und 'within comment' geliefert. Wird Scan – Ende innerhalb eines TEXT – Denoters gefunden, so wird der schon analysierte Teil des Denoters und 'within text' geliefert.

PROC next symbol (TEXT VAR symbol)
s.o. Es wird aber nur der Text des Symbols (ohne Typ) geliefert.

PROC next symbol (FILE VAR f, TEXT CONST symbol)
Arbeitet wie obige Prozeduren, jedoch auf einen FILE.

PROC next symbol (FILE VAR f, TEXT CONST symbol, INT VAR type)
Arbeitet wie obige Prozeduren, jedoch auf einen FILE.

scan

PROC scan (TEXT CONST scan text)
Meldet einen 'scan text' für den Scanner zur Verarbeitung an. Die Prozedur 'scan' muß vor dem ersten Aufruf von 'next symbol' gegeben werden. Im Gegensatz zu 'continue scan' normiert 'scan' den inneren Zustand des Scanners, d.h. vorherige Scan – Vorgänge haben keinen Einfluß mehr auf das Scanning.

PROC scan (FILE VAR f)
Wie obige Prozedur, jedoch auf einen FILE. Die zu scannende Zeile ist die nächste Zeile im FILE 'f' ('scan' macht zuerst ein 'getline').

TEIL 6: Das Archiv 'std.zusatz'

Das Archiv 'std.zusatz' enthält Pakete, die nur bei Bedarf insertiert werden sollen. Eine Einbindung in das EUMEL Grundsystem würde dieses ungebührlich unfangreich machen.

Das Archiv enthält zusätzliche Software für:

– mathematische Operationen	:	complex , longint , vector , matrix
– Analyse	:	reporter , referencer
– Taschenrechnerfunktion zur Editor – Erweiterung	:	TeCal , TeCal Auskunft

6.1. Erweiterungen um Mathematische Operationen

6.1.1 COMPLEX

Das Packet COMPLEX erweitert das System um den Datentyp COMPLEX (komplexe Zahlen) und Operationen auf komplexen Zahlen. Folgende Operationen stehen für COMPLEX zur Verfügung:

– Einfache Operatoren		:= , = , <> , + , – , *
– Eingabe/Ausgabe	:	get , put
– Denotierungsprozedur		complex , complex i , complex one , complex zero
– Komponenten		real part , imag part
– bes. Funktionen	:	ABS , CONJ , phi , dphi , sqrt

COMPLEX Operationen

TYPE COMPLEX

Komplexe Zahl, bestehend aud Realteil 're' und Imaginärteil 'im'.

:=

OP := (COMPLEX VAR a, COMPLEX CONST b)
Zuweisung.

=

BOOL OP = (COMPLEX CONST a, b)
Vergleich von 'a' und 'b' auf Gleichheit.

<>

BOOL OP <> (COMPLEX CONST a, b)
Vergleich von 'a' und 'b' auf Ungleichheit.

+

COMPLEX OP + (COMPLEX CONST a, b)
Summe von 'a' und 'b'.

-

COMPLEX OP - (COMPLEX CONST a, b)
Differenz von 'a' und 'b'.

COMPLEX OP * (COMPLEX CONST a, b)
Multiplikation von 'a' mit 'b'.

COMPLEX OP / (COMPLEX CONST a, b)
Division von 'a' mit 'b'.

get

PROC get (COMPLEX VAR a)
Einlesen eines komplexen Wertes vom Bildschirm in der Form zweier REAL – Denoter. Die Eingabe kann editiert werden.

put

PROC put (COMPLEX CONST a)
Ausgabe eines komplexen Wertes auf dem Bildschirm in Form zweier REAL – Werte. Hinter jedem REAL – Wert wird ein Leerzeichen angefügt.

complex

COMPLEX PROC complex (REAL CONST re, im)
Denotierungsprozedur. Angabe in kartesischen Koordinaten.

complex i

COMPLEX PROC complex i
Denotierungsprozedur für den komplexen Wert '0.0 + i 1.0'.

complex one

COMPLEX PROC complex one
Denotierungsprozedur für den komplexen Wert '1.0 + i 0.0'.

complex zero

COMPLEX PROC complex zero
Denotierungsprozedur für den komplexen Wert '0.0 + i 0.0'.

imag part

REAL PROC imag part (COMPLEX CONST number)
Liefert den Imaginärteil des komplexen Wertes 'number'.

real part

REAL PROC real part (COMPLEX CONST number)
Liefert den Real – Teil des komplexen Wertes 'number'.

ABS

REAL OP ABS (COMPLEX CONST x)
REAL – Betrag von 'x'.

CONJ

COMPLEX OP CONJ (COMPLEX CONST number)
Liefert den konjugiert komplexen Wert von 'number'.

dphi

REAL PROC dphi (COMPLEX CONST x)
Winkel von 'x' (Polardarstellung).

phi

REAL PROC phi (COMPLEX CONST x)
Winkel von 'x' (Polardarstellung) in Radiant.

sqrt

COMPLEX PROC sqrt (COMPLEX CONST x)
Wurzelfunktion für komplexe Werte.

6.1.2 LONGINT

LONGINT ist ein Datentyp, für den (fast) alle Prozeduren und Operatoren des Datentyps INT implementiert wurden. LONGINT unterscheidet sich von INT dadurch, daß erheblich größere Werte darstellbar sind.

Für den Datentyp LONGINT stehen folgende Operationen zur Verfügung:

– Operatoren		:= , = , <> , < , <= ,> , >= , + , – , * , ** , ABS , DECR , DIV , INCR , MOD , SIGN
– Eingabe/Ausgabe	:	get , put
– Math. Prozeduren	:	abs , int , longint , max , max logint , min , random , sign , text , zero

LONGINT – Operationen

TYPE LONGINT

Datentyp

:=

OP := (LONGINT VAR links, LONGINT CONST rechts) :
Zuweisungsoperator

=

BOOL OP = (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf Gleichheit.

<>

BOOL OP <> (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf Ungleichheit.

<

BOOL OP < (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf kleiner.

<=

BOOL OP <= (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf kleiner gleich.

BOOL OP > (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf größer.

BOOL OP >= (LONGINT CONST links, rechts)
Vergleichen zweier LONGINTs auf größer gleich.

LONGINT OP + (LONGINT CONST argument)
Monadischer Operator. Ohne Wirkung.

LONGINT OP + (LONGINT CONST links, rechts)
Addition zweier LONGINTs.

LONGINT OP – (LONGINT CONST argument)
Vorzeichenumkehrung.

LONGINT OP – (LONGINT CONST links, rechts)
Subtraktion zweier LONGINTs.

LONGINT OP * (LONGINT CONST links, rechts)
Multiplikation von zwei LONGINTs.

**

LONGINT OP ** (LONGINT CONST argument, exponent)
Exponentiation zweier LONGINTs mit positivem Exponenten.

FEHLER :

LONGINT OP ** : negative exponent
Der 'exponent' muß > = 0 sein.
0 ** 0 is not defined
'argument' und 'exponent' dürfen nicht gleich 0 sein.

LONGINT OP ** (LONGINT CONST argument, INT CONST exponent)
Exponentiation eines LONGINT mit positiven INT Exponenten.

FEHLER :

LONGINT OP ** : negative exponent
Der 'exponent' muß > = 0 sein.
0 ** 0 is not defined
'argument' und 'exponent' dürfen nicht gleich 0 sein.

ABS

LONGINT OP ABS (LONGINT CONST argument)
Absolutbetrag eines LONGINT.

DECR

OP DECR (LONGINT VAR resultat, LONGINT CONST ab)
resultat := resultat – ab

DIV

LONGINT OP DIV (LONGINT CONST links, rechts)
Division zweier LONGINTs.

FEHLER :
Division durch 0
'rechts' muß <> 0 sein.

INCR

LONGINT OP INCR (LONGINT VAR resultat, LONGINT CONST dazu)
resultat := resultat + dazu

MOD

LONGINT OP MOD (LONGINT CONST links, rechts)
Modulo – Funktion für LONGINTs. Der Rest einer LONGINT – Division wird ermittelt.

FEHLER :
text (links) + 'MOD 0'
'rechts' muß ungleich null sein.

SIGN

INT OP SIGN (LONGINT CONST longint)
Feststellen des Vorzeichens von 'longint'. Liefert:

```
 0 wenn 'longint' = 0,
 1 wenn 'longint' > 0,
-1 wenn 'longint' < 0.
```

get

PROC get (LONGINT VAR zahl)
Eingabe eines LONGINTs vom Terminal.

PROC get (FILE VAR file, LONGINT VAR zahl)
Einlesen von 'zahl' aus der sequentiellen Datei 'file'. Die Datei muß mit 'input' assoziiert sein (vergl. 'sequential file').

