RA

STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM
REKENAFDELING

# PENULTIMATE DRAFT REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 68

CHAPTERS 10, 11 AND 12

A. van Wijngaarden (Editor),
B.J. Mailloux, J.E.L. Peck
and C.H.A. Koster.

MR 99
October 1968

RA

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

10. Standard declarations

a) A "standard declaration" is one of the constituent declarations of the standard-prelude {2.1.b} {; it is either an "environment enquiry", supplying information concerning a specific property of the implementation (2.3.c), a "standard priority" or "standard operation", a "standard mathematical constant or function", a "synchronization operation" or a "transput declaration"}.

b) A representation of the standard-prelude is obtained by altering each form in 10.1, 10.2, 10.3, 10.4 and 10.5 in the following steps:

Step 1: Each sequence of symbols between ⊣ and ⊢ in a given form is altered in the following steps:

Step 1.1: If *D* occurs in the given sequence of symbols, then the given sequence is replaced by a chain of a sufficient number of sequences separated by comma-symbols; the first new sequence is a copy of the given sequence in which copy *D* is deleted; the n-th new sequence, n > 1, is a copy of the given sequence in which copy *D* is replaced by a sub-symbol followed by n-2 comma-symbols followed by a bus-symbol;

Step 1.2: If, in the given sequence of symbols, as possibly modified in Step 1.1, *L int* (*L real* or *L compl*) occurs, then that sequence is replaced by a chain of *int lengths* {10.1.a} (*real lengths* {10.1.c}) sequences separated by comma-symbols, the n-th new sequence being a copy of the given sequence in which copy each occurrence of *L*(*L*) has been replaced by (n-1) times *long*(*long*);

Step 2: Each occurrence of ⊣ and ⊢ in a given form, as possibly modified in Step 1, is deleted;

Step 3: If, in a given form, as possibly modified in Steps 1 and 2, *L int* (*L real* or *L compl*, *L bits*, *L bytes*, both *L int* and *L real* or both *L int* and *L compl*) occurs, then the form is replaced by a sequence of *int lenghts* {10.1.a} (*real lenghts* {10.1.c}, *bits widths* {10.1.f}, *bytes widths* {10.1.j},the minimum of *int lenghts* and *real lenghts*) new forms; the n-th new form is a copy of the given form in which copy each occurrence of *L*(*L*, *K*, *S*) is replaced by (n-1) times *long*(*long*, *leng*, *short*) ;

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

10. continued

Step 4: If *P* occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by four new forms obtained by replacing *P* consistently throughout the form by either - or + or × or / ;

Step 5: If *Q* occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by four new forms obtained by replacing *Q* consistently throughout the form by either *minus* or *plus* or *times* or *over* ;

Step 6: If *R* occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by six new forms obtained by replacing *R* consistently throughout the form by either < or ≤ or = or ≠ or ≥ or > ;

Step 7: Each occurrence of *F* in any form, as possibly modified or made in the Steps above, is replaced by a representation of the letter-aleph-symbol {1.1.1.n, 5.5.8} ;

Step 8: If, in some form, as possibly modified or made in the Steps above, % occurs followed by the representation of an identifier (field-selector, indication), then that occurrence of % is deleted and each occurrence of the representation of that identifier (field-selector, indication) in any form is replaced by the representation of one same identifier (field-selector, indication) which does not occur elsewhere in a form, and Step 8 is taken ;

Step 9: If a representation of a comment occurs in any form, as possibly modified or made in the Steps above, then this representation is replaced by a representation of an actual-declarer or closed-clause suggested by the comment ;

Step 10: If, in any form, as possibly modified or made in the Steps above, a representation of a routine-denotation occurs whose elaboration involves the manipulation of real numbers, then this denotation may be replaced by any other denotation whose elaboration has approximately the same effect { ; the degree of approximation is left undefined in this Report (see also 2.2.3.1.c) } ;

Step 11: The standard-prelude is that declaration-prelude whose representation is the same as the sequence of all the forms, as possibly modified or made in the Steps above.

{The declarations in this Chapter are intended to describe their effect clearly. The effect may very well be obtained by a more efficient method.}

10.1. Environment enquiries

a) **int** *int lengths* = **c** *the number of different lengths of integers* **c** ;
b) **L** **int** *L max int* = **c** *the largest L integral value* **c** ;
c) **int** *real lengths* =
  **c** *the number of different lengths of real numbers* **c** ;
d) **L** **real** *L max real* = **c** *the largest L real value* **c** ;
e) **L** **real** *L small real* = **c** *the smallest L real value such that both*
  **L1** + *L small real* > **L1** *and* **L1** - *L small real* < **L1** **c** ;
f) **int** *bits widths* =
  **c** *the number of different widths of bits* **c** ;
g) **int** *L bits width* =
  **c** *the number of elements in L bits; see* **L** **bits** {10.2.8.a} **c** ;
h) **op** **abs** = (**char** *a*) **int** :
  **c** *the integral equivalent of the character 'a'* **c** ;
i) **op** **repr** = (**int** *a*) **char** :
  **c** *that character 'x', if it exists, for which* **abs** *x* = *a* **c** ;
j) **int** *bytes widths* =
  **c** *the number of different widths of bytes* **c** ;
k) **int** *L bytes width* =
  **c** *the number of elements in L bytes; see* **L** **bytes** {10.2.9.a} **c** ;
l) **char** *null character* = **c** *some character* **c** ;

10.2. Standard priorities and operations

10.2.0. Standard priorities

a) **priority** **minus** = *1*, **plus** = *1*, **times** = *1*, **over** = *1*, **modb** = *1*, **prus** = *1*,
  ∨ = *2*, ∧ = *3*, = = *4*, ≠ = *4*, < = *5*, ≤ = *5*, ≥ = *5*, > = *5*, - = *6*,
  + = *6*, × = *7*, ÷ = *7*, ÷: = *7*, /= *7*, **elem** = *7*, ↑ = *8*, **lwb** = *8*, **upb** = *8*,
  **lws** = *8*, **ups** = *8*, ⊥ = *9* ;

10.2.1. Rows and associated operations

a) **mode** % **rows** = **c** *an* actual-declarer *specifying a mode united from*
  {4.4.3.a} *all modes beginning with* 'row of' **c** ;

