RUHR-UNIVERSITÄT BOCHUM

ARBEITSBERICHTE

des

Rechenzentrums

Direktor: Prof. Dr. H. Ehlich

Nr. 7603 BO.E2.06

BOGOL-TAS, eine Spracherweiterung von ALGOL 60 durch Codeprozeduren zur Systemprogrammierung

von

Manfred Rosendahl

Mai 1976

463 Bochum, Universitätsstr. 150, Geb. NA

Der vorliegende Bericht beschreibt eine Methode zur Erweiterung der Sprache ALGOL60. Mit Hilfe dieser Erweiterungen kann ALGOL60 zur Implementierung von Programmiersystem-Software benutzt werden. So sind mit Hilfe von BOGOL-TAS bisher folgende weitere Programmpakete entstanden:

BOGOL-STRING [4]: Ein Programmsystem, das in ALGOL die Möglichkeiten von SNOBOL [12] und CDL [13, 14] zur Textmanipulation verfügbar macht.

BOGOL-DATA [5] : Ein Programmpaket, das es in ALGOL60 erlaubt ALGOL68 Datenstrukturen [15, 16] zu definieren und zu bearbeiten.

Des weiteren ist ein Programmpaket zur symbolischen Formelmanipulation geplant.

INHALTSVERZEICHNIS:

1. Einleitung

Für manche Probleme, sowohl in der Anwendungs- als auch in der Systemprogrammierung wünscht man bei ALGOL60 [1] einige zusätzliche Eigenschaften. Unter anderem:

- Erweiterung Datenstruktur (Listen, Verweise)
- Adressrechnung
- Funktionsprozeduren, die auch nicht skalare Werte und Variablen als Funktionswert übergeben.
- Prozeduren, deren Parameterzahl und Parameterbedeutung von Aufruf zu Aufruf verschieden sein kann.

Beispiele (noch nicht in ALGOL60 Notation):

```
var procedure MAX(X); code; real A,B,C; MAX(A,B,C):= 0;
```

Die Variable mit dem größten Wert wird auf Null gesetzt. Hier ist die Anzahl der Parameter variabel und der Funktionswert kann wie eine Variable behandelt werden.

```
string procedure CAT(X); code; array A,B,C [0:n]; A:=CAT(B,C);
```

Die Strings B und C werden verkettet und ergeben den String A.

Hier ist die Parameterzahl wiederum variabel; der Funktionswert ist eine nicht skalare Größe, hier ein Feld variabler Länge.

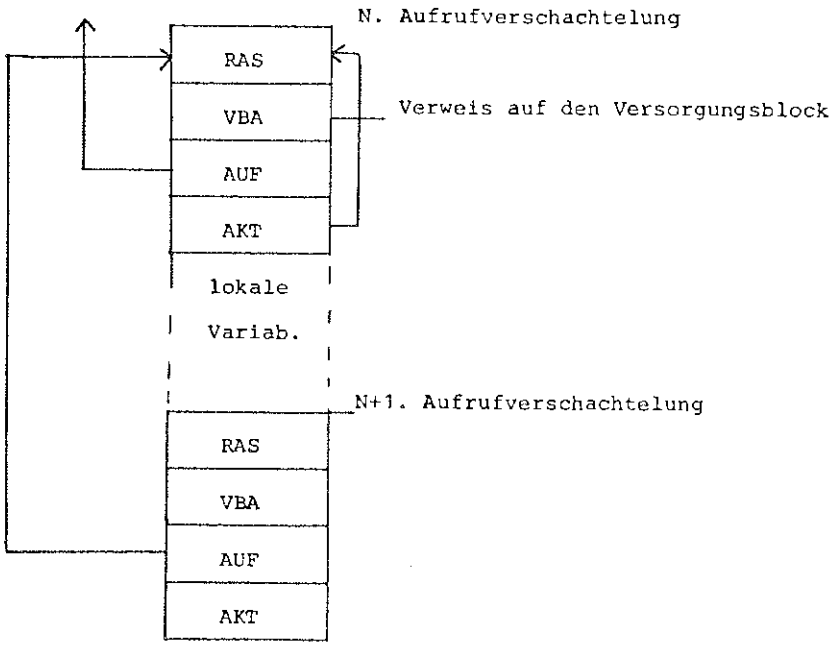
Um solche Möglichkeiten in ALGOL60 ohne Compileränderung zu erhalten, muß man die Erweiterbarkeit von ALGOL60 ausnützen. Bei vollständigen Implementierungen ist dies durch die Möglichkeit der getrennt übersetzbaren Prozeduren gegeben. Bei diesen Prozeduren kann vom Compiler kein Zusammenhang zwischen den aktuellen Parametern im Prozeduraufruf und den formalen Parametern in der Prozedurdeklaration festgestellt werden.

Auf der anderen Seite benötigt ALGOL60 als Programmiersprache mit rekursiven Prozeduren ein Laufzeitsystem mit einem Keller, der bei jeder Prozedurverschachtelung nach einheitlichen Regeln weitergeführt wird. Dies erlaubt es in einer Prozedur an alle Größen, insbesondere auch Parameter, der aktuellen Aufrufverschachtelung zu gelangen. Dadurch kann die Parameteraktivierung und die Übergabe der Funktionswerte durch Code-Prozeduren realisiert werden.

2. Vorbemerkungen

2.1. Kurzer Überblick über das ALGOL60 Laufzeitsystem.

Vom ALGOL60 Laufzeitsystem [3] wird ein Keller aufgebaut, der für jede neue Aufrufhierachie folgende Eintragungen enthält:



RAS enthält die Rückkehradresse. AUF weist auf die Basisadresse der vorletzten, der Aufrufhierachie. AKT weist auf die Basisadresse der aktuellen Hierachie. VBA weist auf den Versorgungsblock des die aktuelle Hierachie erzeugenden Prozeduraufrufs. Dieser enthält die Information über die aktuellen Parameter in folgender Form:

FA	PZ
HV1	ZV1
⋮	⋮
HVn	ZVn

FA Fehleradresse
PZ Parameterzahl
HVi Hauptversorgung des i. Parameters
HVi Zusatzversorgung des i. Parameters

Führt man den in der Zelle HVi stehenden Befehl aus, so wird der i. Parameter aktiviert. Die diesen Parameter kennzeichnende Information wird in Registern übergeben (z. B. Wert, Adresse, Startadresse einer Prozedur, Adresse einer Feldbeschreibung). ZVi enthält Information über den Typ des i. Parameters.

Durch eine Verweiskette über AUF auf die Basis der Aufrufhierachie und dann über deren VBA gelangt man auch an den Versorgungsblock der übergeordneten Hierachie. Man kann also in einer Prozedur durch eine Code-Prozedur an den Versorgungsblock der Prozedur gelangen und so die Parameterzahl und die Parametertypen ermitteln und auch die Parameter aktivieren.

2.2. Deklaration und Handhabung

Die benutzten Prozeduren werden als getrennt übersetzte (Code-)Prozeduren deklariert. Dabei muß lediglich der Prozedurtyp und, ob mit oder ohne Parameter, angegeben werden. Die Deklaration hat dann die Form:

[real / integer / boolean] procedure ⟨Name⟩ [(X)]; code;

Die Deklaration bei der Benutzung und die bei der Übersetzung braucht nicht übereinzustimmen. Die hier angegebenen Prozeduren liegen in der Bibliothek BOGOL und sind nach □BIBANM., BOGOL bzw. □BIBVERL., BOGOL, LFD, -STD-,,-STD- verfügbar.

Die im folgenden definierten Parametertypen dienen nur zur Beschreibung und haben keine Beziehung zur Deklaration im Programm.

var	Eine Größe, die einen Wert und eine Adresse hat, z. B. Variable, Feldelement oder Aufruf einer Funktionsprozedur mit Parameter falls dieser auch eine Adresse liefert.
pvar	Wie var zusätzlich noch Feld = 1. Feldelement und Funktionsprozedur ohne Parameter = Funktionswert, falls auch eine Adresse geliefert wird.
type	ein Wert mit beliebiger Typenkennung.
ref	Adresse
real	ein Wert mit Typenkennung 0, Zahl in ALGOL60.
integer	ein Wert mit Typenkennung, der ganzzahlig ist.
array	Feldname
proc	Name einer Prozedur
any	beliebiger Parametertyp, z. T. keine Marke jedoch integer label.
fix	eine Festkomma-Zahl.
string	eine pvar Größe, die einen Stringanfang (Kopfwort) enthält.

type I — ein Wert mit Typenkennung I.

number — ein Wert mit Typenkennung 0 oder 1 (Zahl).

ftype — ein Wert mit Typenkennung = 1.

2.2.1. Einsetzen der Deklarationen: PREKOM

Um das Deklarieren der in einem BOGOL-Programm oder auch generell in einem ALGOL-Programm vorkommenden Code-Prozeduren zu ersparen, wurde ein Precompiler PREKOM geschrieben, der fehlende Deklarationen einsetzt. Der Operator PREKOM befindet sich in der Bibliothek QUEMOD. Durch □STARTE,PREKOM wird ein Kommando PREKOM folgender Form definiert:

```
PREKOM
1 ZIEL          =  -
2 QUELLE        =  -
-------------------------------------------
3 BEREICH
4 DEKLARATION   =  - NORMAL
5 NUMERIERUNG   =  -
6 PROTOKOLL     =  -
7 ZUSATZ        =  -
```

Das in der QUELLE befindliche ALGOL-Programm wird einschließlich der hinzugefügten Deklarationen im ZIEL abgelegt. Dem Precompiler sind die in DEKLARATION stehenden Prozedurnamen bekannt. Bei voreingestellt - NORMAL - sind die ALGOL-Standardprozeduren bekannt und werden nicht deklariert. Bei DEKLARATION= -BOGOL- sind dieBOGOL-Prozeduren bekannt. Dann werden die EA-Prozeduren READ, PRINT etc. ebenfalls deklariert.

Bezüglich genauer Beschreibung und eigener Prozedurdateien siehe [7].

3. Prozeduren zur Parameterbehandlung

3.1. Parameterzahl PARZ

integer procedure PARZ;

PARZ liefert die aktuelle PARameterZahl des letzten Aufrufs der Prozedur, in der PARZ aufgerufen wird.

Anm.: PARZ entspricht in seiner Leistung der TR 440 Prozedur PARZAHL (440.FO.11), die jedoch mit den Prozeduren des BOGOL-Systems nicht kompatibel ist.

3.2. Parametertyp PART

integer procedure PART (N); value N; integer N;

PART (N) liefert den PARameter-Typ des N-ten aktuellen Parameters des letzten Aufrufs der Prozedur, in der sich der PART-Aufruf befindet.

Abgeliefert wird eine Zahl T mit $1 \leq T \leq 8$, mit folgender Verschlüsselung:

T	Parametertyp des aktuellen Parameters
1	einfach oder indizierte Variable
2	Konstante oder Ausdruck
3	Name eines Feldes
4	eine Marke, aber kein integer-label
5	integer-label
6	switch-Name
7	Name einer eigentlichen Prozedur oder einer Funktionsprozedur
8	ein string

3.3. Parameterinhalt PARV, PARA, PARL

procedure PARV (N,A,B); value N; var A, B; integer N;

PARV (N,A,B) liefert "PARameter-Value" des N-ten Parameters. Die den Parameter beschreibenden Werte werden auf den Variablen A und B abgelegt.

Nach dem Aufruf ist A und B in Abhängigkeit vom Parametertyp wie folgt belegt:

T	A	B
1	Wert der Variablen	Adresse der Variablen
2	Wert des Ausdrucks	undefiniert oder Adresse
3	Wert des 1. Feldelements	Adresse des 1. Feldelements
4	Der Aufruf von PARV bewirkt einen Sprung auf die Marke	
5	Wert der Konstanten in Gleitkomma	Adresse der Marke
6	Aufruf nicht erlaubt	
7	Falls Funktionsprozedur ohne Parameter, dann Funktionswert sonst undefiniert	Falls nicht Funktionsprozedur ohne Parameter, dann Startadresse der Prozedur, sonst undefiniert oder Adresse
8	Kopfwort des strings d.i. Länge in Festkomma	Adresse des Kopfwortes

PARA (N,A,B)

value N; integer N; var A,B;

Wie PARV, jedoch wird für den N-ten Parameter, falls er ein Array ist, nicht die Adresse und Wert des 1. Elements geliefert, sondern in B wird die Adresse des Informationsvektors abgeliefert.

PARL (N,H,B);

value N; integer N; var H,B;

Wie PARA, jedoch wird anstelle des A-Registers das H-Register übergeben.
Das H-Register hat bei folgenden Parametertypen Bedeutung:

[INTEGER] label: ⟨H⟩ = Basisadresse der Hierachie des integer label.

Prozedurname: ⟨H⟩ = Basisadresse der Hierachie aus der der Globalvektor zu übertragen ist.