FEHLER :
Datei zu
Leseversuch nach Daateiende
Leseversuch auf output – FILE

put

PROC put (LONGINT CONST longint)
Ausgabe eines LONGINTs auf dem Bildschirm. Anschließend wird ein Leerzeichen ausgegeben. Hardwareabhängig sind die Aktionen, wenn eine Ausgabe über die Bildschirmzeilengrenze vorgenommen wird. Meist wird jedoch die Ausgabe auf der nächsten Zeile fortgesetzt.

PROC put (FILE VAR file, LONGINT CONST zahl)
Ausgabe von 'zahl' in die sequentielle Datei 'file'. 'file' muß mit 'output' assoziiert sein.

FEHLER :
Datei zu
Schreibversuch auf input – FILE

abs

LONGINT PROC abs (LONGINT CONST argument)
Absolutbetrag eines LONGINT.

int

INT PROC int (LONGINT CONST longint)
Konvertierung von LONGINT nach INT.

FEHLER :
integer overflow
'longint' ist größer als 'maxint'.

longint

LONGINT PROC longint (INT CONST int)
Konvertierung von 'int' nach LONGINT.

LONGINT PROC longint (TEXT CONST text)
Konvertierung von 'text' nach LONGINT.

max

LONGINT PROC max (LONGINT CONST links, rechts)
Liefert das Maximum zweier LONGINTs.

maxlongint

LONGINT PROC max longint
Liefert größten LONGINT Wert.

min

LONGINT PROC min (LONGINT CONST links, rechts)
Liefert das Minimum zweier LONGINTs.

random

LONGINT PROC random (LONGINT CONST lower bound, upper bound)
Pseudo – Zufallszahlen – Generator im Intervall 'lower bound' und 'upper bound' einschließlich. Es handelt sich hier um den 'LONGINT Random Generator'.

sign

INT PROC sign (LONGINT CONST longint)
Feststellen des Vorzeichens von 'longint'. Liefert:

```
 0 wenn 'longint' = 0,
 1 wenn 'longint' > 0,
-1 wenn 'longint' < 0.
```

text

TEXT PROC text (LONGINT CONST longint)
Konvertierung von 'longint' nach TEXT.

TEXT PROC text (LONGINT CONST longint, INT CONST laenge)
Konvertierung von 'longint' nach TEXT. Die Anzahl der Zeichen soll 'laenge' betragen. Für

```
LENGTH (text (longint)) < laenge
```

werden die Zeichen rechtsbündig in einen Text mit der Länge 'laenge' eingetragen. Ist der daraus entstehende TEXT kleiner als 'laenge', werden die an 'laenge' fehlenden Zeichen im TEXT mit Leerzeichen aufgefüllt. Für

```
LENGTH (text (longint)) > laenge
```

wird ein Text mit der Länge 'laenge' geliefert, der mit '*' – Zeichen gefüllt ist.

zero

LONGINT PROC zero
Liefert LONGINT Wert Null.

6.1.3 VECTOR

Der Datentyp VECTOR erlaubt Operationen auf Vektoren aus Elementen vom Typ REAL. Im Gegensatz zur Struktur 'ROW m REAL' muß die Anzahl der Elemente nicht zur Übersetzungszeit deklariert werden, sondern kann zur Laufzeit festgelegt werden. Somit kann eine zur Übersetzungszeit unbekannte Anzahl von REALs bearbeitet werden, wobei nur soviel Speicherplatz wie nötig verwendet wird. Die maximale Größe eines VECTOR beträgt 4000 Elemente.

Der in den Operationen ':=', 'idn' und 'vector' benutzte Datentyp INITVECTOR wird nur intern gehalten. Er dient der Speicherplatzersparnis bei der Initialisierung.

– Operatoren	:	:= , = , <> , + , – , * , / LENGTH , SUB
– Eingabe/Ausgabe		get , put
– Besondere Vector – Operationen		length , nilvector , norm , vector , replace

OP := (VECTOR VAR ziel, VECTOR CONST quelle)
Zuweisung. Nach der Zuweisung gilt auch

```
length (quelle) = length (ziel)
```

d.h. der linke Operand besitzt nach der Zuweisung genauso viele Elemente wie 'quelle', unabhängig davon, ob 'ziel' vor der Zuweisung mehr oder weniger Elemente als 'quelle' besaß. Beispiel:

```
VECTOR VAR y :: vector (10, 1.0),
           z :: vector (15, 2.0);
     ...
y := z; (* length (y) liefert nun 15 ! *)
```

OP := (VECTOR VAR ziel, INITVECTOR CONST quelle)
Dient zur Initialisierung eines VECTORs. Beispiel:

```
VECTOR VAR x :: vector (17);
```

'vector' erzeugt ein Objekt vom Datentyp INITVECTOR. Dieses Objekt braucht nicht soviel Speicherplatz wie ein VECTOR – Objekt. Dadurch wird vermieden, daß nach erfolgter Zuweisung nicht ein durch 'vector' erzeugtes Objekt auf dem Heap unnötig Speicherplatz verbraucht.

BOOL OP = (VECTOR CONST a, b)
Vergleich zweier Vektoren. Der Operator liefert FALSE, wenn die Anzahl der Elemente von 'a' und 'b' ungleich ist oder wenn zwei Elemente mit gleichem Index ungleich sind. Beispiel:

```
VECTOR VAR x :: vector (10, 1.0),
           y :: vector (15, 2.0),
           z :: vector (10, 1.0);
      ... x = y ... (* FALSE *)
      ... x = z ... (* TRUE  *)
```

BOOL OP <> (VECTOR CONST a, b)
Vergleich zweier Vektoren auf Ungleichheit (NOT (a = b)).

VECTOR OP + (VECTOR CONST a)
Monadisches '+' für VECTOR. Keine Auswirkung.

VECTOR OP + (VECTOR CONST a, b)
Elementweise Addition der Vektoren 'a' und 'b'. Beispiel:

```
VECTOR VAR x, (* 'x' hat undefinierte Länge *)
           a :: vector (10, 1.0),
           b :: vector (10, 2.0);
     ...
x := a + b; (* 'x' hat nun 10 Elemente mit Werten'3.0'
*)
```

FEHLER :
VECTOR OP + : LENGTH a <> LENGTH b
'a' und 'b' haben nicht die gleiche Anzahl von Elementen.

VECTOR OP – (VECTOR CONST a)
Monadisches '–'.

VECTOR OP – (VECTOR CONST a, b)
Elementweise Subtraktion der Vektoren 'a' und 'b'.

FEHLER :
VECTOR OP – : LENGTH a <> LENGTH b
'a' und 'b' haben nicht die gleiche Anzahl von Elementen.

*

REAL OP * (VECTOR CONST a, b)
Skalarprodukt zweier Vektoren. Liefert die Summe der elementweisen Multiplikation der Vektoren 'a' und 'b'. Beachte eventuelle Rundungsfehler! Beispiel:

```
REAL VAR a;
VECTOR VAR b :: vector (10, 2.0),
           c :: vector (10, 2.0);
     ...
a := b * c; (* 40.0 *)
```

FEHLER :
REAL OP * : LENGTH a <> LENGTH b
'a' und 'b' haben nicht die gleiche Anzahl von Elementen.

VECTOR OP * (VECTOR CONST a, REAL CONST s)
Multiplikation des Vektors 'a' mit dem Skalar 's'.

VECTOR OP * (REAL CONST s, VECTOR CONST a)
Multiplikation des Skalars 's' mit dem Vektor 'a'.

/

VECTOR OP / (VECTOR CONST a, REAL CONST s)
Division des Vektors 'a' durch den Skalar 's'. Beispiel:

```
VECTOR VAR a, (* 'a' hat undefinierte Laenge *)
           b :: vector (10, 4.0);
     ...
a := b / 2.0;
(* 'a' hat nun 10 Elemente mit Werten '2.0' *)
```

LENGTH

INT OP LENGTH (VECTOR CONST a)
Liefert die Anzahl der Elemente von 'a'.

SUB

REAL OP SUB (VECTOR CONST v, INT CONST i)
Liefert das 'i' – te Element von 'v'.

FEHLER :

OP SUB : subscript overflow
Der Index 'i' liegt außerhalb des Vektors (i > LENGTH v).
OP SUB : subscript underflow
Der Index 'i' liegt außerhalb des Vektors (i < 1).

PROC get (VECTOR VAR a, INT CONST l)
Einlesen der Elemente von 'a' vom Terminal, wobei 'l' die Anzahl der Elemente angibt.

FEHLER :

PROC get : size <= 0
Die angeforderte Elementanzahl 'l' muß > 0 sein.