10.2.1. continued

b) op lwb = (int n, rows a)int : c the lower bound in the n-th quintuple of the descriptor of the value of 'a', if that quintuple exists c ;

c) op upb = (int n, rows a)int : c the upper bound in the n-th quintuple of the descriptor of the value of 'a', if that quintuple exists c ;

d) op lws = (int n, rows a)bool : c true (false) if the lower state in the n-th quintuple of the descriptor of the value of 'a' equals 1(0), if that quintuple exists c ;

e) op ups = (int n, rows a)bool : c true (false) if the upper state in the n-th quintuple of the descriptor of the value of 'a' equals 1(0), if that quintuple exists c ;

f) op lwb = (rows a)int : 1 lwb a ;

g) op upb = (rows a)int : 1 upb a ;

h) op lws = (rows a)bool : 1 lws a ;

i) op ups = (rows a)bool : 1 ups a ;

### 10.2.2. Operations on boolean operands

a) op ∨ = (bool a, b)bool : (a | true | b) ;

b) op ∧ = (bool a, b)bool : (a | b | false) ;

c) op ¬ = (bool a)bool : (a | false | true) ;

d) op = = (bool a, b)bool : (a ∧ b) ∨ (¬a∧¬b) ;

e) op ≠ = (bool a, b)bool : ¬(a = b) ;

f) op abs = (bool a)int : (a | 1 | 0) ;

### 10.2.3. Operations on integral operands

a) op < = (L int a, b)bool : c true if the value of 'a' is smaller than that of 'b' and false otherwise c ; {2.2.3.1.c}

b) op ≤ = (L int a, b)bool : ¬(b < a) ;

c) op = = (L int a, b)bool : a ≤ b ∧ b ≤ a ;

d) op ≠ = (L int a, b)bool : ¬(a = b) ;

e) op ≥ = (L int a, b)bool : b ≤ a ;

f) op > = (L int a, b)bool : b < a ;

g) op − = (L int a, b) L int : c the value of 'a' minus that of 'b' c ; {2.2.3.1.c}

10.2.3. continued

h) op - = (L int a) L int : L0 - a ;

i) op + = (L int a, b) L int : a - - b ;

j) op + = (L int a) L int : a ;

k) op abs = (L int a) L int : (a < L0 | -a | a) ;

l) op × = (L int a, b) L int : (L int s := L0, i := abs b ;
   while i ≥ L1 do(s := s + a ; i := i - L1) ; (b < L0 | -s | s)) ;

m) op ÷ = (L int a, b) L int :
   (b ≠ L0 | L int q := L0, r := abs a ;
   while(r := r - abs b) ≥ L0 do q := q + L1 ;
   (a < L0 ∧ b ≥ L0 ∨ a ≥ L0 ∧ b < L0 | -q | q)) ;

n) op ÷: = (L int a, b) L int : a - a ÷ b × b + (a < 0 | abs b | 0) ;

o) op / = (L int a, b) L real : (L real c = a, d = b; c / d) ;

p) op ↑ = (L int a, int b) L int :
   (b ≥ 0 | L int p := L1; to b do p := p × a; p) ;

q) op leng = (L int a) long L int : c the long L integral value equivalent
   to the value of 'a' c ; {2.2.3.1.d}

r) op short = (long L int a) L int : c the L integral value, if it exists,
   equivalent to the value of 'a' c ; {2.2.3.1.d}

s) op odd = (L int a)bool : abs a ÷: L2 = L1 ;

t) op sign = (L int a)int : (a > L0 | 1 |: a < L0 | -1 | 0) ;

u) op ⊥ = (L int a, b) L compl : (a, b) ;

10.2.4. Operations on real operands

a) op < = (L real a, b)bool : c true if the value of 'a' is
   smaller than that of 'b' and false otherwise c ; {2.2.3.1.c}

b) op ≤ = (L real a, b)bool : ¬(b < a) ;

c) op = = (L real a, b)bool : a ≤ b ∧ b ≤ a ;

d) op ≠ = (L real a, b)bool : ¬(a = b) ;

e) op ≥ = (L real a, b)bool : b ≤ a ;

f) op > = (L real a, b)bool : b < a ;

g) op - = (L real a, b) L real : c the value of 'a' minus that of 'b' c
   {2.2.3.1.c} ;

h) op - = (L real a) L real : L0 - a ;

10.2.4. continued

i) op + = (L real a, b) L real : a - - b ;

j) op + = (L real a) L real : a ;

k) op abs = (L real a) L real : (a < L0 | -a | a) ;

l) op × = (L real a, b) L real : c the value of 'a' times that of 'b' c ;
{2.2.3.1.c}

m) op / = (L real a, b) L real : c the value of 'a' divided by that of
'b' c ; {2.2.3.1.c}

n) op leng = (L real a) long L real :
c the long L real value equivalent to the value of 'a' c ; {2.2.3.1.d}

o) op short = (long L real a) L real : c the L real value, if it
exists, equivalent to the value of 'a' c ; {2.2.3.1.d}

p) op round = (L real a) L int : c a L integral value, if one exists,
equivalent to a L real value differing by not more than one-half
from the value of 'a' c ;

q) op sign = (L real a)int : (a > L0 | 1 |: a < L0 | -1 | 0) ;

r) op entier = (L real a) L int : (L int j := L0 ;
(j ≤ a | e : j := j + L1 ; (j ≤ a | e | j - L1) |
f : j := j - L1 ; (j > a | f | j))) ;

s) op ⊥ = (L real a, b) L compl : (a, b) ;

10.2.5. Operations on arithmetic operands

a) op P = (L real a, L int b) L real : (L real c = b; a P c) ;

b) op P = (L int a, L real b) L real : (L real c = a; c P b) ;

c) op R = (L real a, L int b) bool : (L real c = b; a R c) ;

d) op R = (L int a, L real b) bool : (L real c = a; c R b) ;

e) op ⊥ = (L real a, L int b) L compl : (L real c = b; a ⊥ c) ;

f) op ⊥ = (L int a, L real b) L compl : (L real c = a; c ⊥ b) ;

g) op ↑ = (L real a, int b) L real : (L real p := L1 ;
to abs b do p := p × a ; (b ≥ 0 | p | L1 / p)) ;

10.2.6. Operations on character operands

a) op < = (char a, b) bool : abs a < abs b ; {10.1.h}

b) op ≤ = (char a, b) bool : ¬(b < a) ;

c) op = = (char a, b) bool : a ≤ b ∧ b ≤ a ;

d) op ≠ = (char a, b) bool : ¬(a = b) ;

10.2.6. continued

```
e) op ≥ = (char a, b) bool :  b ≤ a  ;
f) op > = (char a, b) bool :  b < a  ;
```