PARL kann auch bei Parametertyp 6 (label) angewandt werden.
Dann ist ⟨B⟩ = Adresse

Beispiel: Es soll eine Funktion MAX (siehe 440.FO.11, 8), die den kleinsten Wert aus den Type-Parametern oder Arrays der Parameterliste liefert, in Algol geschrieben werden.

Diese hat dann (ohne Deklarationen) die Gestalt:

```
real procedure MAX;
begin
F2 := FK(2);
```

```
N := PARZ; M := 10E-150;
for I := 1 step 1 until N do
begin PARA (I,A,B);
      T := PART (I);
      if T ≠ 2 then
      begin if M<A then M := A end
      else if T = 3 then
      begin AA := B2V (B, -2);
            EA := B2V (B, -1);
WW:          A := VAL (AA);
            if M<A then M := A;
            AA := AD (AA, F2);
            if AA ≠ EA then goto WW;
      end else
      begin PRINT ('('MAX, PARAMETER FALSCH')');
            ALARM;
      end;
end;
MAX:=M;
end;
```

Die Parameter werden der Reihe nach auf ihren Typ getestet. Falls ein Array vorliegt, werden Anfangsadresse (AA) und Endadresse (EA) aus dem Versorgungsvektor bestimmt und in einer Schleife die Werte dazwischen getestet.

Als Beispiel zur Anwendung dieser Prozeduren diene ferner:

Beispiel 3.2

real procedure SUM (A_1, A_2, ..., A_N) soll die Summe von N-Variablen liefern. Der Aufruf soll mit variabler Parameterzahl erfolgen können, die Parameter sollen auf ihren Typ geprüft werden. Die Prozedur sieht dann wie folgt aus:

```
real procedure SUM;
begin
integer procedure PART(X); code;
integer procedure PARZ; code; procedure ALARM; code;
real S,A,B;
integer I, OG;
OG := PARZ; S := 0;
for I := 1 step 1 until OG do
```

```
begin if PART (I) gt 2 then goto ERROR;
S := S+APARV(I);
end;
goto ENDE;
ERROR: PRINT ('('FALSCHER TYP, PARAMETER')');
       TYPE (I);
       ALARM;
ENDE : SUM := S;
end;
```

Der Reihe nach werden die Parameter aktiviert und nur bei einer Variablen oder einem Ausdruck (Typ = 2) wird weitergemacht.

3.4. Parameterwert APARV

type procedure APARV(N); value N; integer N;

Der Aufruf von APARV(N) wirkt so, als ob der N. Parameter in der Parameterliste im Programm an dieser Stelle erscheinen würde. Der Wert von APARV(N) ist also der Wert des N. Parameters. Falls dieser vom Typ pvar ist, also eine Adresse besitzt, hat auch APARV(N) diese Adresse.

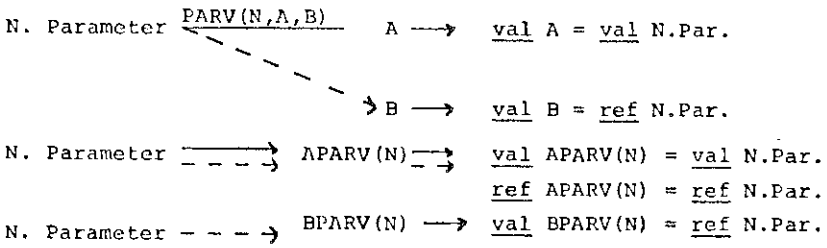
3.5. Parameteradresse BPARV

ref procedure BPARV(N); value N; integer N;

BPARV(N) liefert die Adresse des N. Parameters. Der N. Parameter muß von Typ pvar sein, also eine Adresse besitzen.

APARV und BPARV haben bezüglich der einzelnen Parametertypen die gleiche Wirkung wie PARV. D. h. ist der Parameter ein Feldname (array), so wird das erste Feldelement genommen. Ist der Parameter der Name einer Funktionsprozedur ohne Parameter, so wird der Wert des Funktionsaufrufs genommen.

3.6. Übersicht über die Prozeduren PARV, APARV, BPARV



Beispiel: Die Aufrufe

```
PARV(i, A, B)
A:= APARV(i)
A:= VAL(BPARV(i))
```

liefern nach A den gleichen Wert. Bezüglich VAL siehe 6.2. Bei A:= VAL(BPARV(i)) darf jedoch der i. Parameter keine Konstante oder ein Ausdruck sein, da er dann keine Adresse besitzt.

4. Übergabe des Ergebnis einer Funktionsprozedur, RETURN, RESULT

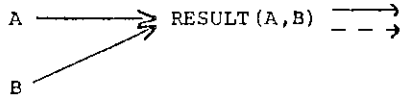
Beim ALGOL60 Laufzeitsystem wird am Prozedurende in einem Register (A-Register) der Funktionswert übergeben, der Kellerpegel auf AUF zurückgestellt und zur Rückkehradresse gesprungen. Dies läßt sich auch durch eine Code-Prozedur erreichen. Dabei kann jedoch so programmiert werden, daß auch die anderen Register erhalten bleiben. So kann auch im B-Register Information übergeben werden, z. B. Adressen oder Verweise auf Felder. Dadurch wird erreicht, daß sich ein Prozeduraufruf für seine Umgebung wie eine Variable oder ein Feld verhält. Da jedoch aus syntaktischen Gründen nicht überall ein Prozeduraufruf stehen kann, muß dieser zuvor noch in eine Rahmenprozedur eingekleidet werden. Die zur Ergebnisübergabe benötigten Prozeduren sind:

```
procedure RETURN;
```

Bewirkt Ende der Prozedur in der RETURN aufgerufen wurde.

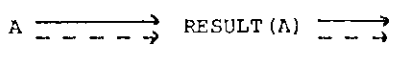
```
procedure RESULT(A[,B]); type A; ref B;
```

Bei Prozedurende wird der Wert von A als Funktionswert, der von B als Adresse übergeben. Falls der Parameter B fehlt ist die Adresse von A auch die Adresse des Prozeduraufrufs.




```
val proc = val A
ref proc = val B
```

oder falls B fehlt.




```
val proc = val A
ref proc = ref A
```

Die Adresse von A bzw. der Wert von B muß natürlich eine Adresse sein, die nach Beendigung der Prozedur und Zurücksetzen des Kellers noch gültig ist, als z. B. darf es keine lokale Adresse der Prozedur sein.

Beispiel:
```
procedure MAX;
begin N:=PARZ
  M:=10^-150;
  for i:=1 step 1 until N do
  begin PARV(i,A,B);
    if A gt M then
    begin M:=A;MB:=B; end;
  end;
  RESULT(A,B); RETURN;
end;
```

Ein Aufruf der Form MAX(X,Y,Z):=0 ist syntaktisch falsch. Dies kann jedoch umgangen werden durch:

```
procedure ZUW(A,B); A:=B;
```
und den Aufruf: `ZUW(MAX(X,Y,Z),0);`

Beim Durchreichen an ZUW verhält sich der Prozeduraufruf von MAX wie eine Variable, insbesondere besitzt er auch eine Adresse.

5. Variable Prozeduraufrufe

Für manche Anwendungen benötigt man Prozeduraufrufe, deren Parameterzahl erst zur Laufzeit festliegt. Um einen solchen Prozeduraufruf zu bewirken, muß man einen Versorgungsblock generieren und anschließend mit diesem Versorgungsblock die Prozedur aufrufen.

Zur Generierung des Versorgungsblocks dient:

```
real procedure HVZV(N); integer N;
```

HVZV liefert die Haupt- und Zusatzversorgung des N. Parameters.

Um eine lokale Größe oder einen formalen Parameter in den Prozeduraufruf zu bringen, dient:

```
real procedure FOHVZV(X);
```

FOHVZV liefert die Haupt- und Zusatzversorgung der Größe X. Der Prozeduraufruf kann dann erfolgen mit:

```
[real] procedure CALL(P ,VBO); [real] procedure P; real VBO
```

P ist die zu startende Prozedur, VBO das Kopfelement des Versorgungsblocks. Die übrigen Elemente müssen darauf folgen.

Beispiel: Die Prozedur DIFF(B,A_1, A_2 ,...,A_n) soll B- $\sum_i A_i$ berechnen mit Hilfe der Prozedur SUM (siehe oben).

```
real procedure DIFF;
begin PARV(1,B,BB); N:=PARZ
  for i:= 2 step 1 until N do
  VB [ i-1] := HVZV(i);
  VB [ 0]:= SB( HVZV( 0),1);
  DIFF := B - CALL( SUM, VB [ 0]);
end;
```

5.1. Aufbau von Versorgungsblöcken HVZV, FOHVZV

type procedure HVZV(N); integer N;

HVZV(N) liefert den Versorgungsblock des N. Parameters des aktuellen Aufrufes der Prozedur in der HVZV aufgerufen wurde. Für N = 0 wird das sogenannte Kopfwort geliefert. In diesem Kopfwort steht rechtsbündig die Parameterzahl. Soll für den generierten Prozeduraufruf die Parameterzahl geändert werden, so kann das Kopfwort durch Festkommaarithmetik(AD,SB) geändert werden.

type procedure FOHVZV(X); any X;

Sei X eine beliebige lokale Größe der Prozedur oder ein formaler Parameter. FOHVZV(X) liefert das Versorgungsblockelement, wie es für X als aktuellen Parameter benötigt wird.

5.2. Prozeduraufruf mit generiertem Versorgungsblock, CALL, FOCALL

[type] procedure CALL(P, VBO);
[type] procedure P; var VBO;

Hat man aber der Variablen VBO einen Versorgungsblock abgelegt, d. h. Kopfwort auf VBO, so kann man mit CALL (P,VBO) die Prozedur P mit diesem Versorgungsblock, d. h. mit den dadurch gegebenen aktuellen Parametern aufrufen. Voraussetzung ist jedoch, daß alle Parameter durchgereicht wurden oder Konstanten sind. D. h. abgesehen von Konstanten dürfen alle Elemente des Versorgungsblocks nur durch HVZV generiert worden sein.

[type] procedure FOCALL(P,VBO);
[type] procedure P; var VBO;

FOCALL(P,VBO) hat die gleiche Wirkung wie
CALL(P,VBO), jedoch dürfen die Versorgungsblockelemente nur durch FOHVZV generiert worden sein. Möchte man jedoch Prozeduraufrufe generieren,

die sowohl lokale Größen, als auch durchgereichte Parameter enthalten, so kann man statt HVZV(N) auch FOHVZV(APARV(N)) schreiben, und so mit FOCALL auch einen Prozeduraufruf eines ohne Parameterliste durchgereichten Parameters erreichen.

BEISPIEL:
Es soll ein Aufruf von DIFF erzeugt werden, wobei der erste Parameter eine lokale eingelesene Größe ist und die übrigen durchgereicht werden.

```
real procedure F;
begin real A;
N := PARZ;
for I := 1 step 1 until N do
VB[I+1] := FOHVZV(APARV(I));
READ(A);


VB[1] := FOHVZV(A);
VB[0] := AD(HVZV(0), 1);
F := FOCALL(DIFF, VB[0]);
end;
```

6. Werte und Adressen von lokalen Größen und formalen Parametern

6.1. Adressen REF, REFI, FOBPARV

ref procedure REF(A); var A;
REF(A) liefert die Adresse der Größe A.

ref procedure REFI(A,I); var A; integer I;
REFI(A,I) liefert die Adresse ref A + 2 · I.
Sei A ein Feld, so liefert
REFI(A[0], I) die Adresse von A[I].
REF und REFI dürfen nicht für A einen formalen Parameter enthalten. Dazu dient:

```
ref procedure FOBPARV(A); pvar A;
```

Sei A eine lokale Größe oder ein formaler Parameter, so liefert FOBARV(A) die Adresse von A. Die einzelnen Parametertypen werden dabei wie bei PARV behandelt. Z. B.: Falls A ein Feld ist, liefert FOBPARV(A) die Adresse des 1. Feldelementes von A.