PROC put (VECTOR CONST v)
Ausgabe der Werte der Elemente von 'v' auf dem Terminal.

length

INT PROC length (VECTOR CONST a)
Liefert die Anzahl der Elemente von 'a'. Beispiel:

```
VECTOR VAR a :: vector (10, 1.0),
           b :: vector (15, 2.0);
    ...
... length (a) ... (* 10 *)
... length (b) ... (* 15 *)
```

nilvector

INITVECTOR PROC nilvector
Erzeugen eines Vektors mit einem Element mit dem Wert '0.0'.

norm

REAL PROC norm (VECTOR CONST v)
Euklidische Norm (Wurzel aus der Summe der Quadrate der Elemente).

replace

PROC replace (VECTOR VAR v, INT CONST i, REAL CONST r)
Zuweisung des i – ten Elementes von 'v' mit dem Wert von 'r'. Beispiel:

```
VECTOR VAR v :: ...;
    ...
replace (v, 13, 3.14);
(* Das 13. Element von 'v' bekommt den Wert '3.14' *)
```

FEHLER :

PROC replace : subscript overflow
Der Index 'i' liegt außerhalb des Vektors (i > LENGTH v).
PROC replace : subscript underflow
Der Index 'i' liegt außerhalb des Vektors (i < 1).

INITVECTOR PROC vector (INT CONST l)
Erzeugen eines Vektors mit 'l' Elementen. Ein INITVECTOR – Objekt benötigt nicht soviel Speicherplatz wie ein VECTOR – Objekt. Die Elemente werden mit dem Wert '0.0' initialisiert.

FEHLER :
PROC vector : size < = 0
Die angeforderte Elementanzahl 'l' muß > 0 sein.

INITVECTOR PROC vector (INT CONST l, REAL CONST value)
Erzeugen eines Vektors mit 'l' Elementen. Ein INITVECTOR – Objekt benötigt nicht soviel Speicherplatz wie ein VECTOR – Objekt. Die Elemente werden mit dem Wert 'value' initialisiert. Beispiel:

```
VECTOR VAR v := vector (17, 3.14159);
(* 'v' hat 17 Elemente mit den Wert '3.14159' *)
```

FEHLER :
PROC vector : size < = 0
Die angeforderte Elementanzahl 'l' muß > 0 sein.

6.1.4 MATRIX

Der Datentyp MATRIX erlaubt Operationen auf m x n Matrizen. Im Gegensatz zur Struktur 'ROW m ROW n REAL' muß die Anzahl der Elemente nicht zur Übersetzungszeit deklariert werden, sondern kann zur Laufzeit festgelegt werden. Somit kann eine zur Übersetzungszeit unbekannte Anzahl von REALs bearbeitet werden, wobei nur soviel Speicherplatz wie nötig verwendet wird. Die maximale Größe einer MATRIX beträgt 4000 Elemente.

Der in den Operationen ':=', 'idn' und 'matrix' benutzte Datentyp INITMATRIX wird nur intern gehalten. Er dient der Speicherplatzersparnis bei der Initialisierung.

– Operatoren		:= , = , <> , + , – , * COLUMNS , DET , INV , ROWS , TRANSP ,
– Eingabe/Ausgabe	:	get , put
– Besondere Matrix – Operationen	:	column , idn , matrix , row , sub transp , replace column , replace element , replace row

OP := (MATRIX VAR l, MATRIX CONST r)
Zuweisung von 'r' auf 'l'. Die MATRIX 'l' bekommt u.U. eine neue Anzahl von Elementen. Beispiel:

```
MATRIX VAR a :: matrix (3, 4, 0.0),
           b :: matrix (5, 5, 3.0);
                ...
a := b; (* 'a' hat jetzt 5 x 5 Elemente *)
```

OP := (MATRIX VAR l, INITMATRIX CONST r)
Dient zur Initialisierung einer Matrix. Beispiel:

```
MATRIX VAR x :: matrix (17, 4);
```

'matrix' erzeugt ein Objekt vom Datentyp INITMATRIX. Dieses Objekt braucht nicht soviel Speicherplatz wie ein MATRIX – Objekt. Dadurch wird vermieden, daß nach erfolgter Zuweisung nicht ein durch 'matrix' erzeugtes Objekt auf dem Heap unnötig Speicherplatz verbraucht.

BOOL OP = (MATRIX CONST l, r)
Vergleich zweier Matrizen. Der Operator '=' liefert FALSE, wenn die Anzahl Spalten oder Reihen der Matrizen 'l' und 'r' ungleich ist und wenn mindestens ein Element mit gleichen Indizes der zwei Matrizen ungleiche Werte haben. Beispiel:

```
MATRIX VAR a :: matrix (3, 3),
           b :: matrix (3, 3, 1.0),
           c :: matrix (4, 4);
      ... a = b ...
(* FALSE wegen ungleicher Werte   *)
      ... a = c ...
(* FALSE wegen ungleicher Groesse *)
      ... b = c ...
(* FALSE wegen ungleicher Groesse *)
```

BOOL OP <> (MATRIX CONST l, r)
Vergleich der Matrizen 'l' und 'r' auf Ungleichheit.

MATRIX OP + (MATRIX CONST m)
Monadisches '+'. Keine Auswirkungen.

MATRIX OP + (MATRIX CONST l, r)
Addition zweier Matrizen. Die Anzahl der Reihen und der Spalten muß gleich sein.
Beispiel:

```
MATRIX VAR a :: matrix (3, 43, 1.0),
           b :: matrix (3, 43, 2.0),
                summe;
summe := a + b;
(* Alle Elemente haben den Wert '3.0' *)
```

FEHLER:
MATRIX OP + : COLUMNS l <> COLUMNS r
Die Anzahl der Spalten von 'l' und 'r' sind nicht gleich.
MATRIX OP + : ROWS l <> ROWS r
Die Anzahl der Zeilen von 'l' und 'r' sind nicht gleich.

MATRIX OP – (MATRIX CONST m)
Monadisches Minus. Beispiel:

```
MATRIX VAR a :: matrix (3, 4, 10.0)
a := - a; (* Alle Elemente haben den Wert '- 10.0' *)
```

MATRIX OP – (MATRIX CONST l, r)
Subtraktion zweier Matrizen. Die Anzahl der Reihen und Spalten muß gleich sein.

FEHLER:
MATRIX OP – : COLUMNS l <> COLUMNS r
Die Anzahl der Spalten von 'l' und 'r' sind nicht gleich.
MATRIX OP – : ROWS l <> ROWS r
Die Anzahl der Zeilen von 'l' und 'r' sind nicht gleich.

MATRIX OP * (REAL CONST r, MATRIX CONST m)
Multiplikation einer Matrix 'm' mit einem Skalar 'r'. Beispiel:

```
MATRIX VAR a :: matrix (3, 4, 2.0);
     ...
a := 3 * a; (* Alle Elemente haben den Wert '6.0' *)
```

MATRIX OP * (MATRIX CONST m, REAL CONST r)
Multiplikation einer Matrix 'm' mit einem Skalar 'r'.

MATRIX OP * (MATRIX CONST l, r)
Multiplikation zweier Matrizen. Die Anzahl der Spalten von 'l' und die Anzahl der Zeilen von 'r' müssen gleich sein. Beispiel:

```
MATRIX VAR a :: matrix (3, 4, 2.0),
           b :: matrix (4, 2, 3.0),
           produkt;
produkt := a * b;
(* Alle Elemente haben den Wert '24.0' *)
```

FEHLER :
MATRIX OP * : COLUMNS l <> ROWS r
Die Anzahl der Spalten von 'l' muß mit der Anzahl der Zeilen von 'r' übereinstimmen.

VECTOR OP * (VECTOR CONST v, MATRIX CONST m)
Multiplikation des Vektors 'v' mit der Matrix 'm'.

FEHLER :
VECTOR OP * : LENGTH v <> ROWS m
Die Anzahl der Elemente von 'v' stimmt nicht mit den Anzahl der Zeilen von 'm' überein.

VECTOR OP * (MATRIX CONST m, VECTOR CONST v)
Multiplikation der Matrix 'm' mit dem Vektor 'v'.

FEHLER :
VECTOR OP * : COLUMNS m <> LENGTH v
Die Anzahl der Spalten von 'm' stimmt nicht mit der Anzahl der Elementen von 'v' überein.

COLUMNS

INT OP COLUMNS (MATRIX CONST m)
Liefert die Anzahl der Spalten von 'm'. Beispiel:

```
MATRIX VAR a :: matrix (3, 4),
           b :: matrix (7, 10);
put (COLUMNS a); (* 4 *)
put (COLUMNS b); (* 10 *)
```

DET

REAL OP DET (MATRIX CONST m)
Es wird der Wert der Determinanten von 'm' geliefert.