10.2.7. Complex structures and associated operations

```
a) struct L compl = (L real re, im) ;
b) op re = (L compl a) L real : re of a ;
c) op im = (L compl a) L real : im of a ;
d) op abs = (L compl a) L real : L sqrt(re a ↑ 2 + im a ↑ 2) ;
e) op conj = (L compl a) L compl : re a ⊥ - im a ;
f) op = = (L compl a, b) bool : re a = re b ∧ im a = im b ;
g) op ≠ = (L compl a, b) bool :¬(a = b) ;
h) op + = (L compl a) L compl : a ;
i) op - = (L compl a) L compl :- re a ⊥ - im a ;
j) op + = (L compl a, b) L compl : (re a + re b) ⊥ (im a + im b) ;
k) op - = (L compl a, b) L compl : (re a - re b) ⊥ (im a - im b) ;
l) op × = (L compl a, b) L compl :
     (re a × re b - im a × im b) ⊥ (re a × im b + im a × re b) ;
m) op / = (L compl a, b) L compl :
     (L real d = re(b × conj b) ; L compl n = a × conj b ;
     (re n / d) ⊥ (im n / d)) ;
n) op leng = (L compl a) long L compl : leng re a ⊥ leng im a ;
o) op short = (long L compl a) L compl : short re a ⊥ short im a ;
p) op P = (L compl a, L int b) L compl : (L compl c = b; a P c) ;
q) op P = (L compl a, L real b) L compl : (L compl c = b; a P c) ;
r) op P = (L int a, L compl b) L compl : (L compl c = a; c P b) ;
s) op P = (L real a, L compl b) L compl : (L compl c = a; c P b) ;
t) op ↑ = (L compl a, int b) L compl : (L compl p := L1  ;
     to abs b do p := p × a ; (b ≥ 0 | p | L1 / p)) ;
```

10.2.8. Bits structures and associated operations

```
a) struct L bits = ([1 : L bits width] bool L F) ; {See 10.1.g}
```

{The field-selector is hidden from the user in order that he may not break open the structure; in particular, he may not subscript the field.}

10.2.8. continued

```
b)  op = = (L bits a, b) bool :
      (for i to L bits width do ((L F of a)[i] ≠ (L F of b)[i] | l) ;
      true . l : false) ;
c)  op ≠ = (L bits a, b) bool : ¬(a = b) ;
d)  op ∨ = (L bits a, b) L bits :
      (L bits c ; for i to L bits width do
      (L F of c)[i] := (L F of a)[i] ∨ (L F of b)[i] ; c) ;
e)  op ∧ = (L bits a, b) L bits :
      (L bits c ; for i to L bits width do
      (L F of c)[i] := (L F of a)[i] ∧ (L F of b)[i] ; c) ;
f)  op ≤ = (L bits a, b) bool : (a ∨ b) = b ;
g)  op ≥ = (L bits a, b) bool : b ≤ a ;
h)  op ↑ = (L bits a, int b) L bits :
      if abs b ≤ L bits width then bits c := a ; to abs b do
      (b > 0 | for i from 2 to L bits width do (L F of c)[i-1] :=
      (L F of c)[i] ; (L F of c)[L bits width] := false |
      for i from L bits width by -1 to 2 do (L F of c)[i]:=
      (L F of c)[i-1] ; (L F of c)[1] := false) ; c fi ;
i)  op abs = (L bits a) L int :
      (L int c := L0 ; for i to L bits width do
      c := L2 × c + abs(L F of a)[i] ; c) ;
j)  op bin = (L int a) L bits : if a > 0 then
      L int b := a ; L bits c ; for i from L bits width by -1 to 1 do
      ((L F of c)[i] := odd b ; b := b ÷ L2) ; c fi ;
k)  op elem = (int a, L bits b) bool : (L F of b)[a] ;
l)  op L btb = ([1:] bool a) L bits :
      (int n = upb a ; (n ≤ L bits width | L bits c ;
      for i to L bits width do (L F of c)[i] := (i ≤ L bits width -n|false|
      a [i - L bits width + n]) ; c)) ;
```

10.2.9. Bytes and associated operations

```
a)  struct L bytes = ([1: L bytes width] char L F) ; {See 10.2.8.a and
      10.1.k}
b)  op < = (L bytes a, b) bool : (string := a) < (string := b) ;
c)  op ≤ = (L bytes a, b) bool : ¬(b < a) ;
d)  op = = (L bytes a, b) bool : a ≤ b ∧ b ≤ a ;
```

10.2.9. continued

```
e) op ≠ = (L bytes a, b) bool :¬(a = b) ;
f) op ≥ = (L bytes a, b) bool : b ≤ a ;
g) op > = (L bytes a, b) bool : b < a ;
h) op elem = (int a, L bytes b) char : (L F of b)[a] ;
i) op L ctb = (string a) L bytes :
      (int n = upb a ; (n ≤ L bytes width | L bytes c ;
      for i to L bytes width do (L F of c)[i] :=
      (i ≤ n | a[i] | null character) ; c)) ;
```

10.2.10. Strings and associated operations

```
a) mode string = [1 : flex] char ;
b) op < = (string a, b) bool :
      (int m = upb a, n = upb b ; int p = (m < n | m | n),
      int i := 1 ; bool c ; (p < 1 | n ≥ 1 | e :
      (c := a[i] = b[i] |:  (i := i + 1) ≤ p | e) ;
      (c | m < n | a[i] < b[i]))) ;
c) op ≤ = (string a, b) bool :¬(b < a) ;
d) op = = (string a, b) bool : a ≤ b ∧ b ≤ a ;
e) op ≠ = (string a, b) bool :¬(a = b) ;
f) op ≥ = (string a, b) bool : b ≤ a ;
g) op > = (string a, b) bool : b < a ;
h) op R = (string a, char b) bool : (string c = b ; a R c) ;
i) op R = (char a, string b) bool : (string c = a ; c R b) ;
j) op + = (string a, b) string :
      (int m = upb a, n = upb b ; [1 : m + n] char c ;
      c[1 : m] := a ; c[m + 1 : m + n] := b ; c) ;
k) op + = (string a, char b) string : (string s = b ; a + s) ;
l) op + = (char a, string b) string : (string s = a ; s + b) ;
```

{The operations defined in b, h and i imply that if abs "a" < abs "b", then "" < "a" ; "a" < "b" ; "aa" < "ab" ; "aa" < "ba" ; "ab" < "b". }

10.2.11. Operations combined with assignations

```
a) op minus = (ref L int a, L int b) ref L int : a := a - b ;
b) op minus = (ref L real a, L real b) ref L real : a := a - b ;
c) op minus = (ref L compl a, L compl b) ref L compl : a := a - b ;
```