6.2. Werte von Adressen VAL, VALI

var procedure VAL(A); ref A;
Sei A eine Adresse, dann ist der Wert von VAL(A) der Wert des Ganzwortes mit der Adresse A. Die Adresse von VAL(A) ist der Wert von A. Der Aufruf VAL(A) kann also überall, wo eine Variable verlangt wird, benutzt werden, zum Beispiel READ(VAL(A));

```
val VAL(A) = val val A
ref VAL(A) = val A
```

var procedure VALI(A,I); ref A; integer I;
Zur Adresse A wird 2 · I addiert und von dieser Adresse wird der Wert genommen, der den Wert von VAL(A,I) ergibt.

```
ref VAL(A,I) = val A + 2 · val I
val VAL(A,I) = val (val A + 2 · val I)
```

6.3. Inhalt von lokalen Größen (Wert und Adresse) FOPARV, FOPARA, FOPARL, FOAPARV

procedure FOPARV(A,A,B); var A, B; any X;
Von der Größe X wird der Wert auf A und die Adresse auf B abgelegt. Die Behandlung der einzelnen Parameterarten ist die gleiche wie bei PARV (3.3). Ist X ein Feldname, so enthält A den Wert und B die Adresse des ersten Feldelements. Ist X eine parameterlose Funktionsprozedur, so enthält A den Funktionswert.

procedure FOPARA(X,A,B); var A,B; any X;
Die Wirkung von FOPARA entspricht der von FOPARV. Die Behandlung der einzelnen Parametertypen von X entspricht jedoch der von PARA (3.3), so wird zum Beispiel bei einem Feldnamen X auf B die Adresse des Informationsvektors abgelegt.

procedure FOPARL(X,H,B); var H,B; any X;
Wirkt wie FOPARV, jedoch erfolgt die Behandlung der Parametertypen analog wie bei PARL, d. h. das H-Register wird auf Variable H abgelegt.

any procedure FOAPARV(X); any X
FOAPARV(X) liefert den Wert X. Dabei werden jedoch die Parameter analog zu APARV bzw. PARV behandelt. Das heißt, wenn X ein Feldname ist wird Wert und Adresse des 1. Feldelementes übergeben, und wenn X eine parameterlose Funktionsprozedur ist, wird der Funktionswert übergeben.

6.4. Zusammenfügen von Wert und Adresse zu einer Größe VR

var procedure VR(A,B); type A; ref B;

Das Ergebnis eines Aufrufs VR(A,B) ist eine Größe, die den Wert A und die Adresse B hat.

6.5. Behandlung formaler Parameter AP

Viele der BOGOL-TAS Prozeduren haben zur Erhöhung der Geschwindigkeit eine einfachere Parameterbehandlung. Insbesondere wird keine neue Indexbasis bei Aufruf der Prozedur eingerichtet. Diese führt jedoch, falls die Parameter im Aufruf formale Parameter in der Prozedur sind, in der der Aufruf steht, zu Fehlern. Um dies zu verhindern, muß der Parameter dann in einen Aufruf von AP eingekleidet werden.

type procedure AP(X); type;

AP(X) hat also die gleiche Bedeutung wie X.

Beispiel:
```
procedure F(X,Y);
begin real A;
A := FK(AP(X));
end;
```

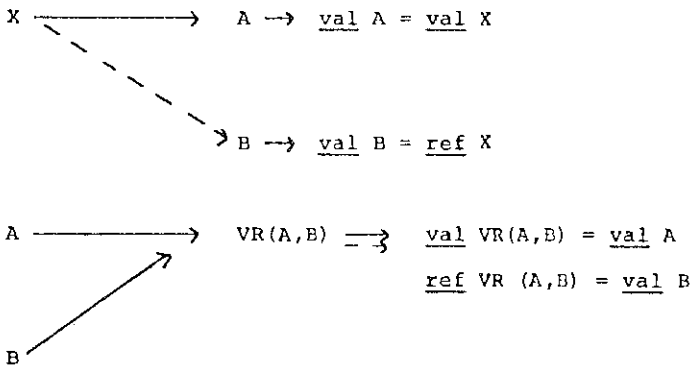
Diese Einkleidung der formalen Parameter in einen AP-Aufruf ist bei folgenden Prozeduren notwendig:

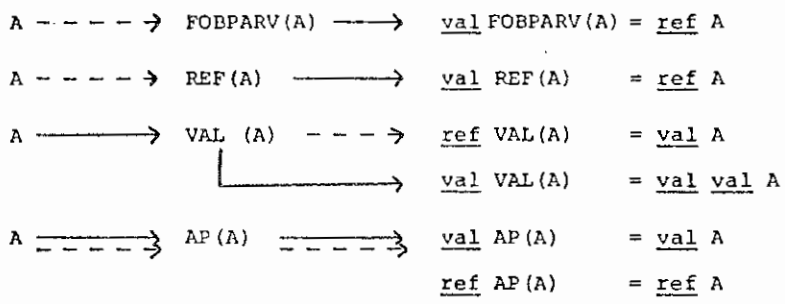
CALL, FK, FOCALL, GK, HVZV, PART, PARZ, REF, RETURN, TK, TKO, TK1, TK2, TK3, VAL, ZTO, ZT1, ZT2, ZT3

6.6. Übersicht über die Prozeduren REF, FOBPARV, VAL, FOPARV, VR, AP

FOPARV(X,A,B) ⟶ Wert (val)

- - - - - - → Adresse (ref)





6.7. Abspeichern von Werten auf Adressen

procedure AS(A,B); ref A; type B;

Der Wert von B wird auf das Ganzwort mit der Adresse A abgespeichert.

Beispiel: Alle Variablen, die zwischen A und Z deklariert worden sind, sollen auf den Wert von A gesetzt werden. Zur Demonstration werden 2 Versionen angegeben:

Version 1:

```
procedure SETZ;
begin real AA,AB,ZA,ZB; integer I,N;
PARV(1,AA,AB);
PARV(2,ZA,ZB);
N:=GK(SB(ZB,AB));
for I:= 2 step 2 until N do
AS(AD(AB,I),AA);
end;
```

Version 2:

```
procedure SETZ(A,Z);
begin real AB,ZB; integer N,I;
AB:= BFOPARV(A);
ZB:= BFOPARV(Z);
N:= GK(SB(ZB, AB));
for I:= 2 step 2 until N do
ZUW(VALI(AB, I),A);
end;
```

7. Abspeichern von Programmstücken (Parametern)

7.1. Information über einen Parameter PAR

type procedure PAR(N); integer N;

Der Funktionswert von PAR(N) enthält alle Information, die benötigt wird um den N. Parameter zu aktivieren. Dies ist die Hauptversorgung und die zur Aktivierung einzustellende Indexbasis (hier die Basisadresse der Aufrufhierachie). Dieser Funktionswert kann abgespeichert werden und später in AKT (7.3) zur Aktivierung des Parameters benutzt werden. Die Aktivierung muß natürlich erfolgen, bevor die Prozedur, in der PAR aufgerufen wurde, verlassen wird.

7.2. Information über Größen, einschließlich formaler Parameter FOPAR

type procedure FOPAR(X); any X;

Der Funktionswert von FOPAR(X) enthält alle Information, die benötigt wird um die Größe X zu aktivieren, also die Hauptversorgung und die Basisadresse der aktuellen Hierachie. Der Funktionswert kann abgespeichert und später in AKT (7.3) benutzt werden. Die Aktivierung muß jedoch erfolgen, bevor die Prozedur, in der FOPAR aufgerufen wurde, verlassen wird. X kann u. a. ein beliebiger Ausdruck sein, und so kann ein Programmstück abgespeichert werden. X darf auch ein formaler Parameter sein.

7.3. Aktivierung eines ahgespeicherten Parameters bzw. einer Größe AKT

Die mittels PAR (Parameter) bzw. FOPAR (beliebige Größe) abgespeicherten Elemente werden durch AKT aktiviert.

any procedure AKT(A); var A;

A enthält dabei den mittels PAR oder FOPAR erhaltenen Wert.

Zur Erläuterung der Funktion einige Beispiele:

A := PAR(N); B := AKT(A) wirkt wie

B := APARV(N); Behandlung der Parametertypen jedoch wie PARA.

A := FOPAR(I / J · K(; B := AKT(A) wirkt wie

B := I + J · K;

Dabei werden jedoch die aktuellen Werte für I, J, K zum Zeitpunkt des AKT-Aufrufes eingesetzt. Das B hat also hier die Bedeutung einer ref procedure, d. h. eines Verweises auf ein Programmstück.

Beispiel: Jensen-device ohne Prozedur:

```
A:= FOPAR(I·I); S:=0;
for I:=1 step 1 until 100 do S:=S+AKT(A);
```

8. Ablaufsteuerung

8.1. Erweiterter Sprung LABEL, GOTO

In Algol kann in einem Block nur innerhalb dieses Blocks oder in einen übergeordneten Block gesprungen werden. Andere Sprungziele innerhalb der Aufrufverschachtelung müssen jeweils auf Parameterposition durchgereicht werden. Mit LABEL und GOTO kann dies umgangen werden.

Ferner hat dies für getrennt übersetzte Prozeduren Bedeutung. Möchte man aus einer getrennt übersetzten Prozedur auf ein label im Hauptprogramm springen, so geht dies nur, wenn das label als Parameter übergeben wird. Für Variablen ist im ALGOL-TR 440 das common-Konzept verwirklicht, nicht jedoch für Marken. Um dies auch für Marken zu bekommen, können die Prozeduren LABEL, GOTO benutzt werden.

```
procedure LABEL (L,A);
label L; var A;
```

L ist eine Marke im Hauptprogramm und A eine Variable eines in Haupt- und Unterprogrammen erklärten common-Speichers, oder eine Variable die sowohl zum Zeitpunkt des LABEL-Aufrufes als auch des GOTO-Aufrufes gültig ist. Die Information über label L wird auf Variable A abgelegt.

```
procedure GOTO (A);
var A;
```

Falls der Aufruf LABEL (L,A) erfolgte, bewirkt GOTO(A) einen Sprung nach L;

Beispiel: Mit Hilfe von LABEL und GOTO kann eine Prozedur verlassen werden und nachher exakt in den gleichen Aufruf zurückgesprungen werden. So kann eine Prozedur verlassen werden ohne die Aufrufhierachie zu zerstören.

```
begin real A,B,C,NL;
real procedure FAK(N);
begin LABEL (LA, A); NL:=N;
goto LC;
LA: if N>1 then FAK := N*FAC(N-1) else FAK := 1;
NL:=N;
LABEL(LB,B); goto LD;
LB:
end;

C:=FAC(10);
goto ENDE;
LC: PRINT(<< N:= >>, N); GOTO(A);
LD: PRINT(<<RUECKSPRUNG, N:=>>, N); GOTO(B);
ENDE:
end
```

Merkt man sich die Label der einzelnen Aufrufe getrennt, kann jederzeit auf eine beliebige niedrigere Rekursionsstufe zurückgegangen werden.

```
begin
integer A[0:10]; integer Y;
procedure F(X); integer X;
begin LABEL(LA,A[X]); PRINT(<<X=>>, X);
LA: if X < 10 then  F(X+1) else
begin PRINT(<<NEUES X?>> ); READ Y ;
      GOTO(A[Y]);
end; end;
LABEL(ENDE, A[0]);
F(1);
ENDE:
end;
```

8.2. Assoziativer Sprung MARKE, SPRUNG

```
procedure MARKE(X,L); label L; type X;
```

Der label L wird in Zukunft unter dem Wert X ansprechbar sein. X kann ein beliebiger Ganzwortwert oder ein String mit Länge = 6 sein.

```
procedure SPRUNG(X); type X;
```

Es wird auf den x zugeordneten label gesprungen.

MARKE und SPRUNG kann u. a. auch dazu benutzt werden, um ohne Switch ein Sprungziel zu berechnen.

Beispiel:
```
MARKE(1,1); MARKE(2,2); MARKE(3.3);
LABEL(I+J);
```
Es kann errechnet werden, auf welchen integer label gesprungen werden soll.

8.3. Schneller Unterprogrammsprung SUNTPR, SRUECK

Möchte man in einem Programm ein Programmstück als Unterprogramm ohne Parameter benutzen, so kann man dies schneller als durch ein normales ALGOL-Unterprogramm durch SUNTPR und SRUECK erreichen.

```
procedure SUNTPR(L); label L;
```

Es erfolgt ein Unterprogrammsprung nach label L.

```
procedure SRUECK;
```

Es erfolgt der Rücksprung aus dem letzten mit SUNTPR angesprungenen Unterprogramm. Die Anzahl der möglichen Verschachtelungen dieser Aufrufe ist 255.

9. Systemdienste und Systemkontakte

9.1. Systemdienste

Um Dienstleistungen des Systems anzusprechen dient der Hardwarebefehl SSR (Springe ins System und Reserviere). Hier wird er durch folgende Prozedur zugänglich:

```
procedure SSR (TYP, VB1[, REG1]);
value TYP; number TYP; var VB1, REG 1;
```

Die Adresse eines SSR-Befehls besteht aus dem Paar a,b mit $0 \leq a,b \leq 255$.
In TYP ist dies in der Form a*256 + b in Festkomma- oder Gleitkommadarstellung verschlüsselt.

Im B-Register wird beim SSR-Befehl die Adresse des Versorgungsblocks übernommen. Als 1. Element des Versorgungsblocks wird VB1 angenommen dessen Adresse übergeben wird. Bei manchen SSR-Befehlen wird in denRegistern ein Ergebnis übergeben. Um dies sicherzustellen, kann die SSR-Prozedur mit einem 3. Parameter versehen werden. REG1 ist das 1. Element des Blockes, in dem die Register übergeben werden. Dies geschieht in der gleichen Form wie bei der SU-Prozedur (10.13). Benötigt man nur den Inhalt des A-Registers als Ergebnis, so kann man die Prozedur auch als

```
integer procedure SSR (TYP,VB1);
```

deklarieren. Das Funktionsergebnis ist dann der Inhalt des A-Registers nach Ausführung des SSR-Befehls.