FEHLER :

OP DET : no square matrix
Die Matrix ist nicht quadratisch, d.h. ROWS m < > COLUMNS m

INV

MATRIX OP INV (MATRIX CONST m)
Liefert als Ergebnis die Inverse von 'm' (Achtung: starke Rundungsfehler möglich).

FEHLER:

OP INV : no square matrix
Die Matrix 'm' ist nicht quadratisch,
d.h. ROWS m < > COLUMNS m
OP INV : singular matrix
Die Matrix ist singulär.

ROWS

INT OP ROWS (MATRIX CONST m)
Liefert die Anzahl der Zeilen von 'm'. Beispiel:

```
MATRIX VAR a :: matrix (3, 4),
           b :: matrix (7, 10);
     ...
put (ROWS a); (* 3 *)
put (ROWS b); (* 7 *)
```

TRANSP

MATRIX OP TRANSP (MATRIX CONST m)
Liefert als Ergebnis die transponierte Matrix 'm'.

get

PROC get (MATRIX VAR m, INT CONST rows, columns)
Einlesen von Werten für die Matrix 'm' vom Terminal mit 'rows' – Zeilen und 'columns' – Spalten.

put

PROC put (MATRIX CONST m)
Ausgabe der Werte einer Matrix auf dem Terminal.

column

VECTOR PROC column (MATRIX CONST m, INT CONST i)
Die 'i'-te Spalte von 'm' wird als VECTOR mit 'ROWS m' Elementen geliefert.
Beispiel:

```
MATRIX CONST a :: matrix (3, 4);
VECTOR VAR b   :: column (a, 1);
(* 'b' hat drei Elemente mit den Werten '0.0' *)
```

FEHLER:
PROC column : subscript overflow
Der Index 'i' liegt außerhalb der Matrix 'm' (i > COLUMNS m).
PROC column : subscript underflow
Der Index 'i' liegt außerhalb der Matrix 'm' (i < 1).

idn

INITMATRIX PROC idn (INT CONST size)
Erzeugen einer Einheitsmatrix vom Datentyp INITMATRIX. Beispiel:

```
MATRIX VAR a :: idn (10);
(* Erzeugt eine Matrix mit 10 x 10 Elementen, deren
 Werte '0.0' sind, mit der Ausnahme der Diagonalele-
 mente, die den Wert '1.0' haben.*)
```

FEHLER :
PROC idn : size <= 0
Die angeforderte 'size' Anzahl Spalten oder Zeilen muß > 0 sein.

matrix

INITMATRIX PROC matrix (INT CONST rows, columns)
Erzeugen eines Datenobjekts vom Datentyp INITMATRIX mit 'rows' Zeilen und 'columns' Spalten. Alle Elemente werden mit dem Wert '0.0' initialisiert. Beispiel:

```
MATRIX CONST :: matrix (3, 3);
```

FEHLER:
PROC matrix : rows < = 0
Die angeforderte Zeilenanzahl 'rows' muß > 0 sein.
PROC matrix : columns < = 0
Die angeforderte Spaltenanzahl 'columns' muß > 0 sein.

INITMATRIX PROC matrix (INT CONST rows, columns, REAL CONST value)
Erzeugen eines Datenobjekts vom Datentyp MATRIX mit 'rows' Zeilen und 'columns' Spalten. Alle Elemente der erzeugten MATRIX werden mit dem Wert 'value' initialisiert. Beispiel:

```
MATRIX CONST :: matrix (3, 3, 3.14);
```

FEHLER:
PROC matrix : rows < = 0
Die angeforderte Zeilenanzahl 'rows' muß > 0 sein.
PROC matrix : columns < = 0
Die angeforderte Spaltenanzahl 'columns' muß > 0 sein.

row

VECTOR PROC row (MATRIX CONST m, INT CONST i)
Die 'i' – te Reihe von 'm' wird als VECTOR mit 'COLUMNS m' Elementen geliefert. Beispiel:

```
MATRIX CONST a :: matrix (3, 4);
VECTOR VAR b   :: row (a, 1);
(* 'b' hat vier Elemente mit den Werten '0.0'*)
```

FEHLER:
PROC row : subscript overflow
Der Index 'i' liegt außerhalb der Matrix 'm' (i > ROWS m).
PROC row : subscript underflow
Der Index 'i' liegt außerhalb der Matrix 'm' (i < 1).

sub

REAL PROC sub (MATRIX CONST m, INT CONST row, column)
Liefert den Wert eines Elementes von 'm', welches durch die Indizes 'row' und 'column' bestimmt wird. Beispiel:

```
MATRIX VAR m :: matrix (5, 10, 1.0);
put (sub (m, 3, 7));
```

FEHLER:

PROC sub : row subscript overflow
Der Index 'row' liegt außerhalb von 'm' (row > ROWS m).
PROC sub : row subscript underflow
Der Index 'row' liegt außerhalb von 'm' (row < 1).
PROC sub : column subscript overflow
Der Index 'column' liegt außerhalb von 'm' (column > ROWS m).
PROC sub : row subscript underflow
Der Index 'column' liegt außerhalb von 'm' (column < 1).

transp

PROC transp (MATRIX VAR m)
Transponieren der Matrix 'm', wobei kaum zusätzlicher Speicherplatz benötigt wird.

replace column

PROC replace column (MATRIX VAR m, INT CONST column index, VECTOR CONST column value)

Ersetzung der durch 'column index' definierten Spalte in der MATRIX 'm' durch den VECTOR 'column value'. Beispiel:

```
MATRIX VAR a :: matrix (3, 5, 1.0);
VECTOR VAR b :: vector (3, 2.0);
      ...
replace column (a, 2, b);
(* Die zweite Spalte von 'a' wird durch die Werte von
   'b' ersetzt *)
```

FEHLER:

PROC replace column : LENGTH columnvalue <> ROWS m
Die Anzahl der Zeilen der MATRIX 'm' stimmt nicht mit der Anzahl der Elemente von 'columnvalue' überein.
PROC replace column : column subscript overflow
Der Index 'columnindex' liegt außerhalb von 'm'
(columnindex > COLUMNS m).
PROC sub : column subscript underflow
Der Index 'columnindex' liegt außerhalb von 'm' (columnindex < 1).

replace element

PROC replace element (MATRIX VAR m , INT CONST row, column, REAL CONST value)

Ersetzung eines Elementes von 'm' in der 'row'-ten Zeile und 'column'-ten Spalte durch den Wert 'value'. Beispiel:

```
MATRIX VAR a :: matrix (5, 5);
      ...
replace element (1, 1, 3.14159);
```

FEHLER:

PROC replace element : row subscript overflow
Der Index 'row' liegt außerhalb von 'm' (row > ROWS m).
PROC replace element : row subscript underflow
Der Index 'row' liegt außerhalb von 'm' (row < 1).
PROC replace element : column subscript overflow
Der Index 'column' liegt außerhalb von 'm' (column > COLUMNS m).
PROC replace element : row subscript underflow
Der Index 'column' liegt außerhalb von 'm' (column < 1).

replace row

PROC replace row (MATRIX VAR m, INT CONST rowindex,
VECTOR CONST rowvalue)

Ersetzung der Reihe 'rowindex' in der MATRIX 'm' durch den VECTOR 'rowvalue'.
Beispiel:

```
MATRIX VAR a :: matrix (3, 5, 1.0);
VECTOR VAR b :: vector (5, 2.0);
      ...
replace row (a, 2, b);
(* Die 2. Reihe von 'a' wird durch Werte von 'b'ersetzt *)
```

FEHLER:

PROC replace row : LENGTH rowvalue <> COLUMNS m
Die Anzahl der Spalten der MATRIX 'm' stimmt nicht mit der Anzahl der Elemente von 'rowvalue' überein.
PROC replace row : row subscript overflow
Der Index 'rowindex' liegt außerhalb von 'm'
(rowindex > ROWS m).
PROC sub : row subscript underflow
Der Index 'rowindex' liegt außerhalb von 'm' (rowindex < 1).

6.2 Programmanalyse

Das Packet 'reporter' ermöglicht:

a) Ablaufinformationen ("trace");
b) Häufigkeitszählung ("frequency count");
c) Programmunterbrechung bei Nichterfüllung einer Bedingung ("assertion").