10.2.11. continued

d) *op plus* = (*ref L int a, L int b*) *ref L int* : a := a + b ;
e) *op plus* = (*ref L real a, L real b*) *ref L real* : a := a + b ;
f) *op plus* = (*ref L compl a, L compl b*) *ref L compl* : a := a + b ;
g) *op times* = (*ref L int a, L int b*) *ref L int* : a := a × b ;
h) *op times* = (*ref L real a, L real b*) *ref L real* : a := a × b ;
i) *op times* = (*ref L compl a, L compl b*) *ref L compl* : a := a × b ;
j) *op over* = (*ref L int a, L int b*) *ref L int* : a := a ÷ b ;
k) *op modb* = (*ref L int a, L int b*) *ref L int* : a := a ÷: b ;
l) *op over* = (*ref L real a, L real b*) *ref L real* : a := a / b ;
m) *op over* = (*ref L compl a, L compl b*) *ref L compl* : a := a / b ;
n) *op Q* = (*ref L real a, L int b*) *ref L real* : a Q (*L real* := b) ;
o) *op Q* = (*ref L compl a, L int b*) *ref L compl* : a Q (*L compl* := b) ;
p) *op Q* = (*ref L compl a, L real b*) *ref L compl* : a Q (*L compl* := b) ;
q) *op plus* = (*ref string a, string b*) *ref string* : a := a + b ;
r) *op prus* = (*ref string a, string b*) *ref string* : a := b + a ;
s) *op plus* = (*ref string a, char b*) *ref string* : a := a + b ;
t) *op prus* = (*ref string a, char b*) *ref string* : a := b + a ;

## 10.3. Standard mathematical constants and functions

a) *L real L pi* = *c* a L real value close to π ; see Math. of Comp. v. 16, 1962, pp. 80-99 *c* ;

b) *proc L sqrt* = (*L real x*) *L real* : *c* if x ≥ 0, a L real value close to the square root of 'x' *c* ;

c) *proc L exp* = (*L real x*) *L real* : *c* a L real value, if one exists, close to the exponential function of 'x' *c* ;

d) *proc L ln* = (*L real x*) *L real* : *c* a L real value, if one exists, close to the natural logarithm of 'x' *c* ;

e) *proc L cos* = (*L real x*) *L real* : *c* a L real value close to the cosine of 'x' *c* ;

f) *proc L arccos* = (*L real x*) *L real* : *c* if abs x ≤ L1, a L real value close to the inverse cosine of 'x', L0 ≤ L arccos(x) ≤ L pi *c* ;

g) *proc L sin* = (*L real x*) *L real* : *c* a L real value close to the sine of 'x' *c* ;

10.3. continued

h) *proc L arcsin = (L real x) L real : c if abs x ≤ L1, a L real value close to the inverse sine of 'x', abs L arcsin(x) ≤ L pi / L2 c ;*

i) *proc L tan = (L real x) L real :*
*c a L real value, if one exists, close to the tangent of 'x' c ;*

j) *proc L arctan = (L real x) L real :*
*c a L real value close to the inverse tangent of 'x',*
*abs L arctan(x) ≤ L pi / L2 c ;*

k) *proc L random = L real expr c the next pseudo-random L real value from a uniformly distributed sequence on the interval (L0, L1) c ;*

l) *proc L set random = (L real x) : (c the next call of L random is made to deliver the value of 'x' c ; L random) ;*

## 10.4. Synchronization operations

a) *op down = (ref int dijkstra) : (do(if dijkstra ≥ 1 then dijkstra minus 1 ; l else c if the closed-statement replacing this comment is contained in a unitary-phrase which is a consituent unitary-phrase of the smallest collateral-phrase, if any, beginning with a parallel-symbol and containing this closed-statement, then the elaboration of that unitary-phrase is halted* {6.0.2.c} *; otherwise, the further elaboration is undefined c fi) ; l : skip) ;*

b) *op up = (ref int dijkstra) : (dijkstra plus 1 ; c the elaboration is resumed of all phrases whose elaboration is not terminated but is halted because the name possessed by 'dijkstra' referred to a value smaller than one c) ;*

{See 2.2.5; for insight into the use of *down* and *up*, see E.W. Dijkstra, Cooperating Sequential Processes, EWD123, Tech. Univ. Eindhoven, 1965, and also 11.13. }

10.5. Transput declarations {"So it does!" said Pooh. "It goes in!"
"So it does!" said Piglet. "And it comes out!"
"Doesn't it?" said Eeyore. "It goes in
and out like anything."
Winnie-the-Pooh, A.A. Milne.}

10.5.0. Transput modes and straightening

10.5.0.1. Transput modes

a) *mode* % *simplout* = *union*(ǂ *L* *int* ǂ, ǂ *L* *real* ǂ, ǂ *L* *compl* ǂ,
*bool*, *char*, *string*) ;

b) *mode* % *outtype* = *union*(ǂ *D* *L* *int* ǂ, ǂ *D* *L* *real* ǂ, ǂ *D* *bool* ǂ,
ǂ *D* *char* ǂ, ǂ *D* *outstruct* ǂ) ;

c) *mode* % *outstruct* = *c* *an actual-declarer specifying a mode united from* {4.4.3.a} *all modes, except that specified by* *tamrof*, *which are structured from* {2.2.4.1.d} *only modes from which the mode specified by* *outtype* *is united* *c*;

d) *mode* % *tamrof* = *struct*(*string* *F* *F*) ;{See the remarks under 5.5.8};

e) *mode* % *intype* = *union*(ǂ *ref* *D* *L* *int* ǂ, ǂ *ref* *D* *L* *real* ǂ,
ǂ *ref* *D* *bool* ǂ, ǂ *ref* *D* *char* ǂ, ǂ *ref* *D* *outstruct* ǂ) ;

10.5.0.2. Straightening

a) *op* % *straightout* = (*outtype* *x*) [] *simplout* :
*c* *the result of "straightening" 'x'* *c* ;

b) *op* % *straightin* = (*intype* *x*) [] *ref* *simplout* :
*c* *the result of straightening 'x'* *c* ;

The result of straightening a given value is a multiple value obtained in the following steps:

Step 1: If the given value is (refers to) a value from whose mode that specified by *simplout* is united, then the result is a new instance of a multiple value composed of a descriptor consisting of an offset 1 and one quintuple (1, 1, 1, 1, 1) and the given value as its only element, and Step 4 is taken;

Step 2: If the given value is (refers to) a multiple value, then, letting n stand for the number of elements of that value, and $y_i$ for the result of straightening its i-th element, Step 3 is taken; otherwise, letting n stand for the number of fields of the (of the value referred to by the) given value, and $y_i$ for the result of straightening its i-th field, Step 3 is taken ;

10.5.0.2. continued

Step 3: If the given value is not (is) a name, then, letting $m_i$ stand for the number of elements of $y_i$, the result is a new instance of a multiple value composed of a descriptor consisting of an offset 1 and one quintuple $(1, m_1 + \ldots + m_n, 1, 1, 1)$ and elements, the l-th of which, where $l = m_1 + \ldots + m_{k-1} + j$, is the (is the name referring to the) j-th element of $y_k$ for $k = 1, \ldots, n$ and $j = 1, \ldots, m_k$.