Die Fehleradresse im SSR Versorgungsblock braucht nicht angegeben zu werden. Der Verlauf des SSR kann mit der Prozedur SUCC oder EASTEU (-25) (siehe BODAT[6]) abgefragt werden.

Möglich sind also die Aufrufe:

SSR (TYP,VB1)	keine Ergebnisübergabe
A := SSR (TYP,VB1)	Ergebnis in A
SSR (TYP,VB1,REG1)	Ergebnis in REG1 und folgenden Zellen.

Bezüglich der verschiedenen SSR-Befehle siehe [8].

Beispiel:

```
A := IE(<< 0>>, 1);
SSR(6·256+4, A);
Im Ablaufprotokoll wird der Kopftext unterdrückt.
array S[0:5], VB[0:5];
CAS(S,<<TEST>>)
C2(AD(REF(VB[1]),1), FK(1));
C2(REF(VB[1]), REF(S[1]));
SSR(6*256+8, VB[0]);
Es wird der String 'TEST' als Kopftexterweiterung eingetragen
VB[1]:= FK(128);
A:= SSR(256+8, VB[0]);
Es werden der Zustandswahlschalter Z0 gesetzt.
In A wird der Stand aller Wahlschalter nach Ausführung des SSR-Befehls
angezeigt.
```

9.2. Alarmbehandlung

9.2.1. Setzen einer Alarmadresse

```
procedure ALSETZ(L);
label L;
```

Die Alarmadresse des Operators, die sich normalerweise in der Kontrollprozedur befindet, wird auf die Adresse des Labels L umgesetzt. Beim Aufruf als integer-procedure erhält man alsFunktionsergebnis die alte Alarmadresse mit(TK=2) geliefert.

Achtung: Der Label L sollte im äußersten Block des Hauptprogramms liegen bzw. zum Zeitpunkt des Fehlers in der Aufrufverschachtelung.

9.2.2. Testen der Alarmursache ALTEST

```
integer procedure ALTEST(I);
var I;
```

Hierbei wird mittels SSR 4 8 die Alarmursache untersucht und entsprechend Variable I gesetzt. Dabei bedeutet:

i	Alarmursache
0	Interrupt
1	Arithmetischer Alarm
2	Typenkennungs-Alarm
3	Speicherschutz-Alarm
4	Überlauf Register U
5	Befehlsalarm
6	Dreierprobenalarm

Einen Interrupt erhält man, falls nach □XAN□. die Anweisung HALT,<OLN> gegeben wurde.

Als Funktionswert wird übergeben:

-1 Es ist noch kein Alarm aufgetreten

0 Es ist kein Ereignisalarm aufgetreten

Die Ereignisalarme werden durch den Funktionswert unterschieden. Siehe dazu Beschreibung BO.E2.O2. Mit dem Funktionswert 4 wird ein durch □XAN□. und HALT, Operatorlaufname angehaltener Operatorlauf gemeldet.

Nach einem Alarm ist zunächst ALTEST aufzurufen, danach muß ALSETZ gegeben werden.

Nach ALTEST (d. h. nach Fehler ist i. A. auch REKNRM (10.0) aufzurufen, um die Kellerung der vereinfachten Prozeduraufrufe zu normieren.

Beispiel: s. nächste Seite

```
        ALSETZ(4711);
        A:= B·C;
4711:   ALTEST(I);
        if I = 1 then A:= MAXW else goto FEHLER;
        ALSETZ(FEHLER);
```

Bei der Berechnung von B ·C kann ein Überlauf stattfinden. Dieser wird abgefragt und dann A auf einen Maximalwert gesetzt. Bei anderen Fehlern wird nach FEHLER verzweigt.

9.2.3. Fortsetzen nach Alarm WEITER

```
procedure WEITER;
```

Wird nach einem Alarm WEITER aufgerufen, so wird an der Stelle fortgefahren an der der Alarm aufgetreten ist. WEITER ist natürlich nur bei bestimmten Alarmarten, z. B. HALT oder OPAL sinnvoll.

9.3. Fehlerbehandlung FESETZ, FBSETZ

Die Prozedur ALSETZ fängt Hardware- oder vom Betriebssystem gemeldet Alarme ab. Nicht abgefangen wird hiermit jedoch die Beendigung des Operatorlaufes, wenn eine Standardprozedur, z. B. SQRT einen Fehler erkennt, z. B. falsches Argument. Dies wird abgefangen mit:

```
procedure FESETZ(L); label L;
```

Im Fehlerfall wird auf das Label L gesprungen. Ist der Operator nicht mit ZUSATZ=SSI montiert, muß vorher SSPLOE aufgerufen werden. Dies ist jedoch nur anwendbar, wenn der Operator in "Gebietslage" liegt, d. h. in der Standarddatenbasis erzeugt oder explizit dorthin verlagert wurde. Liegt der Operator in Dateilage vor, so kann der Schreibschutz zum Ändern der Fehleradresse in S&CC nicht aufgehoben werden.

Um dies zu umgehen dient:

```
procedure FBSETZ(L); label L;
```

Auch hier wird L als Fehlerlabel eingetragen. In "Gebietslage", d. h. nach dem Erstellen muß der Operator einmal gestartet und FBSETZ durchlaufen werden. Dabei wird der Operatorkörper entsprechend geändert und nachher braucht in Dateilage kein Schreibschutz aufgehoben werden. FBSETZ selbst schließt sich automatisch kurz. Bei FBSETZ wird implizit SSPLOE aufgerufen.

10. Hilfsprogramme

10.0. Normierung der Kellerung der Prozeduraufrufe REKNRM

procedure REKNRM

Viele BOGOL-Prozeduren arbeiten zur Steigerung der Geschwindigkeit mit einer vereinfachten Aufrufmethode insbesondere ohne Umstellen der Indexbasis. Durch einen besonderen Keller können sie dennoch rekursiv aufgerufen werden. Falls dabei ein Fehler auftritt, der abgefangen wird und so aus diesem rekursiven Aufruf herausgesprungen wird, muß dieser Keller normiert werden. Dies geschieht mit dem Aufruf REKNRM.

10.1. Konvertierung

10.1.1. Gleitkomma-Festkomma GK, FK

Die arithmetischen Operationen beziehen sich beim TR 440-Algol immer auf Gleitkommazahlen. Bei einer Anwendung auf Zahlen mit TK ≠ 0 liefern sie einen Typenkennungsalarm. Im BOGOL-System existieren daher Prozeduren zur Konvertierung.

```
fix procedure FK(X);
value X; real X;
```

Der Wert der Funktionsprozedur FK ist eine Festkommazahl mit dem Wert X. Ist der Wert von X nicht ganzzahlig, so wird gerundet. X muß $< 2^{36}$ sein, sonst ist das Ergebnis falsch.

```
real procedure GK(A);
value A; type A;
```

Der Funktionswert ist eine Gleitkommazahl mit dem Wert der Festkommagröße A. A muß $< 2^{36}$ sein, sonst ist das Ergebnis falsch. Es entsteht immer eine ganzzahlige Gleitkommazahl.

Beispiel:			
	FK(3.1)	liefert	3 mit TK1.
	GK(A)	liefert	3 wenn A = 3 mit TK1 ist.
	GK(FK(A))	liefert	den gerundeten Wert von A. ($A < 2^{36}$)

10.1.2. String→Zahlenwert NUMV, HEXV,

real procedure NUMV(S[,Fehlerlabel]); string S; label Fehlerlabel

Der Funktionswert ist der Wert der aus dem String S gebildeten Zahl. Ist die Zahldarstellung auf s unzuverlässig, so wird, falls angegeben, auf 'Fehlerlabel' gesprungen, anderenfalls eine 0 zurückgeliefert.

real procedure HEXV(S); string S;

Der String S wird als Hexadezimalzahl aufgefaßt. (Zeichen: 0 - 9, A - F). Funktionswert ist der numerische Zahlenwert.

10.1.3. Zahlenwert ⟶ String

string procedure STRV(X [,Fehlerlabel]); real X; label Fehlerlabel;

Das Ergebnis ist der String, der den Ausdruck des Wertes X darstellt. Mit der Prozedur MLSTRV kann ein Format eingestellt werden. Ist die Zahl nicht mit dem eingestellten Format wandelbar, so wird, falls angegeben, auf 'Fehlerlabel' gesprungen, sonst wird Standarddarstellung gewählt.

string procedure STRVZEI(X); integer X;

Das Ergebnis ist ein String, der aus genau einem Zeichen mit dem Zentralcodewert X ($0 \leq X \leq 255$) besteht.

procedure MLSTRV(X); value X; real X;

Es wird das Format X eingestellt für alle folgenden STRV-Aufrufe. Dabei bedeuten:

X = m Die Zahl soll in möglichst kurzer Darstellung ausgeliefert werden, darf aber maximal m Stellen lang sein. ($m \leq 24$).

-m Ist $m < 0$, so soll keine Exponentialdarstellung gewählt werden.

m.n Die Zahl soll m Stellen lang sein, davon n Stellen hinter dem Dezimalpunkt ($1 \leq n \leq 9$).

m+100 Die Zahl soll ganzzahlig gerundet werden und im Format m.0 ausgeliefert werden ($1 \leq m \leq 24$).

voreingestellt ist MLSTRV(18).

Beispiel:

NUMV(<<3.14>>)	liefert den Wert 3.14.
NUMV(<<ABC>>)	liefert den Wert 0
NUMV(<<A>>,L)	springt nach Marke L.
HEX(<<FF>>)	liefert den Wert 255
MLSTRV(6.2)	
STRV(10.1)	liefert den String ␣10.10
STRV(1000,L)	springt nach Marke L
MLSTRV(105)	
STRV(100.2)	liefert den String␣␣100
MLSTR(-5)	
STRV(100)	liefert den String 100

10.1.4. Strings ⟷ Zeichenfelder PACK, UNPACK

```
procedure PACK(F, S, ANZ);
pvar F, S; integer ANZ;
```

Ein ab F bzw. auf dem Feld F stehendes Zeichenfeld der Länge ANZ wird als String der Länge ANZ auf S abgelegt.

```
integer procedure UNPACK(S, F);
string S; pvar F;
```

Der String S wird in ein Zeichenfeld gewandelt und ab F bzw. auf das Feld F abgespeichert. Die Länge des Strings wird als Funktionswert übergeben.

Ein Zeichenfeld enthält 1 Zeichen/Wort wobei jeweils der Zentralcode-Wert mit TKO (Gleitkomma) abgelegt ist.

Beispiel:

```
for i:= 1 step 1 until 26 do
A[i ]:= 191+i;
PACK(A[1], S[0], 26);
```

Auf S wird der String 'ABC...Z' abgelegt.

```
CAS(S,<<XYZ>>);
UNPACK(S, A[1]);
```

Es ist anschließend: A[1]= 215
A[2]= 216
A[3]= 217

10.2. Festkommaarithmetik AD, SB, ML, DV

```
fix procedure AD(A,B);
value A, B; type A,B;
```

Funktionswert =A+B (Festkommaaddition)

```
fix procedure SB(A,B);
value A, B; type A,B;
```

Funktionswert = A-B (Festkommasubtraktion)

```
fix procedure ML (A,B)
value A, B; number A,B;
```

Funktionswert = A*B (Festkommamultiplikation)

```
fix procedure DV(A,B)
value A,B; number A,B;
```

Funktionswert = A/B (Festkommadivision)

Bei den Operationen AD und SB dürfen A und B beliebige Typenkennung haben. Bei TK=2 oder 3 werden die Größen als positive Zahlen aufgefaßt. Die TK des Ergebnisses ist MAX(TK(A), TK(B)). Falls die Argumente Gleitkommazahlen sind, werden sie zuvor in Festkommazahlen gewandelt.

Bei ML und DV muß es sich um Zahlen (TK=1 oder TK=0) handeln.

Beispiel: ML(AD(3,4),5) liefert den Festkommawert von (3+4)·5=35
DV(A,B) liefert eine Integer-Division auch für real-Größen A und B. Sind A und B nicht ganzzahlig werden sie zuvor gerundet.

10.3. Logische Operationen AUT, VEL, ET

Die Vergleichsoperatoren =, ne, lt, le, ge, gt können auch auf Werte mit TK≠0 angewandt werden.

Bei = und ne wird der Wert der Bits 1-48 verglichen. Bei TK=0 wird zuvor normalisiert.
Bei den Operationen lt, le, ge gt bestimmt die größte der beiden Typenkennungen den Vergleich.
Bei TK=1 werden die Zahlen mit Vorzeichen verglichen.
Bei TK=2,3 werden die Bits 1-48 als positive Zahlen aufgefaßt und verglichen.