Installation

Das Programm befindet sich in der Datei 'reporter' und kann wie üblich insertiert werden. Jedoch muß es mit 'check off' übersetzt werden, damit keine Zeilennummern für 'reporter' generiert werden. Dies ist notwendig, damit die Zeilennummern des zu testenden Programms nicht mit den Zeilennummern des Programms 'reporter' verwechselt werden können. Beispiel:

```
check off; insert ("reporter"); check on
```

Mit dem Kommando

```
generate reports ("testdatei")
```

werden die oben erwähnten Prozeduraufrufe ('report') in das zu testende Programm, welches in der Datei 'testdatei' steht, geschrieben. Die Prozeduraufrufe werden nach jedem Prozedur–, Operator– oder Refinement–Kopf eingefügt und erhalten den entsprechenden Namen als Parameter. Diese Prozeduraufrufe werden gekennzeichnet, damit sie von der Prozedur

```
eliminate reports ("testdatei")
```

automatisch wieder entfernt werden können. Beispiel (für die eingefügten Prozeduraufrufe):

```
...
PROC beispiel (INT CONST mist):
##report ("PROC beispiel");##
...
```

Automatische Ablaufinformation

Ist ein Programm mit 'generate reports' mit 'report' – Aufrufen versehen worden, kann es wie gewohnt übersetzt werden. Wird das Programm vom ELAN – Compiler korrekt übersetzt und dann gestartet, wird bei jedem Antreffen eines 'report' – Aufrufs der Parameter (Name der Prozedur, Operator oder Refinement) in eine Datei, die TRACE – Datei geschrieben. Die TRACE – Datei wird beim Programmlauf automatisch von 'reporter' unter dem Namen 'TRACE' eingerichtet.

Mit Hilfe dieser Datei kann der Programmablauf verfolgt werden. Es ist damit auch möglich festzustellen, wo eine "Endlos – Rekursion" auftritt. Die Ablaufinformationen bestehen nur aus den Namen der angetroffenen Prozeduren und Refinements. Trotzdem können die Anzahl der Informationen sehr umfangreich werden. Deshalb gibt es die Möglichkeit, die Erzeugung der Ablaufinformationen ab – bzw. wieder anzuschalten. Dazu gibt es die Möglichkeit, in das zu testende Programm die Prozeduren

```
report on
report off
```

einzufügen und das zu testende Programm mit diesen Prozeduraufrufen (erneut) zu übersetzen.

Benutzereigene Ablaufinformation

Zusätzlich zu den von 'generate reports' eingefügten 'report' – Aufrufen kann ein Benutzer eigene Aufrufe an geeigneten Stellen in ein Programm schreiben. Dafür werden weitere 'report' – Prozeduren zur Verfügung gestellt, die als ersten Parameter ein TEXT – Objekt (meist Name des Objekts oder der Ausdruck selbst) und als zweiten ein INT/REAL/TEXT/ BOOL – Objekt (der zu überprüfende Wert oder Ausdruck) enthalten. Beispiel:

```
...
PROC beispiel (INT CONST mist):
##report ("beispiel");##      (* automatisch eingefuegte *)
INT VAR mist :: ...; ...
##report ("mist:", mist);##   (* vom Benutzer per Hand einge-
                                fuegt *)
...
```

Folgende 'report' – Routinen stehen zur Verfügung, damit man sie "von Hand" in ein zu testendes Programm einfügen kann:

```
PROC report on
PROC report off
PROC report (TEXT CONST message)
PROC report (TEXT CONST message,  INT CONST value)
PROC report (TEXT CONST message, REAL CONST value)
PROC report (TEXT CONST message, TEXT CONST value)
PROC report (TEXT CONST message, BOOL CONST value)
```

Wichtig: Hier – wie bei allen anderen "von Hand eingefügten" Aufrufen – sollte ein Nutzer sich an die Konvention halten, diese in "##" einzuklammern. Mit 'eliminate reports' werden diese Einfügungen automatisch entfernt. Sollen diese Aufrufe aber immer im Programm erhalten bleiben (jedoch nicht wirksam sein), sollten sie

a) vor 'generate reports' – Aufruf mit jeweils '###' eingefaßt werden. Beispiel:
   ### report ("...") ###
   So steht das 'report' – Statement in einem Kommentar. 'generate reports' wandelt '###' – – > '####' um, so daß ein solches Statement wirksam wird. 'eliminate reports' wandelt ein '####' – – > '###' zurück.

b) nach 'generate reports' in '####' eingefaßt werden.

Häufigkeitszählung

Eine Häufigkeitszählung erhält man, in dem man in das zu testende Programm die Aufrufe

```
count on
count off
```

einfügt. Ist die Häufigkeitszählung eingeschaltet, merkt sich 'reporter' die Anzahl der Durchläufe für jede Prozedur bzw. Refinement. Mit der Prozedur

```
generate counts ("zu testende datei")
```

werden die vermerkten Häufigkeiten in das zu testende Programm direkt eingefügt. Die Häufigkeiten werden wie oben beschrieben gekennzeichnet, so daß sie mit 'eliminate reports' entfernt werden können.

Assertions

Zusätzlich zu den oben erwähnten Möglichkeiten bietet 'reporter' noch die Prozedur

```
assert
```

an. Diese Prozedur kann von einem Programmierer an einer Stelle in das zu testende Programm eingefügt werden, an der bestimmte Bedingungen erfüllt sein müssen. Die Prozedur 'assert' steht in zwei Formen zur Verfügung:

```
PROC assert (BOOL CONST zusicherung)
PROC assert (TEXT CONST message, BOOL CONST zusicherung)
```

Ist der Wert von 'zusicherung' nicht TRUE, wird der Programmlauf abgebrochen.

reporter – Kommandos

count on

PROC count on
Schaltet die Häufigkeitszählung ein.

count off

PROC count off
Schaltet die Häufigkeitszählung aus.

eliminate reports

PROC eliminate reports (TEXT CONST datei)
Entfernt gekennzeichnete 'report' – Aufrufe aus der Datei 'datei'.

generate reports

PROC generate reports (TEXT CONST datei)
Fügt 'report' – Aufrufe in die Datei 'datei' ein und kennzeichnet diese mit '##'.

report on

PROC report on
Schaltet die Ablaufinformationen in die Datei 'TRACE' ein.

report off

PROC report off
Schaltet die Ablaufinformationen wieder aus.

generate counts

PROC generate counts (TEXT CONST datei)
Bringt die Häufigkeitszählung (wie oft eine Prozedur oder Refinement durchlaufen wurde) in die Programmdatei 'datei'. Mit 'eliminate reports' werden diese wieder automatisch entfernt.

assert

PROC assert (TEXT CONST message, BOOL CONST value)
Schreibt 'message' und den Wert von 'value' in die TRACE – Datei. Ist 'value' FALSE, wird angefragt, ob das Programm fortgesetzt werden soll.

Referencer

'referencer' wird durch

```
referencer ("ref datei", "referenz liste")
```

aufgerufen, wobei die Datei 'referenz liste' nicht existieren darf. 'referenz liste' enthält nach Ablauf des Programms die gewünschte Liste, die sogenannte Referenzliste.

Achtung: 'referencer' arbeitet ausschließlich mit Namen und verarbeitet nur wenige syntaktische Konstrukte. Darum ist es nur erlaubt, ein PACKET auf einmal von 'referencer' verarbeiten zu lassen. Verarbeitet man mehrere PACKETs auf einmal, kann es geschehen, daß gleichnamige Objekte in unterschiedlichen Paketen zu Warnungen (vergl. die unten beschriebenen Überprüfungen) führen.

In der Referenzliste sind

- alle Objekte mit ihrem Namen (in der Reihenfolge ihres Auftretens im Programm)
- alle Zeilennummern, in der das Objekt angesprochen wird
- die Zeilennummern, in der das Objekt deklariert wurde ('L' für ein lokales und 'G' für ein globales Objekt, 'R' für ein Refinement)

verzeichnet.

Die Referenzliste kann u.a. dazu dienen, zu kontrollieren, ob und wie (bzw. wo) ein Objekt angesprochen wird. Dies lohnt sich selbstverständlich nur bei etwas umfangreicheren Programmen (bei "Mini" – Programmen kann man dies sofort sehen).

Bei der Erstellung der Referenzliste nimmt das Programm 'referencer' gleichzeitig einige Überprüfungen vor, die helfen können, ein Programm zu verbessern:

1. Warnung bei mehrzeiligen Kommentaren.

2. Überdeckungsfehler. Wird ein Objekt global (auf PACKET – Ebene) und nochmals lokal in einer Prozedur deklariert, ist das globale Objekt nicht mehr ansprechbar. Überdeckungen sind nach der gültigen Sprachdefinition z.Zt. noch erlaubt, werden aber bei einer Revision des Sprachstandards verboten sein.

3. Mehrmaliges Einsetzen von Refinements. Wird ein Refinement mehrmals eingesetzt (das ist völlig legal), sollte man überlegen, ob sich dieses Refinement nicht zu einer Prozedur umgestalten läßt.