Step 4: If the given value is not (is) a name, then the mode of the result is 'row of' ('row of reference to') followed by the mode specified by *simplout*.

10.5.1. Channels and files

{aa) "Channels", "backfiles" and files model the transput devices of the physical machine used in the implementation.

bb) A channel corresponds to a device, e.g. a card reader or punch, a magnetic drum or disc, to part of a device, e.g. a piece of core memory, the keyboard of a teleprinter, or to a number of devices, e.g. a bank of tape units or even a set-up in nuclear physics the results of which are collected by the computer. A channel has certain properties (10.5.1.1.d: 10.5.1.1.n).

A "random access" channel is one for which set possible (10.5.1.1.e) is true, and a "sequential acces" channel is one for which set possible is false.

The transput devices of some physical machine may be seen in more than one way as channels with properties. The choice made in an implementation is a matter for individual taste. Some possible choices are given in table I.

cc) All information on a given channel is to be found in a number of backfiles. A backfile (10.5.1.1.b) comprises a threedimensional array of integers (bytes of information), the *book* of the backfile; the lower bounds of the *book* are all one, the upperbounds are nonnegative integers, the *maxpage*, *maxline* and *maxchar* of the backfile; furthermore, the backfile comprizes the position of the "end of file", i.e. the page number, line number and char number up to which the backfile is filled with information, the current position and the "identification-string" of the backfile.

TABLE I: Properties of some possible channels

| properties | card reader | card punch | magnetic tape unit | | | line printer |
|---|---|---|---|---|---|---|
| *reset possible* | false | false | true | true | true | false |
| *set possible* | false | false | false | false | false | false |
| *get possible* | true | false | true | true | false | false |
| *put possible* | false | true | false | true | true | true |
| *bin possible* | false | true | false | true | false | false |
| *idf possible* | false | false | true | true | true | false |
| *max page* | 1 | 1 | very large | very large | very large | very large |
| *max line* | large | very large | 16 | large | 60 | 60 |
| *max char* | 72 | 80 | 84 | large | 144 | 144 |
| *stand conv* | a 48- or 64-character code | | 64-char code | some code | line-pr code | line-pr code |
| *max nmb files* | 1 | 1 | 1 | 1 | 1 | 1 |

| properties | magnetic disc | magnetic drum | | paper tape reader | | tape punch |
|---|---|---|---|---|---|---|
| *reset possible* | true | true | true | false | false | false |
| *set possible* | true | false | true | false | false | false |
| *get possible* | true | true | true | true | true | false |
| *put possible* | true | true | true | false | false | true |
| *bin possible* | true | true | true | false | true | false |
| *idf possible* | true | true | true | false | false | false |
| *max page* | 200 | 1 | 1 | 1 | 1 | 1 |
| *max line* | 16 | 1 | 256 | very large | very large | very large |
| *max char* | 128 | 524288 | 256 | 80 | 150 | 4 |
| *stand conv* | some code | some code | some code | 5-hole code | 7-hole code | lathe code |
| *max nmb files* | 10 | 4 | 32 | 1 | 1 | 1 |

dd) After the elaboration of the declaration of *chainbfile* (10.5.1.1.c), all backfiles form the chains of backfiles referenced by *chainbfile*, each backfile chained to the next one by its field *next*.

Examples:

a) In a certain implementation, channel six is a line printer. It has no input information, *chainbfile* [*6*] is initialized to refer to a backfile the *book* of which is an integer array with upper bounds 2000, 60 and 144 (2000 pages of continuous stationery), with both the current position and the end of file at (1, 1, 1) and *next* equal to nil. All elements of the *book* are left undefined.

b) Channel four is a drum, divided into 32 segments each being one page of 256 lines of 256 bytes. It has 32 backfiles of input information (the previous contents of the drum), so *chainbfile* [*4*] is intialized to refer to the first backfile of a chain of 32 backfiles, the last one having *next* equal to nil. Each of those backfiles has an end of file at position (2, 1, 1).

c) Channel twenty is a tape unit. It can accommodate one tape at a time; one input tape is mounted and another tape laid in readiness. Here, *chainbfile* [*20*] is initialized to refer to a chain of two backfiles.

Since it is part of the standard declarations, all input is part of the program, though not of the particular-program.

ee) A file (10.5.1.2.a) is a structured value which comprizes a reference to a backfile, and the information necessary for the transput routines to work with that backfile. A backfile is associated with a file by means of *open* (10.5.1.2.b), *create* (10.5.1.2.c) or *establish* (10.5.1.2.d). A given channel can accommodate a certain number (10.5.1.1.n) of backfiles at any stage of the elaboration. The association is ended by means of *scratch* (10.5.1.2.u), *close*(10.5.1.2.s) or *lock* (10.5.1.2.t).

ff) When a file is "opened" on a channel for which *idf possible* is false, then the first backfile is taken from the chain of bfiles for that channel, and is made the *bfile* of the file, obliterating the previous backfile, if any, of the file.
When a file is opened on a channel for which *idf possible* is true, then the first backfile which has the given identification string is taken from the chain of backfiles for the channel; this backfile is made the *bfile* of the file.

gg) When a file is "established" on a channel, then a backfile is generated (8.5) with a *book* of the given size, the given identification-string and both the current position and the end of file at (1, 1, 1); when a file is "created" on a channel, then a file is established with a backfile the *book* of which has the maximum size for the channel and an empty string as its identification string.

hh) When a file is "scratched", then its associated backfile is obliterated.

ii) When a file is "closed", then first, if the given identification string is not empty, then the identification of the backfile of the file is replaced by that string; next,

it is attached to the chain referenced by *chainbfile* of the channel. Another file may now be opened with this backfile by a suitable call of *open*.

jj) When a file is "locked", then first, if the given identification string is not empty, then the identification of the backfile of the file is replaced by that string; next,

it is attached to the chain referenced by *lockedbfile* of the channel. No file can now be opened with this backfile.

kk) A file comprizes some fields of the mode 'procedure-boolean', 'procedure-with-reference-to-character-parameter-boolean' or 'procedure-with-integral-parameter-boolean', routines which are called when in transput certain error situations arise.
After opening or creating a file, the routines provided yield the value false when called, but the programmer may assign other routines to those fields. If the elaboration of such a routine is terminated, then the transput routine which called it can take no further action; otherwise, if it yields the value true, then it is assumed that the error situation has been remedied in some way, and, if possible, transput goes on, but if it yields the value false, then *undefined* is called, i.e., some sensible system action is taken (rr).