Für die boolschen Operationen mit Bitmustern gibt es:

```
type procedure AUT(A,B)
value A,B; type A,B;
Funktionswert = not (A equiv B) (bitweise exklusives oder)

type procedure VEL(A,B);
value A,B; type A,B;
Funktionswert = A or B (bitweise oder)

type procedure ET(A,B);
value A,B; type A,B;
Funktionswert = A and B (bitweises und)
```

Die Größen A und B können beliebige Typenkennung haben.
Das Ergebnis hat wiederum die Typenkennung T_{max}. Die Operationen werden auf den 48 Bits des Ganzwortes bitweise durchgeführt.

10.4. Erzeugung beliebiger Konstanten

Um Festkommakonstanten zu erhalten kann FK benutzt werden. Um aber auch Konstanten aus Tetraden und Oktaden zu erhalten dient die Prozedur IE (Infomationseinheit).

```
form procedure IE (S,T);
value S,T; string S; integer T;
```

Der String S wird als Hexadezimalzahl aufgefaßt. Das heißt, er muß aus den Zeichen 0,1,...,9,A...,F bestehen. Diese Hexadezimalzahl wird mit der Typenkennung | T | versehen. Falls T < 0, wird sie außerdem linksbündig abgelegt. Der so erhaltene Wert ist der Funktionswert.

Beispiel: IE('('AF73')',-2) liefert das Ganzwort:

2 AF7300000000

als Funktionswert.

IE('('123 456')',3) liefert das Ganzwort:

3 000000123456

als Funktionswert.

Zwischenräume im string S sind ohne Bedeutung.

Eine besondere Form ist der Aufruf:

```
type 3 procedure IE(S);
value S; string S;
```

Falls IE mit nur 1 Parameter aufgerufen wird, ist das Prozedurergebnis der String S bzw. die 6 ersten Zeichen des Strings. Mit Hilfe von REF oder PARV kann auch auf die Adresse dieses Strings (d. h. des 1. Wortes aus Alphazeichen) zugegriffen werden. Dabei ist jedoch zu beachten, daß Konstanten schreibgeschützt abgelegt werden. Die gleiche Leistung wie der Aufruf IE(S) erbringt auch EQUIV(S); (Algol-Handbuch [2]).

Für weitere Manipulation an einzelnen Bits sei auf Kapitel 4 und die Algol-Prozeduren ASKBIT und SETBIT (440.FO.11) verwiesen.

10.5. Halbwortzugriffe B2V, C2

Um auf Halbworte zuzugreifen, können die Prozeduren B2V, C2 benutzt werden.

```
fix procedure B2V(A,I);
value A,I; ref A; integer I;
```

Der Funktionswert ist der Inhalt des Halbwortes mit der Adresse A+I.
I wird in Gleitkomma angegeben.

Der Aufruf:

B2V(A); liefert den Wert des Halbwortes mit Adresse A.

```
procedure C2(A,W);
value W; address A; type W;
```

Der Wert W wird auf das Halbwort mit der Adresse A abgespeichert.

Beispiel:

```
C2(REF(A), FK(3));
C2(AD(REF(A),1) FK(2));
```

Das linke Halbwort von A erhält den Wert 3, das rechte den Wert 2.

```
B2V(REF(A))     liefert das linke Halbwort von A
B2V(REF(A),1)   liefert das rechte Halbwort von A
for i:= 0 step 1 until N do
begin W := B2V(REF(A[0],i);
      if W = X then goto GEFUNDEN;
end;
```

Eine ab Variable A [0] stehende Halbwortliste wird durchsucht.

10.6. Zeichenverarbeitung

Für einige Zwecke, wie Zeichenverarbeitung, Typenkennung, Tabellensuchen, Shiften, gibt es spezielle Hardwarebefehle, die diese Aufgaben schnell erledigen, jedoch von den höheren Programmsprachen aus nicht angesprochen werden können. Durch spezielle Prozeduren sollen diese Leistungen auch in Algol ansprechbar sein.

Zur Zeichenverarbeitung gibt es:

10.6.1. Bringe nächstes Zeichen BNZ

```
fix procedure BNZ(IL.IR);
type IL; ref IR;
```

(Bringe Nächstes Zeichen)

Der Aufruf wirkt wie der Hardware-Befehl BNZ IL IR. Der neue Akkuinhalt ist das Funktionsergebnis.

IR ist die Adresse des Ganzwortes aus dem das Zeichen geholt wird.

IL = f * 4096+b in Festkomma oder Gleitkomma.
Dabei ist a die Anzahl der Bits, die die Zeichen besitzen sollen. Also:

4 Tetrade

6 Hexade

8 Oktade

12 Viertelwort.

b ist die Nummer des Zeichens im Wort, also 0,1,...,d

wenn d = (48/f)-1 ist.

Nach Ausführung des Befehls haben die Variablen IL und IR folgende neue Werte (immer in Festkomma):

```
b := b+1,                 wenn  b < d
b := +0
⟨IR⟩ := ⟨IR⟩ +2           wenn  b = d
```

Aus einer Zeichenfolge soll ab Adresse A die 3. bis 5. Tetrade nach B, C, D gespeichert werden.

```
IL := FK(4 4096+3);
IR := A;
B  := BNZ(IL,IR);
C  := BNZIL,IR);
D  := BNZ(IL,IR);
```

10.6.2. Speichere nächstes Zeichen CNZ

Analog arbeitet die Prozedur zum Wegspeichern von Zeichen.

```
procedure CNZ(IL,IR,W);
value W; form W; ref IR; type IL;
```

Das Zeichen W wird entsprechend IL und IR weggespeichert.

Die Typenkennung des Wortes, in das gespeichert wurde, wird auf 3 gesetzt.

Für die Anwendung von BNZ, CNZ

Beispiel: Von Adresse A Zeichen Nr. 2 sollen N Oktaden nach Adresse B Zeichen Nr. 4 transportiert werden.

```
IL := FK(8*4096+2);
IR := A;
ILC := FK(8*4096+4);
IRC := B;
for i := 1 step 1 until N do
CNZ(ILC, IRC, BNZ (IL,IR));
```

10.6.3. Transportiere Oktaden TOK

Für den Transport von Oktaden existieren noch spezielle Prozeduren:

```
procedure TOK(A,M,B,N,K,V);
ref A,B; type M,N,K,V;
```

A sei die Adresse der Oktade A_0, B die Adresse der Oktade B_0.

Ist nun $V \geq 0$ erfolgt ein Transport der Oktaden:

$$A_{M+I} \longrightarrow B_{N+I} \quad \text{für } 0 \leq i \leq K.$$

Ist dagegen $V < 0$, so erfolgt der Transport

$$I_{M+I} \longrightarrow B_{N+I} \quad \text{für } 0 \geq i \geq -K.$$

(Also Rückwärtstransport.)

Die Werte von M und N dürfen dabei auch größer als 5 sein. Es wird der Hardwarebefehl TOK benutzt.

10.6.4. Vergleich von Oktaden VOK

Eine analoge Prozedur zum Vergleich von Oktadenstrings ist:

```
boolean procedure VOK(A,M,B,N,K);
ref A; B; type M,N,K;
```

A,M,B,N,K haben die gleiche Bedeutung wie in Prozedur TOK. Nur wird hier kein Transport ausgeführt, sondern die Zeichenfolgen werden verglichen. Ist:

$$A_{M+I} = B_{N+I} \quad \text{für} \quad 0 \leq I \leq K .$$

so ist das Ergebnis true, sonst false.

Beispiel: Die Prozedur VSTR soll 2 Algol-strings auf Gleichheit in Länge und Inhalt prüfen.

```
boolean procedure VSTR;
begin
procedure PARV(X); code;
boolean procedure VOK(X); code;
procedure RESULT(X); code;
procedure RETURN(X); code;
real A,B,C,D;
PARV(1,A,B);
PARV(2,C,D);
if C ne A then RESULT (false) else
RESULT(VOK(B,6,D,6,A));
RETURN;
end;
```

10.6.5. Suche Oktaden SOK

```
fix procedure SOK(S,DS,T,DT,L,W);
ref S,T; type DS,DT,L,W;
```

In der ab Adresse S mit einem Dekrement DS gegebenen Zeichenfolge wird die ab Adresse T mit dem Dekrement DT und der Länge L+1 gegebene Zeichenfolge gesucht. Die Anzahl der Wiederholungen wird durch W gegeben. Die Nummer der Wiederholung wird als Funktionswert mit TK1 übergeben.

Das heißt, es wird getestet

$$\bigvee_{0 \le i < w-1} \left(\bigwedge_{0 < k < L} \langle S+DS+i+k \rangle = \langle T+DT+K \rangle \right)$$

Existiert ein solches i, so wird i+1 als Funktionswert in Festkomma übergeben. Falls es nicht existiert, wird FK(0) als Funktionswert übergeben.

Beispiel: Es soll getestet werden, ob der String X im String Y vorkommt.

```
L:=GK(X[0])-1;  W:=GK(Y[0])-L;
S:=REF(Y[1]);  T:=REF(X[1]);
N := GK(SOK(S,0,T,0,L,W));
```

Falls N=0 ist X nicht in Y enthalten, sonst ist X ab dem N.Zeichen in Y enthalten.

10.7. Typenkennungsoperationen

Ein besonderes Kennzeichen der TR 440 Hardware ist die Typenkennung. Für den Algol-Programmierer ist diese bisher nur über ASKBIT und SETBIT zugänglich.

Zur Behandlung der Typenkennung dienen die Prozeduren:

10.7.1. Typenkennung eines Wortes TK

```
integer procedure TK(A);
value A; type A;
```

Das Funktionsergebnis ist die TK von A in Gleitkommadarstellung.

Beispiel:

```
T:= TK(3.14);  T erhält den Wert 0
T:= TK(FK(5)); T erhält den Wert 1;
array A[0:5];
CAS(A,<<XYZ>>);
T:=TK(A[1]);   T erhält den Wert 3.
```

10.7.2. Typenkennung ZTO, ZT1, ZT2, ZT3

```
procedure ZTO(A)
var A;
```

Die TK von A wird auf 0 gesetzt.

Analog für TK 1,2,3 arbeiten die Prozeduren:

```
procedure ZT1(A);
var A;

procedure ZT2(A);
var A;

procedure ZT3(A);
var A;
```

Außerdem als Funktion:

Bei ZTO, ZT1, ZT2, ZT3 wird der geänderte Inhalt von A zusätzlich als Funktionswert übergeben.

10.7.3. Liefere Funktionswert mit Typenkennung ZT

TKO, TK1, TK2, TK3

```
type procedure ZT(A, TK); type A; integer TK;
```

Der Funktionswert ist das Bitmuster A mit der Typenkennung TK (TK=0,1,2,3). Zu beachten ist, daß bei TK=0,1 übergelaufene Zahlen entstehen können, die Arith. Alarme liefern.

```
type0 procedure TKO(A); type A;
type1 procedure TK1(A); type A;
type2 procedure TK2(A); type A;
type3 procedure TK3(A), type A;
```

Der Funktionswert ist das Bitmuster A mit der jeweiligen Typenkennung.

10.8. Shiften von Worten SH

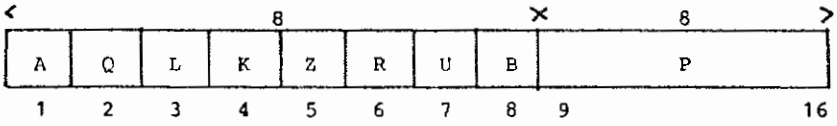
Zum Shiften dient die Prozedur:

```
procedure SH(A,S,P);
value S,P; type A; fix S,P;
```

Der Inhalt von S ist die Adresse des Shift-Befehls. Falls der Aufruf mit 3 Parametern erfolgt, wird dazu noch der Inhalt von P addiert.

A ist das zu shiftende Wort und wird ins A-Register gebracht, das Q-Register wird mit dem nächsten Ganzwort gefüllt. Nach dem Shiften werden A- und Q-Register verändert also auch das auf A folgende Wort.

Der Adressteil eines Shift-Befehls hat folgende Gestalt:



Bezüglich der Bedeutung der einzelnen Bits siehe Befehlsliste [9] Seite 17 und 23 .

Beispiel: S:= FK((128+16), 256);
SH(A, S, FK(10));

A wird um 10 bit nach rechts im Kreis geshiftet.

Ein einfaches Shiften von Zahlen (TK=1) ist natürlich auch durch Multiplikation oder Division möglich.

10.9. Tabellensuchen TLI, SULI

Eine wesentliche Beschleunigung des Tabellensuchens bedeutet die Prozedur

```
fix procedure TLI(A,B);
value A,B; ref A; type B;
```

Von der Adresse von A an wird in einer Liste solange gesucht, bis entweder der Wert B gefunden wird oder ein Ganzwort mit einer von B verschiedenen Typenkennung. Falls der Wert B gefunden wird, wird die Adressendifferenz der Zelle (mit Inhalt B) zu A übergeben, wenn nicht der Wert -1 (in Festkomma).

```
integer procedure SULI(A,B); var A; type B
```

Ähnlich, nur mit anderer Ergebnisübergabe, arbeitet die Prozedur SULL(BO.E2.03). In A wird die Variable angegeben, aber der gesucht wird. Die Ganzwortdifferenz zum befundenen bzw. -1 wird als Gleitkommazahl geliefert.