4. Nicht angewandte Refinements. Wird ein Refinement zwar deklariert, aber nicht "aufgerufen", erfolgt eine Warnung.

5. Nicht angesprochene Daten – Objekte. Werden Daten – Objekte zwar deklariert, aber im folgenden nicht angesprochen, wird eine Warnung ausgegeben. Hinweis: Alle Objekte, die nur wenig angesprochen werden, also nur wenige Zeilennummern in der Referenzliste besitzen, sind verdächtig (Ausnahmen: importierte Prozeduren, LET – Objekte u.a.m.).

referencer – Kommandos

referencer

PROC referencer (TEXT CONST check file, dump file)
Überprüft 'check file'. In 'dump file' steht nach Abschluß die Referenzliste.

6.3 Rechnen im Editor

Das Programm TeCal ermöglicht einfache Rechnungen (ähnlich wie mit einem Taschenrechner) unter der Benutzung des Editors. Gleichzeitig stehen dem Benutzer aber alle Fähigkeiten des Editors zur Verfügung. TeCal ermöglicht Rechnungen auf einfache Weise zu erstellen oder Tabellenspalten zu berechnen. Zur Benutzung müssen 'TeCal' und 'TeCal Auskunft' insertiert werden.

TeCal wird aus dem Editor heraus durch 'ESC t' oder durch das Editor – Kommando

```
tecal
```

aktiviert. Dadurch wird in der untersten Zeile des Bildschirms eine Informationszeile aufgebaut, in der die (Zwischen –) Ergebnisse einer Rechnung zur Kontrolle festgehalten werden.

Arbeitsweise

Wenn TeCal insertiert ist, kann die Taschenrechnerfunktion jederzeit durch ESC t aufgerufen werden. Aus der editierten Datei werden Werte mit ESC L gelesen, durch ESC + (bzw. –,*,/) verknüpft und mit ESC S an die aktuelle Cursorposition geschrieben werden.

Der von TeCal errechnete Wert wird durch ESC S derart ausgegeben, daß an der Stelle an der der Cursor steht die letzte Stelle vor dem Dezimalpunkt geschrieben wird.

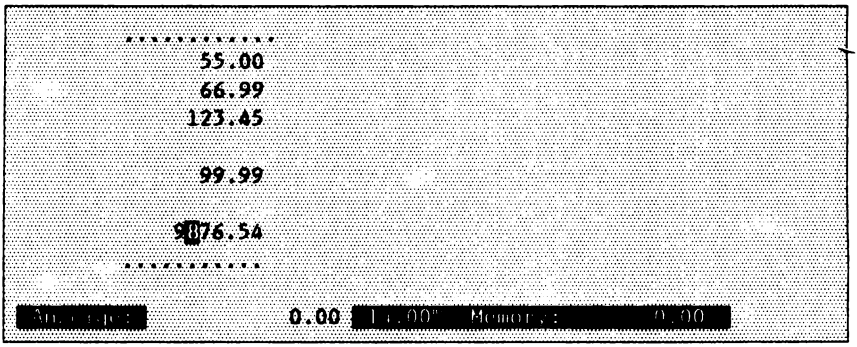
Die Eingabe von Klammern geschieht durch ESC (ESC).

Durch die Hilfsfunktion ESC ? lassen sich die TeCal Funktionen auflisten.

Der Prozentoperator ESC % erlaubt einfache Rechnungen der Form: 'zahl' ESC + ESC % ESC = .

Derartige Folgen können natürlich mit der bekannten Editor – Lernfunktion auch gelernt werden, so daß sich z.B. die Mehrwertsteuerberechnung auf wenige Tasten reduziert.

Spalten können summiert werden, indem auf der untersten Zahl einer Spalte **ESC V** eingegeben wird. Daraufhin werden alle darüberliegende Zahlen addiert, bis ein Zeichen auftritt, das nicht in einer Zahl auftritt (Leerzeichen stören nicht!).



TeCal Prozeduren

evaluate

evaluate (TEXT CONST zeile, INT CONST von)
Ausdruck 'zeile' ab der Stelle 'von' berechnen.

evaluate (TEXT CONST zeile)
Ausdruck 'zeile' ab Stelle 1 berechnen.

kommastellen

kommastellen (INT CONST stellen)
Berechnungen auf 'stellen' Zahlen hinter dem Komma einstellen.

merke

PROC merke (INT CONST zahl)
Integer 'zahl' im Merkregister abspeichern.

PROC merke (REAL CONST zahl)
Real 'zahl' im Merkregister abspeichern.

prozentsatz

PROC prozentsatz (INT CONST zahl)
Prozentsatz von 'zahl' Prozent einstellen. Der Wert wird automatisch konvertiert.

PROC prozentsatz (REAL CONST zahl)
Prozentsatz von 'zahl' Prozent einstellen.

tecal

PROC tecal (FILE VAR f)
Datei 'f', die mit 'sequential file' assoziiert ist, mit TeCal editieren.

PROC tecal (TEXT VAR datei)
'datei' mit TeCal editieren.

PROC tecal
Zuletzt editierte Datei mit TeCal editieren.

tecalauskunft

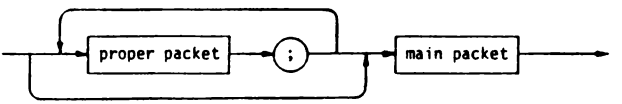
PROC tecalauskunft
Auskunft zeigen.

PROC tecalauskunft (TEXT CONST zeichen)
Auskunft zu 'zeichen' zeigen.

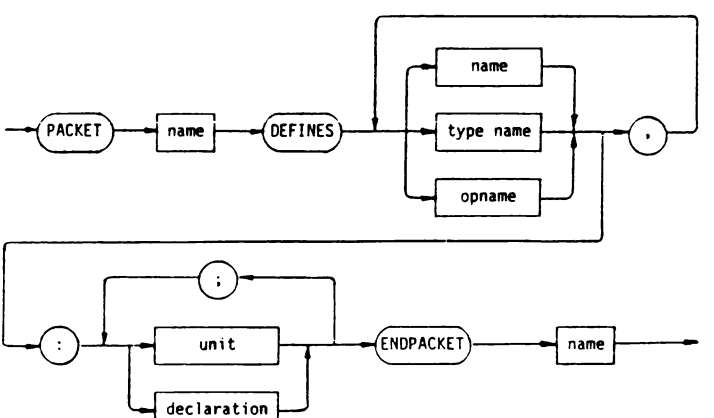
ELAN – Syntaxdiagramme

Die ELAN – Syntaxdiagramme wurden innerhalb des Forschungsprojekts Schulsprache im Fachgebiet Softwaretechnik, Institut für angewandte Informatik an der Technischen Universität Berlin erstellt.