These routines are:

a) *logical file end*, which is called when during input from a file on a sequential channel the end of file of its backfile is passed. If the routine yields the value true, then transput goes on, and if it yields false, then some sensible action is taken.

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Example:

The programmer wishes to count the number of integers on his input-tape. The file *intape* was opened in a surrounding range. If he writes

```
begin int n := 0 ; logical file end of intape := goto f ;
      do(get(intape, loc int) ; n plus 1) ; f : print (n)
end,
```

then the assignment to the field of *intape* violates the scope restrictions (; the scope of the routine *(: goto f)* is smaller than the scope of *intape*), so he has to write

```
begin int n := 0 ; file auxin := intape ;
      logical file end of auxin := goto f ;
      do(get(auxin, loc int) ; n plus 1) ; f : print (n)
end.
```

b) *physical file end*, which is called when the *maxpage*, the *maxline* or the *maxchar* of the backfile of a file is exceeded.
If the routine yields the value true, then transput goes on, and if it yields false, then some sensible action is taken.

Example:

The programmer wishes automatically to give a new line at the end of a line and a new page at the end of a page on his file *f* :

```
proc new line page = bool :
((line ended (f) | new line (f)) ;
(page ended (f) | new page (f)) ; true) ;
```

c) *char error*, which is called when during input a character is read which does not agree with the frame specifying it, with as its actual-parameter a suggested character to replace it(5.5.1.mm).
If the frame is a digit- (zero-, point-, exponent-, complex-) frame and if the character read is not a digit (digit or space, point-symbol, times-ten-to-the-power-symbol, plus-i-times-symbol) then the routine is called with a digit-zero- (digit-zero-, point-, times-ten-to-the-power-, plus-i-times-) symbol. The routine provided by the programmer may give some other character instead of the suggested one. If the routine yields true, then that suggested character as possibly modified by the routine is used, and, if it yields false, then the original suggested character is used.

Example:

The programmer wishes to print a list of all such disagreements. He assigns to the field *char error* of his file *f*

*((<u>ref</u> <u>char</u> sugg) <u>bool</u> :*
*<u>char</u> k ; backspace (f) ; <u>int</u> p = page number (f),*
*l = line number (f), c = char number (f) ; get (f,k) ;*
*print ((new line, "at", p, l, c, "present. """, k,*
*""", . suggested """, sugg, """."")) ; <u>false</u>))*

d) *value error*, which is called when during formatted transput an attempt is made to transput a value under control of a picture with which it is not compatible, or when the number of frames is not sufficient. If the routine yields true, then the current value and picture are skipped, i.e., transput goes on at 5.5.1.dd Step 5; if the routine yields false, then first, on output, the value is output by *put*, and next some sensible action is taken.

e) *format end*, which is called when during formatted transput the format is exhausted while still some value remains to be transput. If the routine yields true, then transput goes on (so the routine must have provided a new current format of the file), and, if the routine yields false, then the current format is repeated, i.e., its first picture again is made to be the current picture of the file.

f) *other error*, which is called with some actual-integral-parameter, when during transput some other error situation arises. No call of this routine occurs explicitly in the standard-prelude, and neither the meaning of its actual-parameter nor that of the value yielded, is defined in this Report. This routine may, in some implementation, be called when an incorrigible hardware error occurs which makes transput involving this file impossible. (The programmer may provide a routine which then closes the file and opens it on some other channel.)

11) The *conv* of a file is used in conversion; after opening, creating or establishing a file, *stand conv* of the channel is used, but an other "conversion key" may be provided by the programmer.

On output, the given character is converted to that integer, if any, in the conversion key, whose ordinal number is the integral equivalent of that character;

on input, the given integer is converted to that character, if any, whose integral equivalent is the lowest ordinal number for which the element of the conversion key is equal to that given integer.

What action is taken when an attempt is made to convert a character with an integral equivalent exceeding the upper bound of the conversion key, or to convert an integer which is not equal to some element of the conversion key is left undefined (parity error, nonexistent code).

mm) The *term* of a file is used in reading strings of a variable number of characters, where either the end of line or any of the characters of *term* serves as a terminator (see 5.5.1.jj and 10.5.2.2.dd). This terminator string may be provided by that programmer.

nn) On a channel for which *reset possible* is true, a file may be "reset", causing its position to be (1, 1, 1). On a sequential access file the end of file remains at the position up to which the backfile contains information, but when after resetting any output is done, the end of file is first set at the current position.

oo) On a random access channel a file may be "set", causing its position to be the given position.

pp) On files opened on a sequential access channel, binary and nonbinary transput may not be alternated, i.e. after opening, creating or resetting such a file, either is possible, but, once one has taken place on the file, the other may not until the file has been reset again.

qq) On files opened on a sequential access channel for which *put possible* and *get possible* both are true, nonbinary input and output may be alternated, but it is not allowed to read past the end of file.

rr) When in transput something happens which is left undefined, for instance by an explicit call of *undefined* (10.5.1.2.y), this does not imply that the elaboration is catastrophically and immediately terminated, but only that some sensible action is taken which is not or cannot be described by this Report alone, and is generally implementation dependent. For instance, in some implementation it may be possible to set a tape unit to any position within the logical file, even if set possible is false (oo).