Beispiel: Bezüglich zweier globaler Ganzwortlisten A und B soll die Prozedur ASSOC(X) jeweils das $X \in A$ zugeordnete $Y \in B$ als Funktionswert liefern. Falls $X \notin A$ soll der Funktionswert 0 sein.

```
common ASSOC
array A,B [1:1000];
real procedure ASSOC (X)
value X; real X;
begin
integer procedure TLI(X); code; integer procedure VAL(X); code;
integer procedure REF(X); code; integer procedure AD(X); code;
integer procedure FK(X); code; real Z;
Z := TLI(REF(A[1]),X);
if Z < FK(0) then ASSOC := 0
else ASSOC := VAL(AD(REF(B[1],Z));
end;
```

10.10. Wortgruppentransporte WTV, WTR

Zur schnellen Umspeicherung von Wortgruppen dienen die Prozeduren WTV, WTR

```
procedure WTV(Q,Z,ANZ); var Q,Z; integer ANZ;
```

Es werden ANZ Worte ab der Adresse von Q nach der Adresse von Z und folgende übertragen:

```
⟨ref Z+2·i⟩ = ⟨ref Q+2·i⟩   i=0,1,.., ANZ-1
```

```
procedure WTR(Q,Z,ANZ); var Q,Z; integer ANZ;
```

Es werden ANZ Werte von Adresse von Q ab rückwärts nach Adresse von Z und rückwärts übertragen

```
⟨ref Z-2·i⟩ =⟨ref Q-2·i⟩   i=0,1,..., ANZ-1
```

Beispiel:

```
array A,B[1:100];
WTV(A[1], B[1], 100);
```

Das Feld A wird in das Feld B kopiert.

```
WTR(A[90], A[100], 90);
```

Das Feld A wird von A[1] bis A[90] um 10 verschoben.
Durch das Beginnen am hinteren Ende geht dies ohne Zwischenspeicherung.

10.11. Teilwortoperationen BT, CT

```
type procedure BT(BIT1, BITZ,A); type BIT1, BITZ, A;
```

Von der Größe A werden ab Bit Nr. BIT1 genau BITZ Bits genommen, nach rechts geshiftet und als Funktionswert übergeben. Der Funktionswert hat dieselbe Typenkennung wie A.

```
procedure CT(BIT1, BITZ, Z,A); integer BIT1, BITZ;
var Z; type A;
```

Die rechten BITZ Bits von A werden ab dem Bit BIT1 nach Z gespeichert. Alle anderen Bits von Z sowie die Typenkennung bleiben unverändert.

Durch BT und CT können aus einem Wort Teilbereiche ausgewählt werden.

Beispiel:

```
E := BT(41,8,A);
```

Auf E wird der Exponent von A abgespeichert.

```
CT(41,8,B,E);
```

B erhält den Exponenten von A.

10.12. Ausführen beliebiger Hardwarebefehle T

Um schließlich einen beliebigen Maschinenbefehl ansprechen zu können, dient die Prozedur

```
procedure T(OP,AD,REG1).
value OP,AD; fix OP,AD; var REG1;
```

Der Befehl mit dem Operationsteil OP und der Adresse AD wird ausgeführt. Beide sind in Internform anzugeben. Sie können jedoch z. B. mit LOC und FK beschafft werden. Vor Ausführung des Befehls werden die Registerstände mit dem Inhalt aus Zelle REG1 und folgende gefüllt, und nach Ausführung des Befehls werden sie wieder dorthin gespeichert. Die Register werden in der Reihenfolge BB, RA, RQ, RD, RH abgespeichert.

Beispiel: Als Beispiel für die Ausführung eines beliebigen TAS-Befehls soll die doppelt-genaue Gleitkomma-Multiplikation genommen werden. Dazu dient der Befehl DML. Der Interncode von DML ist F2, dezimal 242. Der 1. Faktor steht im A- und Q-Register. Der 2. Faktor in 2 Ganzwörtern mit aufeinanderfolgenden Adressen.

Das Resultat wieder in A und Q.
Die Prozedur hat 6 Parameter.
A und AA bilden den 1. Faktor,
B und BB den 2. Faktor und
C und CC das Ergebnis.

```
procedure DOPMULT(A,AA,B,BB,C,CC);
begin
real RB,RA,RQ,RD,RH; real OP1, OP2;
procedure T(X); code;
RA := A;
RQ := AA;
OP1 := B;
OP2 := BB;
T(FK(242),REF(OP1),RB);
C   := RA;
CC  := RQ;
end;
```

10.13. Erzeugen und Ausführen von TAS-Sequenzen TAS, SU

Um jedoch TAS-Sequenzen aus mehreren Befehlen evtl. öfter zu durchlaufen, dienen die Prozeduren:

procedure TAS(B1,OP_1,AD_1,...,OP_n,AO_n);
value OP_1,AD_1,...,OP_n,AD_n; var B1
fix OP_1,AD_1,...,OP_n,AD_n;

Die Befehle mit Operationsteilen OP_v und den Adressen AD_v werden auf Zelle B1 und folgende abgelegt. Insgesamt werden n+1 Halbworte benötigt. Der Ansprung erfolgt durch einen SUE-Befehl. Der letzte Befehl muß ein MU S O Befehl sein (Rücksprung).

Zur Ausführung dieser TAS-Sequenz dient dann die Prozedur

[type]procedure SU (B1, REG1);
value B1; pvar B1,REG1;

Die TAS-Sequenz ab Zelle B1 wird durchlaufen. Zuvor werden die Register aus REG1 geholt und anschließend wieder dorthin sichergestellt.

Benötigt man nicht einen bestimmten Registerinhalt, so kann man den Aufruf:

SU(B1)

wählen.

Die Register werden von REG1 an jeweils in folgender Form abgespeichert:
Die Adresse von REG 1 sei n. Dann wird:

$\langle n \rangle_{25-48}$ = B
⟨n+2⟩ = A-Register
⟨n+4⟩ = Q-Register
⟨n+6⟩ = D-Register
⟨n+8⟩ = H-Register

Der Akkumulator wird zusätzlich als Funktionswert übergeben.

10.14. Array-Manipulationen und Vorbesetzungen ARRAY, EQUIVALENCE(E2.11.) ARRADR, ARRVEC, DATA

procedure ARRAY(F,A,U1,O1,...,UN,ON);
array F; ref A; integer U1,O1,...,UN,ON;

Der Informationsvektor des Feldes F wird so geändert, daß der Inhalt von A als Anfangsadresse und die Werte U1,O1,...,UN,ON als untere bzw. obere Grenzen für F eingetragen werden. Der Speicherplatz ab Adresse A kann nun durch Indizierung von F angesprochen werden. Zusammen mit der Freispeicherverwaltung BO&FSP (12.1.12) erlaubt dies, dynamische own-Felder zu simulieren. Die Anzahl der Demensionen kann nur verkleinert werden, nicht also größer als die der ursprünglichen Deklaration werden.

```
procedure ARRADR(F, A); array F; ref A;
```

In dem Informationsvektor von F wird der Inhalt von A als Anfangsadresse eingetragen. Vereinfachte, schnellere Form von ARRAY. Bei ARRADR bleibt der übrige Inhalt den Informationsvektors unverändert. A wird nicht auf Zulässigkeit überprüft.

```
procedure EQUIVALENCE(F, B[,U1, O1,...,UN,ON]);
array F; var B; integer U1,O1,...,UN,ON;
```

Der Informationsvektor von F wird so geändert, daß B bzw. (falls B ein Feld) das erste Element von B, das erste Element von F wird. Falls angegeben, werden auch die unteren und oberen Grenzen, und damit die Dimensionslängen, entsprechend geändert.

Beispiel: Ein im Hauptprogramm erscheinender Vektor F, soll im Unterprogramm als Matrix M henutzt werden. Die Dimension der Matrix I, K wird übergeben.

```
begin
array F 1:200 ;
procedure P(F,I,K);
begin
array M[1:1, 1:1];
ARRAY(M, FOBPARV(F), 1, I, 1,K);
:
end;
integer I,K;
:
P(F, 10,20);
end
```

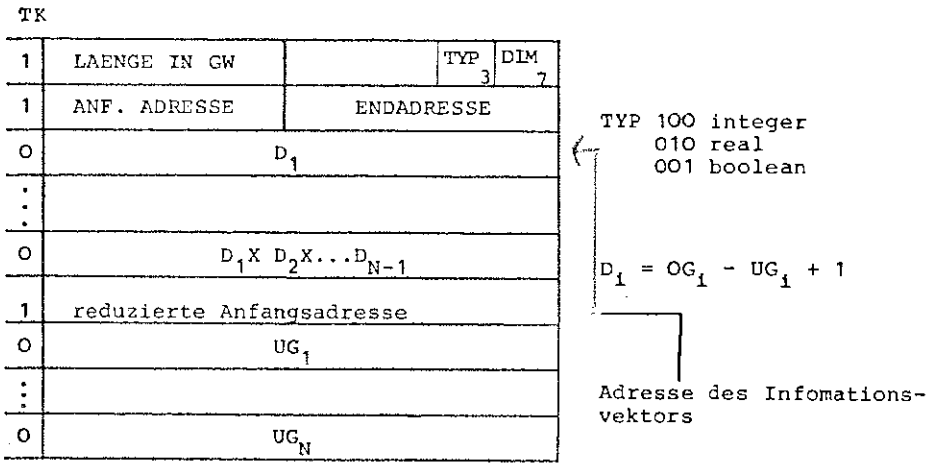
Statt ARRAY könnte hier auch EQUIVALENCE genommen werden. Der Aufruf ist dann entsprechend:

```
EQUIVALENCE(M,F,1,I,1,K);
```

```
procedure ARRVEC(F,V); array F; ref V;
```

V muß die Adresse eines Informationsvektors enthalten. Dieser Informationsvektor wird auf den Informationsvektor von F kopiert.

Zur Benutzung dieser Prozeduren soll der Aufbau eines Informationsvektors eines Feldes gegeben werden:



procedure DATA(F, V_1, V_2,...,V_n); array F; type V_i

Das Feld F wird von vorn mit den Werten V_1, V_2,..,V_n vorbesetzt.
Zur Vorbesetzung eines ganzen Feldes mit einem Wert siehe Algol-Prozedur LOESCH (440.FO.11). Ebenso die Prozeduren zur Bestimmung der Dimension (DIMA), der Länge (GRAD) und der Gesamtlänge(GRAD1).

10.15. Indirekte Referenz INDADR, MARKE, LMARKE

procedure MARKE(S,A); string oder type S; any A;

Der Größe A wird der String bzw. der Wert S zugeordnet.

any procedure INDADR(S); string oder type S;

INDADR wirkt so, als ab dort die Größe stehen würde, der der Wert bzw. String S mit MARKE zugeordnet wurde.

procedure LMARKE(S); string oder type S;

Löscht S in der Liste der Marken.
LMARKE(0) löscht alle eingetragenen Marken.

Wenn S ein String ist, darf dieser höchstens 6 Zeichen lang sein.

Beispiel: Die Funktion V(S) soll für einige Variablen so wirken, als ob statt dessen dort die Variable stehen würde, deren Namen im String S steht.

```
begin
real procedure V(S);
begin real A;
      A:=INDADR(S);
      RESULT(VAL(A));
      RETURN;
      V:=1;
end;
```

```
real A,B,C,D,X; array S[0:1];
:
MARKE(<<A>>, REF(A));
MARKE(<<B>>, REF(B));
MARKE(<<C>>, REF(C));
MARKE(<<D>>, REF(D));
:
READ(S);
X:=V(S);
ZUW(V(S), V(S)+1);
:
end
```

Beispiel: Durch die Funktion F soll jeweils von einer Zahl auf einen Namen (max. 6 Zeichen) geschlossen werden. F soll also einen String der Länge 6 als Ergebnis liefern.

```
begin
real procedure F(I);
begin SO:=FK(6)
      S1:=INDADR(I);
      RESULT(SO);
      RETURN;
      F:=1;
end;
real SO, S1; integer I;
MARKE(<<GELB>>,1);
MARKE(<<GRUEN>>,2);
:
MARKE(<<ROT>>,10);
:
:
PRINT(F(I));
end;
```

10.16. Assoziative Felder LISTE, SUCH

```
procedure LISTE(F,S1,S2,...,SN);
array F[o:m]; string S_i;
```

Am Anfang muß F[0]= 0 sein.

Anschließend werden bei jedem Aufruf jeweils die Strings S1,...,SN an die Liste angefügt. Von den Strings werden nur die ersten 6 Zeichen ausgewertet.

```
integer procedure SUCH(F,S,L);
array F; string S; label L;
```

Es wird in der Liste F der String S gesucht. Falls er gefunden wird, wird der Index als Funktionswert übergeben. Falls der String in der Liste nicht vorkommt, wird das Label L angesprungen. Von den Strings werden jeweils nur die ersten 6 Zeichen ausgewertet.