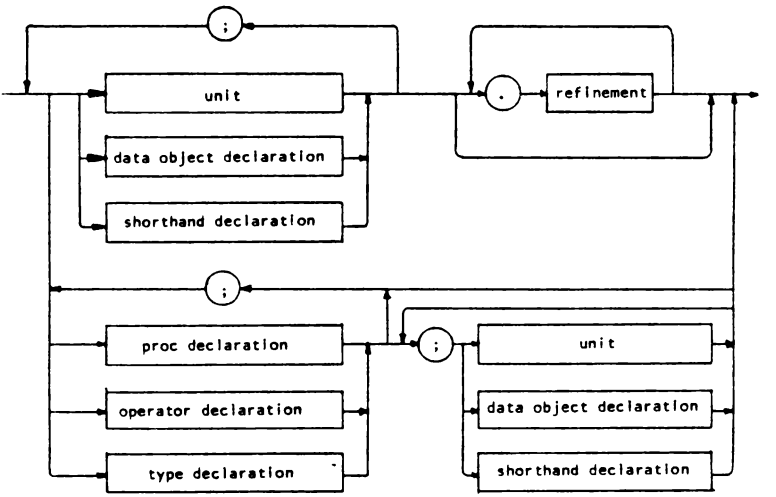
program



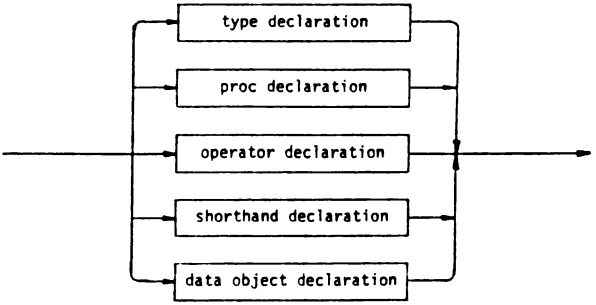
proper packet



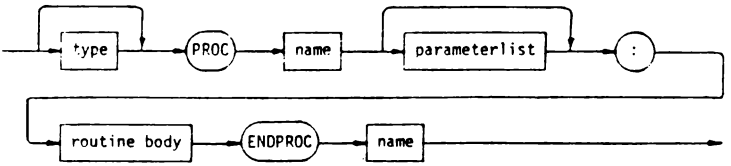
main packet


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

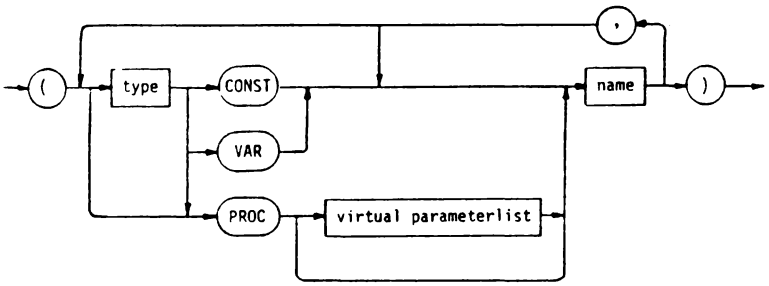
declaration



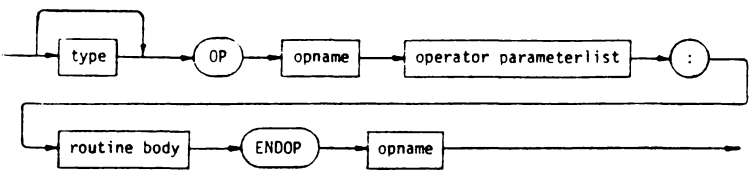
proc declaration



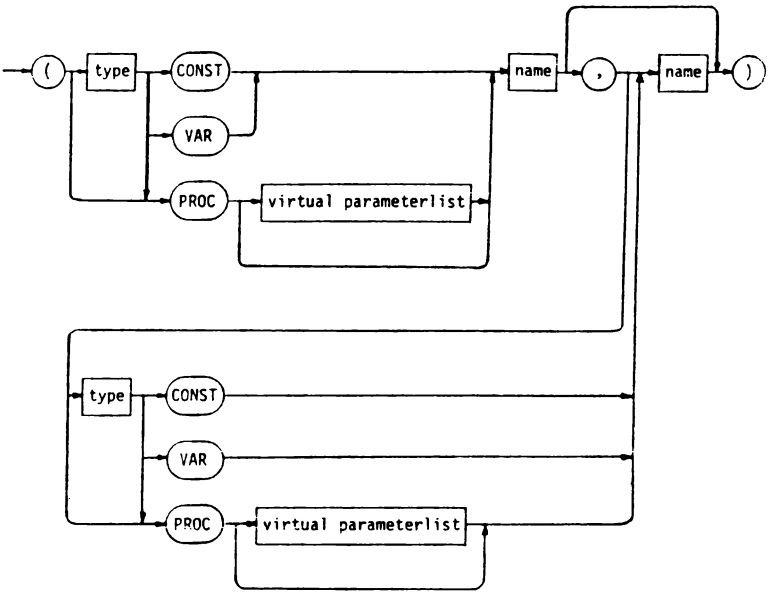
parameter list



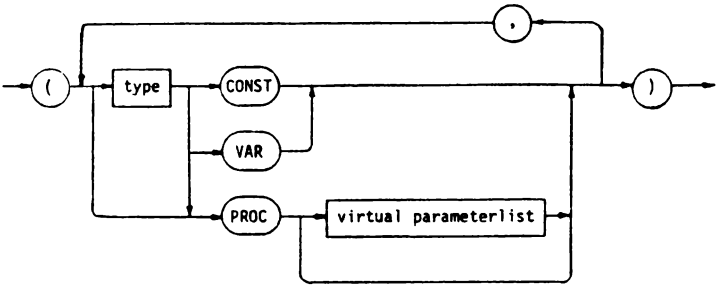
operator declaration



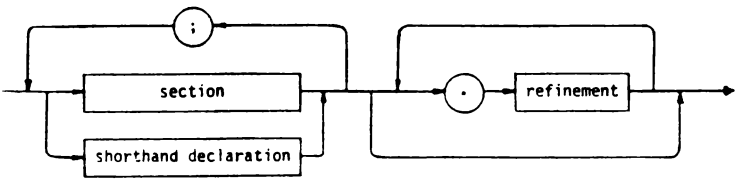
operator parameter list



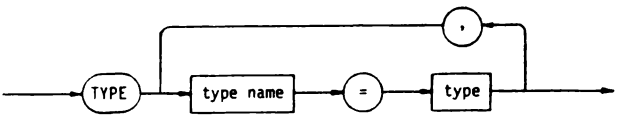
virtual parameterlist



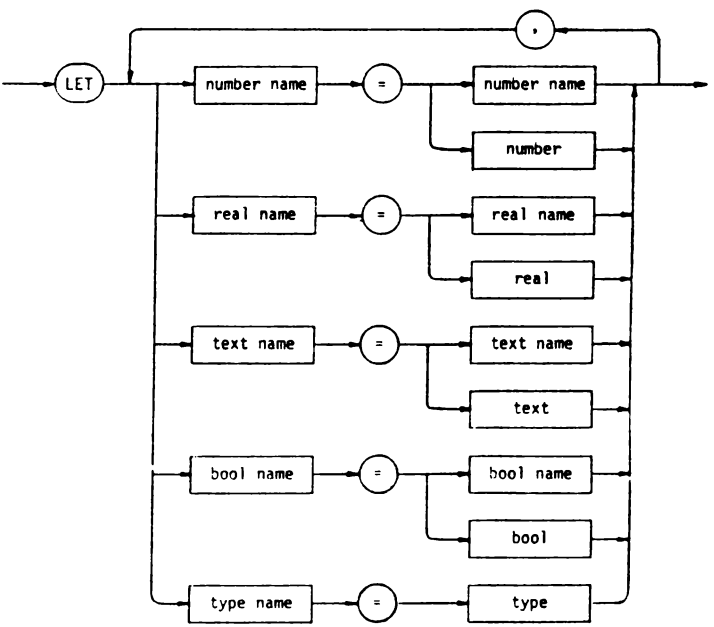
routine body



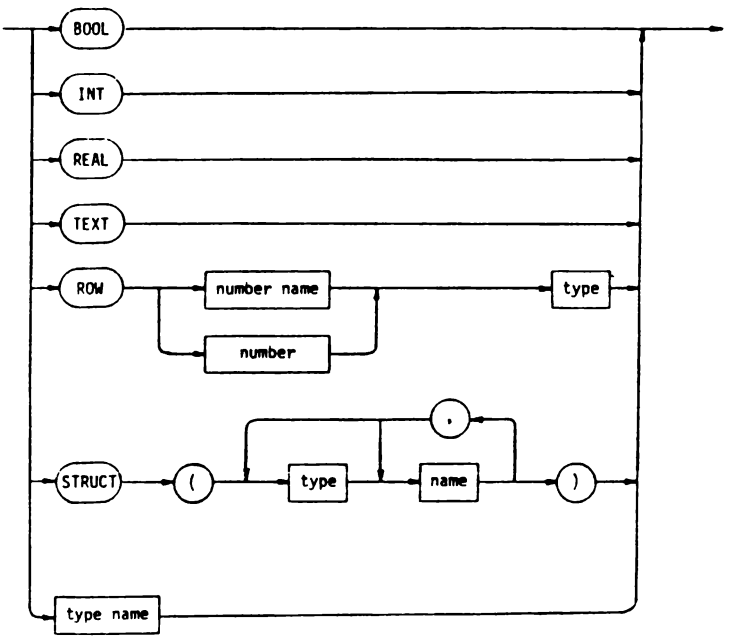
type declaration



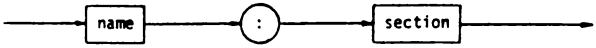
shorthand declaration



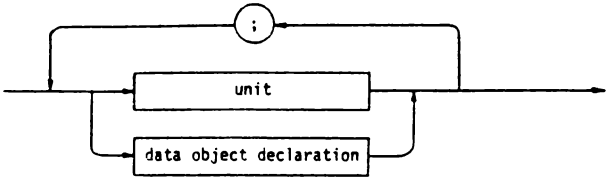
type


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