Example:

```
begin file f1, f2 ; [1 : 10000] int x ; int n ;
      open (f1, "my input", channel 2) ;
      f2 := f1 ; c now f1 and f2 can be used interchangeably c
      conv of f1 := flexocode ; c flexocode is a string,
      defined in the library declarations for this implementation c
      conv of f2 := telexcode ;
      c now f1 and f2 use different codes c
      reset (f1) ; c consequently f2 is reset too c
      for i while ¬ logical file ended (f1) do
      (n := i ; get (f1, x[i])) ;
      c too bad if there are more than 10000 integers in the input c
      reset (f1) ;
      for i to n do put (f2, x[i]) ;
      reset (f2) ; close (f2, "my output")
      c f1 is now closed too c
end }
```

10.5.1.1. Channels

```
a)  int nmb channels = c an integral-clause indicating the number of
      transput channels in the implementation c;
b)  struct % bfile = ([,,]int book, int lpage, lline, lchar, page,
      line, char, maxpage, maxline, maxchar, string idf, ref bfile
      next);
c)  [1 : nmb channels] ref bfile % chainbfile := c some appropriate
      initialization {see 10.5.1.dd} c;
d)  [1 : nmb channels] bool reset possible = c a row-of-boolean-clause,
      indicating which of the physical devices corresponding to the
      channels allow resetting {e.g. rewinding of a magnetic tape} c;
e)  [1 : nmb channels] bool set possible = c a row-of-boolean-clause,
      indicating which devices can be accessed at random c;
f)  [1 : nmb channels] bool get possible = c a row-of-boolean clause,
      indicating which devices can be used for input c;
g)  [1 : nmb channels] bool put possible = c a row-of-boolean-clause,
      indicating which devices can be used for output c;
h)  [1 : nmb channels] bool bin possible = c a row-of-boolean-clause,
      indicating which devices can be used for binary transput c;
i)  [1 : nmb channels] bool idf possible = c a row-of-boolean-clause,
      indicating on which devices backfiles have an identification c;
j)  [1 : nmb channels] int max page = c a row-of-integral-clause,
      giving the maximum number of pages per file for the channels c;
k)  [1 : nmb channels] int max line = c a row-of-integral-clause,
      giving the maximum number of lines per page c;
l)  [1 : nmb channels] int max char = c a row-of-integral-clause,
      giving the maximum number of characters per line c;
m)  [1 : nmb channels] ref [] int stand conv = c a row-of-reference-
      to-row-of-integral-clause giving the standard conversion keys for
      the channels c;
n)  [1 : nmb channels] int max nmb files = c a row-of-integral-clause,
      giving the maximum numbers of files the channels can accomodate c;
o)  [1 : nmb channels] int % nmb opened files;
      for i to nmb channels do nmb opened files [i] := 0;
p)  [1 : nmb channels] ref bfile % lockedbfile;
      for i to nmb channels do lockedbfile [i] := nil;
```

10.5.1.1. continued

```
q)  proc file available = (int channel) bool :
      nmb opened files [channel] < max nmb files [channel];
```

10.5.1.2. Files

```
a)  struct file = (ref bfile % bfile, ref int % chan, % forp,
      ref bool % state def, % state get, % state bin, % opened,
      ref string % format, string term,  [1 :] int conv,
      proc bool logical file end, physical file end, format end,
      value error, proc (ref char) bool char error, proc (int) bool
      other error) ;
b)  proc open = (ref file file, string idf, int ch) :
      if file available (ch)
      then ref bfile bf := chainbfile [ch], obf := nil ;
        while bf :≠: nil do
        (idf of bf = idf ∨ ¬ idf possible [ch] | l | obf := bf ;
        bf := next of bf) ; undefined.
      l : file := (bfile := bf, int := ch, int := 0, bool := false,
        bool, bool, bool := true, nil,,stand conv [ch], false,
        false, false, false, ((ref char a) bool : false), skip) ;
        (obf  :=:  nil | chainbfile [ch] | val next of obf) := next of bf ;
        numb opened files [ch] plus 1 ;
      else undefined fi;
c)  proc create = (ref file file, int ch) :
      establish (file, max page [ch], max line [ch], max char [ch], ch) ;
d)  proc establish = (ref file file, string idf, int mp, ml, mc, ch) :
      if file available (ch) ∧ mp ≤ max page [ch] ∧
        ml ≤ max line [ch]   ∧ mc ≤ max char [ch]
      then bfile bf = ([1 : mp, 1: ml, 1 : mc] int, 1, 1, 1,1, 1, 1,
        mp, ml, mc, idf, nil) ;
        file := (bfile := bf, int := ch, int := 0, bool := false,
        bool, bool, bool := true, nil,, stand conv[ch], false,false,
        false, false, ((ref char a) bool : false), skip) ;
        nmb opened files [ch] plus 1 ;
      else undefined fi;
```

```
e)  proc set = (file file, int p, l, c) :
      if set possible [chan of file] ∧ opened of file
      then page of bfile of file := p ; line of bfile of file := l ;
        char of bfile of file := c ; check plc (file) ;
      else undefined fi;
f)  proc reset = (file file) :
      if reset possible [chan of file] ∧ opened of file
      then page of bfile of file := 1 ; line of bfile of file := 1 ;
        char of bfile of file := 1 ; state def of file := false ;
      else undefined fi;
g)  proc % check plc = (file file) : if opened of file
      then (¬(logical file ended file) | logical file end of file |:
        line ended (file) ∨ page ended (file) ∨ file ended (file)
        | physical file end of file | true) | undefined)
      else undefined fi;
h)  proc line ended = (file file) bool : (opened of file |
      int c = char of bfile of file ; c ≤ 0 ∨ c > max char of bfile of
      file);
i)  proc page ended = (file file) bool : (opened of file |
      int l = line of bfile of file ; l ≤ 0 ∨ l > max line of bfile of
      file);
j)  proc file ended = (file file) bool : (opened of file |
      int p = page of bfile of file ; p ≤ 0 ∨ p > max page of bfile of
      file);
k)  proc logical file ended = (file file) bool : (opened of file |:
      ¬ set possible [chan of file] ∧ state def of file ∧ state get of file |
      bfile b = bfile of file ;
      int p = page of b, lp = lpage of b, l = line of b, ll = lline of b,
          c = char of b, lc = lchar of b ;
      (p < lp | false |: p > lp | true |: l < ll | false |: l > ll |
      true | c ≥ lc) | false);
l)  proc % get string = (file file, ref string s) :
      if get possible [chan of file] ∧ opened of file
      then ref int p = page of bfile of file, l = line of bfile of file,
        c = char of bfile of file ;
```

```
        if ¬ set possible [chan of file] thef state def of file
        then (state bin of file | undefined) fi ;
        state def of file := state get of file := true ;
        state bin of file := false ;
        for i to upb s do
          (check plc (file) ; for j to upb conv of file do
          ((conv of file)[j] = book of bfile of file [p, l, c] | s[i] :=
          repr j ; e) ; undefined.
        e : c plus 1)
      else undefined fi;
m)  proc % put string = (file file, string s) :
      if put possible [chan of file] ∧ opened of file
      then ref int p = page of bfile of file, l = line of bfile of file,
        c = char of bfile of file ;
        if ¬ set possible [ch] thef state def of file
        then (state bin of file | undefined) fi ;
        state get of file := state bin of file := false ;
        state def of file := true ;
        for i to upb s do
          (check plc (file) ; book of bfile of file [p, l, c] :=
          conv of file [abs s[i]]  ; c plus 1 ;
          (p = lpage of bfile of file ∧ l = lline of bfile of file
          | (c > lchar of bfile of file | lchar of bfile of file := c)
          | lpage of bfile of file := p ; lline of bfile of file := l ;
            lchar of bfile of file := c))
      else undefined fi;
n)  proc char in string = (char c, ref int i, string s) bool :
      (for k to upb s do (c = s[k] | i := k ; l) ; false. l : true) ;
o)  proc space = (file file) :
      (char of bfile of file plus 1 ; check plc (file)) ;
p)  proc backspace = (file file) :
      (char of bfile of file minus 1 ; check plc (file));
q)  proc new line = (file file) :
      (line of bfile of file plus 1 ; char of bfile of file := 1 ;
      check plc (file));
```