Beispiel: Sei SW ein switch, so soll auf die Komponente gesprungen werden, deren Namen eingelesen worden ist.

```
begin
switch SW :=ENDE, WEITER, ANFANG, STOP;
array F[0:4], S[0:1];

F[0]:=0;
LISTE(F,«ENDE»,«WEITER»,«ANFANG»,«STOP»);
:
READ(S);
goto SW(SUCH(F,S,FEHLER));
ANFANG:
:
WEITER
:
STOP:
:
FEHLER
:
ENDE:
end;
```

11. Testhilfen

11.1. A&TRAC

Von der ALGOL-Systemprozedur A&TRAC liegt in der Bibliothek BOGOL eine geänderte Version, die bei angemeldeter Bibliothek statt des Standardtrace anmontiert wird.

Bei diesen A&TRAC ist zunächst FTRACE(11.2) und VTRACE(11.3) möglich. Außerdem werden bei Zuweisungen in jeder TK die Werte typenkennungsabhängig ausgedruckt.

Bei normalen A&TRAC erscheint bei Zuweisungen jeweils folgender Ausdruck:

TK : 0 Gleitkommazahl

TK : 1 boolscher Wert, true oder false

TK = 2 undefiniert

TK = 3 Alphazeichen

Nach Anwendung der BOGOL-Prozeduren haben die Worte mit TK ≠ 0 jedoch auch andere Bedeutungen und ihr Inhalt ist von Interesse. Nach Anmeldung des BOGOL - A&TRAC wird bei Zuweisungen für TK ≠ 0 ein anderer Ausdruck geliefert:

TK = 1 : 1 Hexadezimaler Wert / Zahlwert

TK = 2 : 2 Hexadezimaler Wert /

TK = 3 : 3 Hexadezimaler Wert / Oktaden

Beim Ausdruck der Werte werden dabei linksbündige Nullen unterdrückt.

11.2. Trace von Funktionsaufrufen mit Parameterwerten FTRACE

FTRACE steuert einen Trace von Prozeduraufrufen mit Ausgabe der Parameter. Die Ausgabe der Parameter bewirkt eine zusätzliche Parameteraktivierung. Daher darf FTRACE nicht angewandt werden, wenn die Parameteraktivierung Seiteneffekte hat. Dies ist meist insbesondere dann der Fall, wenn auf Parameterposition ein Funktionsaufruf steht. An den Stellen, wo der Funktionstrace ausgeführt werden soll, muß mit TRACE übersetzt worden sein.

Die möglichen Aufrufe sind:

FTRACE(0); Abschalten des Funktionstrace

FTRACE(1); Einschalten des Funktionstrace für alle Prozeduraufrufe

FTRACE(2); Einschalten des Funktionstrace für die in der Tabelle (s. n. Seite) angegebenen Prozeduren.

FTRACE(3 , STR_1,...,STR_n); string STR_1,...,STR_n;
Eintragen der Strings STR_1 bis STR_n als Prozedurnamen in die Tabelle

FTRACE(4); Löschen der Tabelle

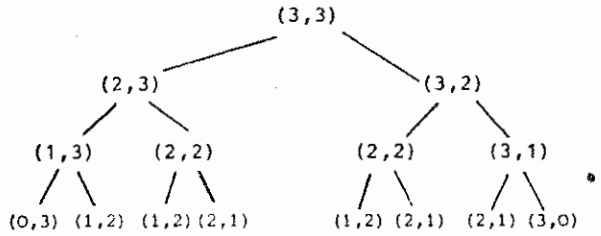
FTRACE(4 , STR_1,..., STR_n); string STR_1,..., STR_n;
Aus der Tabelle werden die den angegebenen Strings entsprechenden Prozedurnamen gelöscht.

FTRACE kann auch über Anweisungen bei Kontrollereignissen gesteuert werden.
Dazu siehe BO.E2.18!

Beispiel: Falls der Aufruf selbst nicht rekursiv ist, kann auch eine rekursive Prozedur mit FTRACE getestet werden.

```
begin
procedure F(A,B);
if A+B>N then
begin F(A-1, B); F(A, B-1) end;
real N;
N := 3;
FTRACE(1);
F(3,3);
end
```

Es wird folgender symmetrischer Aufrufbaum erzeugt:



Die Reihenfolge der Aufrufe kann durch den Funktionstrace verfolgt werden.

11.3. Trace von Zuweisungen auf vorgegebene Variablen VTRACE

VTRACE dient dazu den Trace von Zuweisung auf bestimmte Variablen zu spezifizieren. Dadurch kann insbesondere bei Ausgabe auf die Konsole der Ausdruck übersichtlich gestaltet werden.

Die möglichen Aufrufe sind:

VTRACE(0); Für alle Variablen wird der Trace abgeschaltet.

VTRACE(1); Für alle Variablen wird der Trace eingeschaltet.

VTRACE(2); Für alle Variablen der Tabelle (s. u.) wird der Trace eingeschaltet.

VTRACE(3,S_1,...,S_n); string S_1,...,S_n;
Die Variablen mit den Namen S_1 bis S_n werden in die Tabelle eingetragen.

VTRACE(4); Die gesamte Variablen-Tabelle wird gelöscht.

VTRACE(4, S_1,...,S_n); string S_1,...,S_n;
Die Variablen mit den Namen S_1 bis S_n werden in der Variablentabelle gelöscht.

VTRACE kann auch über Kontrollereignisanweisungen gesteuert werden. Dazu siehe BO.E2.18!

11.4. Dump DUMPE

```
procedure DUMPE(VAR,ANZ[, MODUS]);
var VAR; integer ANZ, MODUS;
```

Es werden ab Variable VAR ANZ Ganzworte gedumpt. Die Adressen werden dabei wie folgt angegeben:

MODUS

0	Adresse dezimal, relativ zum 1. Wort
1	Wie 0, die echte Adresse des 1. Wortes wird als Überschrift ausgegeben.
2	echte Adresse jedes Ganzwortes sedezimal
3	sedizimale Adressen relativ zum 1. Ganzwort

Fehlt der Parameter MODUS, so wird 2 gewählt.

Eine typenkennungsabhängige Interpretation wird durchgeführt:

a) für Alpha-Text
b) für Festkomma-Drittelworte
c) für Gleitkomma-Variable

11.5. Binär dump BDUMPE, ADDRESS, HEXADR (BO.E3.33)

```
procedure (S, V1, V2); string S; var V1, V2;
```

Sind V1 und V2 2 Variablen bzw. Prozeduraufrufe, die eine Variable liefern, so wird unter der Überschrift S der Bereich zwischen den Adresse von V1 und V2 gedumpt. Der Aufruf kann auch mehrere Tripel obiger Form enthalten.

Also:

```
procedure (S1, V11, V21,....,Sn, V1n, V2n);
string Si; var V1i, V2i;
```

Statt var V1 und V2 kann V1 und V2 auch sein:

(1) integer label

(2) Aufruf von ADRESS(N); integer N;
Dann wird der Bereich zwischen der Adresse N und dem anderen Parameter gedumpt

(3) Aufruf von HEXADR(S); string S;
Der String S wird als hexadezimale Adresse aufgefaßt und bildet so die Bereichsgrenze.

12. Weitere ALGOL60- Programmunterstützung

12.1. Systemhilfen

Die folgenden Prozeduren sind z. T. bereits beschrieben. Dann erfolgt hier lediglich ein Verweis mit Programmnummer. Zum Teil sind sie in [11] beschrieben.

12.1.1. Beendigung des Operatorlaufes ABBRUCH, BEENDE, OPABBR, OPSTOP (BO.E3.31)

Dies sind verschiedene Eingänge eines Unterprogramms, das den Operatorlauf beendet.

BEENDE ohne Fehler ohne Endemeldung

ABBRUCH mit Fehler ohne Endemeldung

OPSTOP(X) ohne Fehler mit Endemeldung und X als Operatorlaufname

OPABBR(X) wie OPSTOP(X) mit Fehler

Siehe [11].

12.1.2. Programmierter Alarm, ALARM

procedure ALARM;

ALARM bewirkt einen programmierten Alarm.

12.1.3. Ausdrucken von Fehlermeldungen FETEXT

procedure FETEXT(S); string S;

FETEXT(S) druckt den String S aus. FETEXT benötigt keine weiteren EA-Programme, sondern bewirkt direkt den SSR 6 12 bzw. 6 16 .

12.1.4. Ausführen von Koammandos in Dateien TUE, TUEPRE, TUEPRA (BO.ES.05)

procedure TUE(SGNR, S1, S2, L1, L2); integer SGNR, S1, SL;
label L1, L2;

Die in der Datei mit der Geräte-Nr. SGNR vor Zeile S1 bis Zeile S2 stehenden Befehle werden ausgeführt. Bei Fehler wird nach L1 oder L2 gesprungen. Falls S1 und S2 fehlen wird die gesamte Datei ausgeführt, wenn nur S2 fehlt wird der Satz S1 ausgeführt. Falls die Fehlerlabel fehlen wird hinter dem Aufruf fortgefahren. Die Codierung des Fluchtsymbolzeichens wird im ersten Wort der Common-Zone FLUCHT erwartet. Siehe auch [11].

TUEPRE und TUEPRA schalten die Protokollierung bei TUE ein bzw. aus. Es wird intern der Operator BO&TUE gestartet.

12.1.5. Ausdrucken von Dateien DRUCKE (BO.CO.14)

DRUCKE gibt eine durch die Gerätenummer spezifizierte Datei auf beliebigem Medium aus (Drucker, Stanzer, Plotter etc.)

Siehe auch [11] .

12.1.6. Datum, Genr. Vers. Nr., FKZ, MV, Zeit QDATUM, QDATGV, QFKZ, QMV, QZEIT

Es werden folgende Strings geliefert:

QDATUM	12·10·75	Datum
QDATGV	(7510·12)	Datum als G.V. Nummer
QFKZ	FKZ	Inhalt FKZ
QMV	160785	MV des Systems
ZEIT	17·35·40	Uhrzeit

12.1.7. Datei kreieren DATEI(BO.E2.16)

Es wird eineDatei, die über die symbolische Gerätenummer identifiziert wird, kreiert. Siehe [11].

12.1.8. Datei name QKAPITEL

string procedure QKAPITEL(N); integer N;
QKAPITEL(N) liefert den Namen der Datei mit der Gerätenummer N, einschließlich Datenbasisname als String. Handelt es sich um eine LF-Datei, so wird statt des Datenbasisnamens das BKZ ausgeliefert. Bei WSP-Dateien, die nicht im Standard-DMK liegen, wird stattdessen das DMK ausgeliefert.

12.1.9. Schreibschutz setzen und löschen SSPSET, SSPLOE

procedure SSPSET;
Die Schreibschutzsperre wird (wieder) gesetzt.
procedure SSPLOE;
Die Schreibschutzsperre wird aufgehoben.
Die beiden Prozeduren dürfen nur auf Operatoren in Gebietslage (d. h. in &STDDB erzeugt bzw. explizit dorthin verlagert) angewandt werden.
SSPLOE wird implizit von FBSETZ aufgerufen.

12.1.10. Startsatzauswertung STARTSATZ, SSBEREICH, SSDATEI(BO.E5.03)

Bei Anwendung dieser Prozeduren muß zuvor RB&BASTEL (BO.E5.03) benutzt worden sein, wo der Startsatz in die Datei BASTEL&DATEI geschrieben wird. Die Information in dieser Datei kann dann mit den Prozeduren STARTSATZ und SSBEREICH gezielt ausgewertet werden.

SSDATEI kann einem Teilwert einer Spezifikation als Dateinamen auswerten und dieser Datei im STARTE-Kommando eine symbolische Gerätenummer zuordnen.

12.1.11. Reservieren einer Datei RESERV(BO.E2.17);
Löschen einer Datei LOEDAT(BO.E2.13)

procedure RESERV,[RES ,[L]]; integer N, RES; label L;
Die Prozedur RESERV führt für eine mit der Ger.-Nr. N spezifizierten Datei eine Reservierung, um RES Sätze aus. L ist ein Fehlerlabel.

procedure LOEDAT(N [,L]); integer N; label L;
Die unter der symbolischen Gerätenummer N angegebene Datei wird mit SSR 253 4 gelöscht. L ist ein optionaler Fehlerlabel. Die Datei muß von der Bearbeitung abgemeldet sein (CLODA).

12.1.12. Freispeicherverwaltung BO&FSP(BO.E2.10)

Dieser Prozedurensatz erlaubt die Eröffnung und Bearbeitung eines von Algol-Freispeicher A&FSP unabhängigen Kellers. Er kann insbesondere dazu benutzt werden, um Prozedurergebnisse, die mehrere Ganzworte lang sind, z. B. Strings der Felder, zu übergeben. Alle Stringprozeduren arbeiten mit diesem Freispeicher. Siehe Programmbeschreibung BO.E2.10)

12.1.13. Dateikenndaten DATKEN

DATKEN liefert die Kenndaten einer durch die Gerätenummer identifizierten Datei aus.
procedure DATKEN(GRNR); integer GRNR;
Die Dateikenndaten der Datei mit der Gerätenummer GRNR werden im common DATKEN abgelegt. Die Kenndaten werden wie im Versorgungsblock des SSR 253 19 abgelegt.