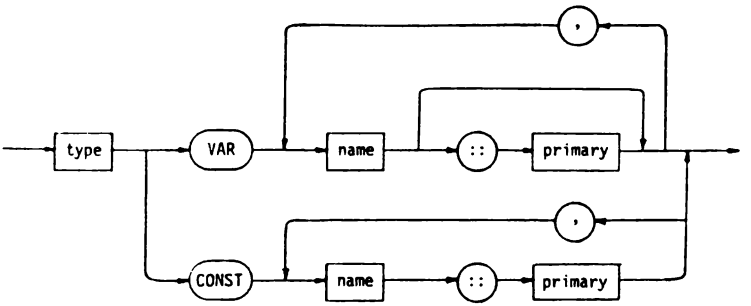
refinement



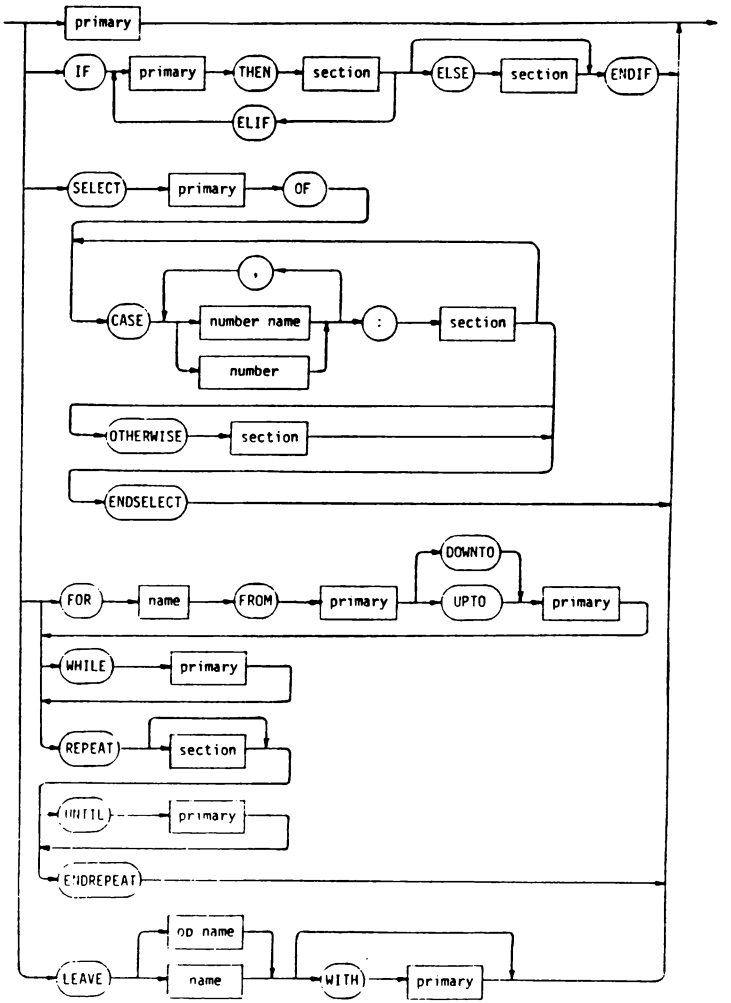
section



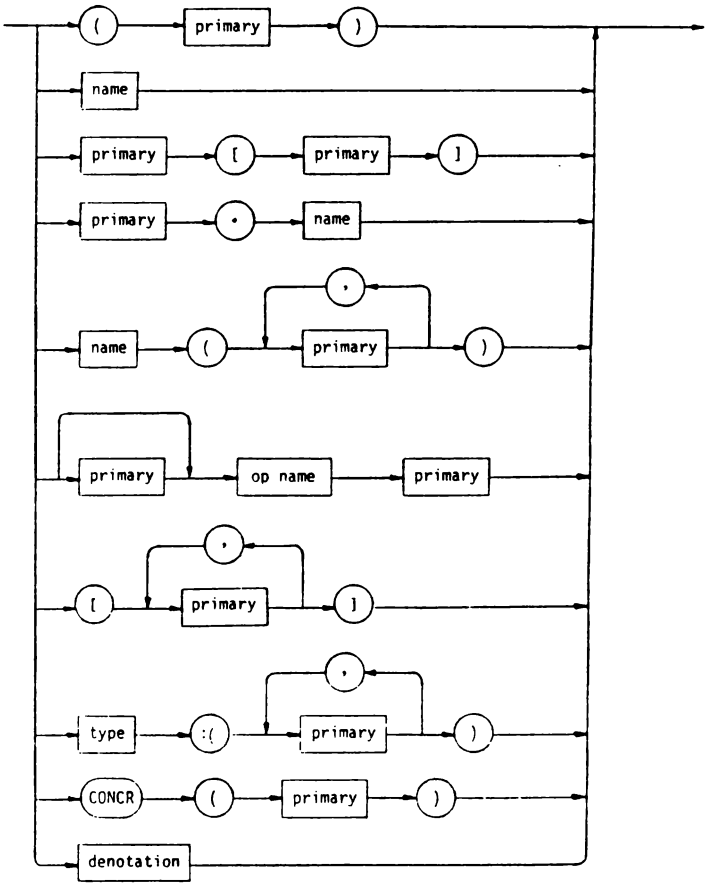
data object declaration



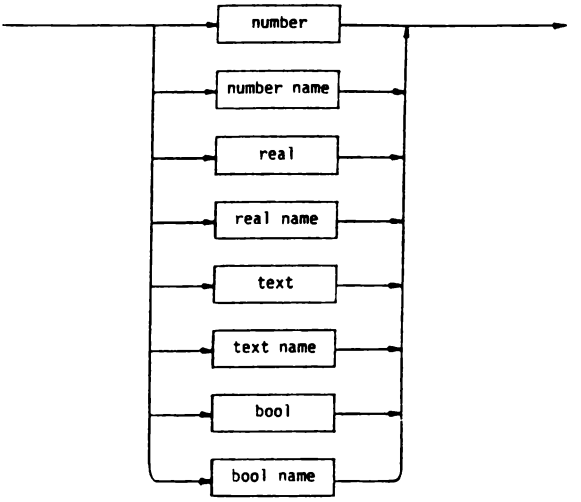
unit


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
OD name
LEAVE
name
WITH
primary

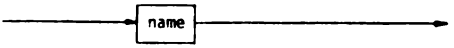
primary


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

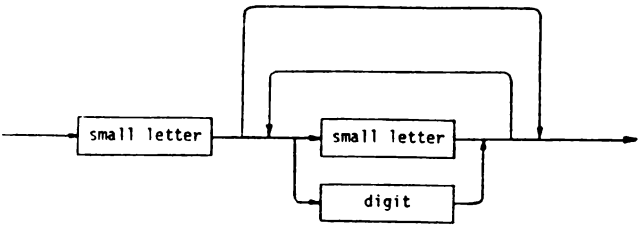
denotation


number
number name
real
real name
text
text name
bool
bool name

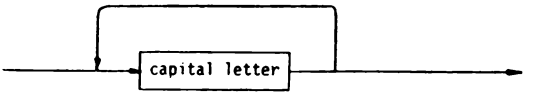
number name, real name, text name, bool name



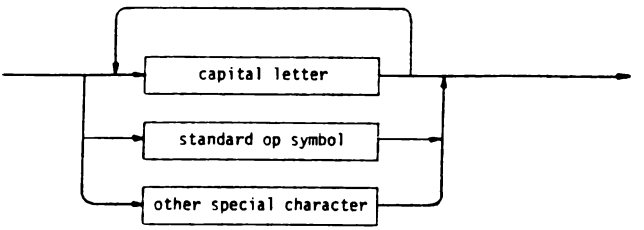
name



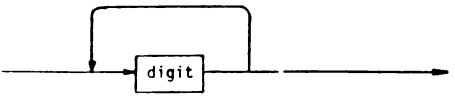
type name



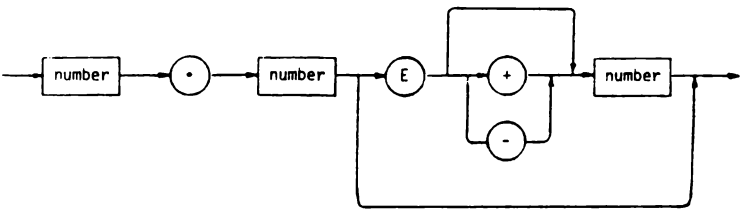
op name



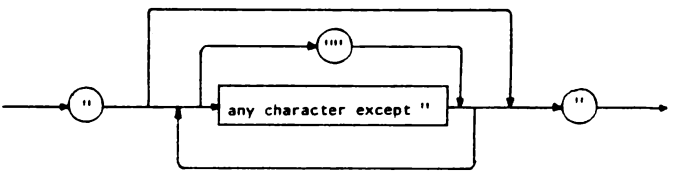
number



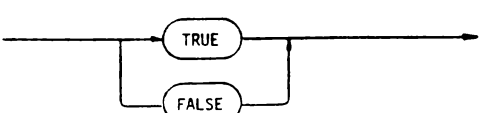
real



text



bool



INDEX