12.1.14. ASKBV

boolean procedure ASKBV(N); integer;
Mit ASKBV(N) wird die Boolsche Variable N des Entschlüsslers abgefragt.

12.2. Stringverarbeitung: BOGOL-STRING(BO.E6.03)

Zur Stringverarbeitung steht ein Unterprogrammpaket BOGOL-STRING [4] zur Verfügung Neben den normalen Stringoperationen (Verketten, Zuweisungen, Teilstrings, etc.) erlaubt dies insbesondere die Möglichkeiten von SNOBOL4 [12] und die syntaxgesteuerte Textverarbeitung von CDL [13,14].

12.3. Erweiterte ALGOL68 ähnliche Datenstruktur=BOGOL-DATA

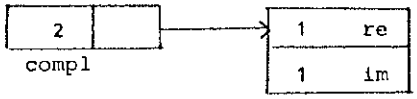
Mit den bisher aufgezeigten Hilfsmitteln wurde eine die Grundelemente von ALGOL68 [15,16] enthaltende Datenstruktur in ALGOL60 implementiert.

Mode-Deklarationen:

ALGOL68: mode compl = struct (real re, real im);

ALGOL60: real compl; real re, im; compl:= MODE(re,1,im,1);

Der Aufruf der Prozedur MODE definiert eine Struktur bestehend aus den Elementen re und im, die jeweils ein Wort im Speicher einnehmen. Erzeugt wird dabei folgende Verweiskette:

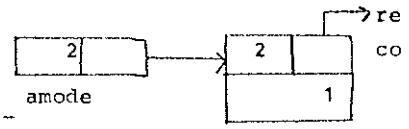


Wie in ALGOL68 können die Modes geschachtelt werden:

ALGOL68: mode amode = struct (compl co, int i);

ALGOL60: real amode,co,i; amode := MODE (co,compl,i,1);

Die dabei erzeugte Verweiskette ist:

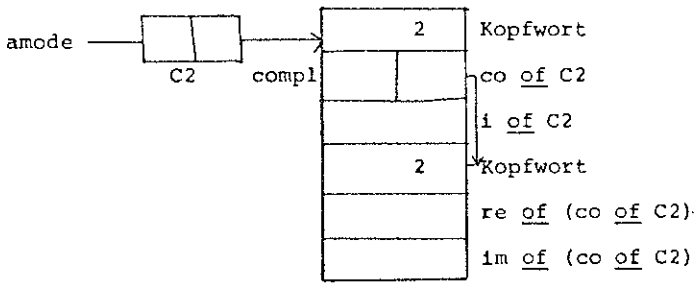


Mit der Prozedur LOC wird in einem Keller für Variablen eines definierten Modes Speicherplatz reserviert:

ALGOL68: ref compl C1 = loc compl; ref amode C2 = loc amode;

ALGOL60: real C1,C2; C1 := LOC (compl); C2 := LOC (amode);

Durch C2:= LOC(amode) wird folgende Struktur erzeugt:



Der Zugriff auf Teilstrukturen kann wie in ALGOL68 erfolgen.

Hier mit einer Prozedur OF:

real procedure OF (A,I);

Von der Strukturgröße A wird die Teilkomponente I ausgewählt. Der Prozeduraufruf wirkt wie die entsprechenden Variable. Das Zurechtfinden in den strukturierten Größen ist durch den im linken Halbwort stehenden Verweis auf den Mode leicht möglich. Auch die Tests aufMode-Gleichheit (ct) und Zuweisungen (ctab) sind durch Prozeduren realisierbar:

boolean procedure CT (A,B);

Das Ergebnis ist true, wenn mode A = mode b ist.

boolean procedure CTAB(A,B);

Wenn mode A = mode B; wird der Inhalt der für B zugewiesenen Speicherplätze auf die für A gebracht.

Um aus Konstanten bzw. Ausdrücken strukturierte Größen zu erhalten, benutzt man die Prozedur:

real procedure STRUCT(MODE, A_1,A_2,....,A_n);

Es wird aus den Werten A_1,A_2,...,A_n eine strukturierte Größe vom Mode MODE generiert.

Eine genaue Beschreibung der Prozeduren erfolgt in[5].

(1) Nauer, P. (Ed.):
Revised Report on the Algorithmic Language ALGOL60.
Numerische Mathematik 4(1963), pp. 420-453, Comm. ACM 6 (1963), pp. 1-17

(2) TR440 ALGOL60 Sprachbeschreibung
TR440 Unterlagensammlung 440.D1.04

(3) Grau, A.A., Hill, U., Langmaack,H.:
Transtation of ALGOL60. Handbuch for Automatic Compuation, Vol. I, Port b.
Berlin, Heidelberg, New-York - Springer 1967

(4) Rosendahl, M.:
BOGOL-STRING, eine flexible Zeichenkettenverarbeitung in ALGOL60.
Arbeitsberichte des Rechenzentrums der Ruhr-Universität Bochum, Nr. 7602, 1976

(5) Hauenschild, M. und Rosendahl, M.:
BOGOL-DATA, Konstruktion von ALGOL68 Datenstrukturen in ALGOL60
Erscheint demnächst als Arbeits des Rechenzentrums der Ruhr-Universität Bochum

(6) Buchmann, R.:
BODAT, ein schnelles und platzsparendes System zur Datenmanipulation und
-Speicherung in ALGOL60 und FORTRAN.
Arbeitsberichte des Rechenzentrums der Ruhr-Universität Bochum, Nr. 7405, 1974

(7) PREKOM, Prekompiler für ALGOL60 zur Einsetzung von Deklarationen.
Programmbeschreibung, Rechenzentrum der Ruhr-Universität Bochum

(8) TR440 Systemdienste
TR440 Unterlagensammlung 440.B0.01

(9) TR440 Große Befehlsliste
TR440 Unterlagensammlung 440.A1.01

(10) BO&FSP, Dynamische Freispeicherverwaltung
Rechenzentrum der Ruhr-Universität Bochum, Programmbeschreibung BO.E2.10

(11) Buchmann, R. und Wupper, H.:
Unzulänglichkeiten des TR440-Programmiersystems und ihre Umgehung
Arbeitsberichte des Rechenzentrum der Ruhr-Universität Bochum, Nr. 7403, 1974

(12) Griswold, R.E., Poage, J.F., Polansky, I.P.:
The SNOBOL4 Programming Language
Prentice-Hall, Englewood, Cliffs, New Jersey, 1970

(13) Koster, C.H.A.:
Using the CDL Compiler-Compiler in Compiler Construction,
Lecture Notes in Computer Science, 1974, pp. 366-426

(14) Koster, A.H.A.:
A compiler compiler
Matematisch Centrum Amsterdam, MR 121, 1971

(15) Lindsay, C.H., van der Meulen, S.G.:
Informal Introduktion to ALGOL68. Amsterdamm, London: North-Holland 1971

(16) A. van Wijngaarden et.al.:
Revised Report on the Algorithmic Language Algol68
Acte Informatica, 5(1975), 1-236

Bisher erschienene Arbeitsberichte des Rechenzentrums

der Ruhr-Universität Bochum

Nr. 7101: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA; eine Dialogsprache für den TR 440 (vergriffen)

Nr. 7102: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA, ein Dialogsystem und seine Implementierung in ALGOL (vergriffen)

Nr. 7103: K.-H. Mohn, M. Rosendahl, H. Zoller
AIDA, Manual für den Benutzer (vergriffen)

Nr. 7104: 4. Jahresbericht des Rechenzentrums (Juni 1970 bis Juni 1971)

Nr. 7105: H. Wupper
WR M302 - Ein einfaches Band-Betriebssystem für einen mittleren Rechner

Nr. 7201: H. Windauer
Existenzsätze zur (0,1,..,R-2,R) - Interpolation

Nr. 7202: W. Schelongowski
DIATRACE - Ein System zur interaktiven Assemblerprogrammierung

Nr. 7203: M. Jäger, M. Rosendahl, R. Staake
Einführung in die Listenverarbeitung anhand der Dialogsprache AIDA

Nr. 7204: R. Mannshardt, P. Pottinger
Einführung in die Benutzung des Teilnehmer-Rechensystems TR 440 in der RUB (vergriffen)

Nr. 7205: 5. Jahresbericht des Rechenzentrums (1.7.1971 bis 30.6.1972)

Nr. 7206: M. Rosendahl
BOGOL-TAS, ein Weg zur systemnahen Programmierung in ALGOL am TR 440

Nr. 7207: W. Stark
ILW, Programmsystem zur Berechnung des Instationären Ladungswechsels von Verbrennungskraftmaschinen (Modulbeschreibung und Eingabekonventionen)

Nr. 7208: W. Stark
ILW, Programmsystem zur Berechnung des Instationären Ladungswechsels von Verbrennungskraftmaschinen (Regelmechanismus und Berechnung der Rohrströmung)

Nr. 7209: H. Ehlich
Anregung und Kritik zum Betriebs— und Programmiersystem der TR 440

Nr. 7210: M. Rosendahl
BOGOL-STRING, eine flexible Zeichenkettenverarbeitung in ALGOL 60

Nr. 7211: H. Camici, H. Claus, H. Ehlich, D. Kipp
Arbeitsbericht über ein Programm zur Haushaltsführung

Nr. 7301: R. Mannshardt, K.-H. Mohn, H. Münch, P. Pottinger
Einführung in die Benutzung des Teilnehmer Rechensystems TR 440
2. geänderte Auflage (vergriffen)

Nr. 7302: K.-H. Mohn
Über einige Anwendungen des Computers in der Medizin

Nr. 7303: R. Buchmann
BODAT, ein schenlles und platzsparendes System zur Datenmanipulation und -speicherung in ALGOL 60 und FORTRAN

Nr. 7304: M. Hauenschild
Ansätze zur komplexen Kreisarithmetik

Nr. 7305: R. Buchmannn
RB&QUELLHALT, ein TR440-Datenbanksystem zur platzsparenden Quellhaltung auf Datenträgern mit direktrem Zugriff (LFD, WSP)

Nr. 7306: 6. Jahresbericht des Rechenzentrums (1.7.1972 bis 31.12.1973)

Nr. 7401: R. Buchmann
Der Systemoperator BO&BS3OP
Messungen und Steuerungen des Betriebssystems auf Operatorebene

Nr. 7402: R. Mannshardt
Herleitung und Prüfung spezieller Runge-Kutta-Verfahren mit einem impliziten Rechenschritt

Nr. 7403: R. Buchmann, H. Wupper
Unzulänglichkeiten des TR 440 Programmiersystems und ihre Umgehung

Nr. 7404: R. Green, K.-H. Mohn
Quellbezogene FORTRAN Optimierungen für den Compiler des TR 440

Nr. 7405: R. Buchmann
BODAT, ein schnelles und platzsparendes System zur Datenmanipulation und -speicherung in ALGOL 60 und FORTRAN (2..ergänzte Auflage)

Nr. 7501: R. Buchmann
Zur Theorie der Montage von Programmoduln

Nr. 7502: 7. Jahresbericht des Rechenzentrums (1.1. bis 31.12.1974)

Nr. 7503: H.-D. Sander
BOTRAN, eine Fortran Spracherweiterung durch Code-Prozeduren

Nr. 7504: W. Schelongowski
OIATRACE - Ein System zur interaktiven Assemblerprogrammierung

Nr. 7505: Camici, Prof. Dr. Ehlich, Schürmann
Über ein Programm zur Material- und Vervielfältigungs-Abrechnung

Nr. 7506: Camici, Prof. Dr. Ehlich, Herrmannies
Über ein Programm für die Telefonabrechnung einer Nebenstellenanlage

Nr. 7507: Camici, Prof. Dr. Ehlich, Kipp
Über ein Programm zur Haushaltsführung

Nr. 7508: Camici, Cipa, Prof. Dr. Ehlich
Bericht über ein Programm zu Verwaltung der Studentendaten

Nr. 7509: J. Riege
Zur mehrdimensionalen Spline-Interpolation bezüglich beliebiger linearer Funktionale

Nr. 7601: H. Ehlich, J. Riege u. K.-H. Schloßer
Ein Programmsystem zur Ausleihverbuchung und interaktiven Rechnerunterstützung in der allgemeinen Buchbestandsverwaltung
Teil 1: Offline-System

Nr. 7602: M. Rosendahl
BOGOL-STRING, eine flexible Zeichenkettenverarbeitung in ALGOL60
2. erweiterte und geänderte Version

Nr. 7603: M. Rosendahl
BOGOL-TA%, eine Spracherweiterung von ALGOL 60 durch Codeprozeduren zur Systemprogrammierung