r) proc new page = (file file) :
(page of bfile of file plus 1 ; line of bfile of file := char of bfile of file := 1 ; check plc (file));

s) proc close = (file file, string idf) :
(opened of file | int ch = chan of file ;
(idf ≠ "" | idf of bfile of file := idf) ;
next of bfile of file := chainbfile[ch] ;
chainbfile[ch] := bfile := bfile of file ;
opened of file := false ; nmb opened files [ch]minus 1) ;

t) proc lock = (file file, string idf) :
(opened of file | int ch = chan of file ;
(idf ≠ "" | idf of bfile of file := idf) ;
next of bfile of file := lockedfile[ch] ;
lockedbfile[ch] := bfile := bfile of files ;
opened of file := false ; nmb opened files[ch] minus 1) ;

u) proc scratch = (file file) :
(opened of file | opened of file := false ;
nmb opened files[chan of file] minus 1) ;

v) proc char number = (file f) int : (opened of f | char of bfile of f) ;

w) proc line number = (file f) int : (opened of f | line of bfile of f) ;

x) proc page number = (file f) int : (opened of f | page of bfile of f) ;

y) proc % undefined = ((false | true) | skip) ; {10.5.1.rr}.

10.5.1.3. Standard channels and files

a) int stand in channel = c an integral-clause such that get possible [stand in channel] is true and idf possible [stand in channel] is false c ;

b) int stand out channel = c an integral-clause such that put possible [stand out channel] is true and idf possible [stand out channel] is false c ;

c) int stand back channel = c an integral-clause such that reset possible [stand back channel], set possible [stand back channel], get possible [stand back channel], put possible [stand back channel] and bin possible [stand back channel] are true and idf possible [stand back channel] is false c ;

10.5.1.3. continued

d) *file % f ; open (f, stand in channel) ;*
*file stand in = f ;*

e) *open (f, stand out channel) ;*
*file stand out = f ;*

f) *open (f, stand back channel) ;*
*file stand back = f ;*

{Certain "standard files" (d, e, f) need not (and cannot) be opened by the programmer, but are opened for him in the standard declarations; *print* (10.5.2.1.a) can be used for output on *stand out, read* (10.5.2.2.a) for input from *stand in*, and *write bin* (10.5.4.1.a) and *read bin* (10.5.4.2.a) for transput involving *stand back.* }

10.5.2. Formatless transput

10.5.2.1. Formatless output

{For formatless output, *print* and *put* can be used. The elements of the given value of the mode specified by [] *union (outtype, proc (file))* are treated one after the other; if an element is of the mode specified by *proc (file)* (i.e. a "layout procedure"), then it is called with the file as its parameter; otherwise, it is straightened (10.5.0.2), and the resulting values are output on the given file one after the other, as follows:

aa) If the mode of the value is specified by *L int*, then first, if there is not enough room on the line for *L int width + 2* characters, then this room is made by giving a new line and, if the page is full, giving a new page; next, when not at the beginning of a line, a space is given and the value is edited as if under control of the picture *n(L int width - 1)z+d.*

bb) If the mode of the value is specified by *L real*, then, first, if there is not enough room on the line for *L real width + L expwidth + 5* characters, then this room is made; next, when not at the beginning of a line, a space is given, and the value is edited as if under control of the picture *+d.n(L real width - 1)den(L expwidth - 1)z+d.*

cc) If the mode of the value is specified by *L compl*, then first, if there is not enough room on the line for *2 × (L real width + L exp width + 5) + 2* characters, then this room is made; next, when not at the beginning of a new line a space is given, and the value is edited as if under control of the picture *+d.n(L real width - 1) den(L expwidth - 1)z+d "." i + d.n(L real width -1)den(L expwidth - 1)z+d.*

dd) If the mode of the value is specified by [] *char* then its elements are written one after the other.

ee) If the mode of the value is specified by *char* then, first if the line is full room is made; then the character is written.

ff) If the mode of the value is specified by *bool* then, if the value is true (false) a flip (flop) is output as in ee. }

```
a)  proc print = ([] union (outtype, proc (file)) x) :
        put (stand out, x) ;
b)  proc put = (file file, [1 :] union (outtype, proc (file)) x) :
        begin outtype ot ; proc (file) pf ;
              for i to upb x do
              (ot ::= x[i]; pf ::= x[i] | pf (file)|
              [1 :] simplout y = straightout ot ;
              for j to upb y do
              (string s ; bool b ; char c ;
              (⊀ (L int i ; (i ::= y[j] |
                 s := L int string (i, L int width + 1, 10) ;
                 sign supp zero (s, 1, L int width + 1))) ⊁) ;
              (⊀ (L real x ; (x ::= y[j] | s := L real conv (x)))⊁) ;
              (⊀ (L compl z ; (z ::= y[j] | s := L real conv (re z)
                 + ". ⊥"+ L real conv (im z))) ⊁) ;
              (b ::= y[j] | s := (b | "1" | "0")) ;
              (c ::= y[j] | nextplc (file) ; put string (file, c) ; end) ;
              (s ::= y[j] | putstring (file, s) ; end) ;
              ref int c = char of bfile of file ; int c 1 = c, n = upb s ;
              c plus (c 1 = 1 | n | n + 1) ;
              (line ended (file) | next plc (file) | c := c 1) ;
              put string (file, (c = 1 | s | "." + s)) ;
        end : skip))
        end ;
```

```
c)  proc L int string = (L int x, int w, r) string : (r > 1 ∧ r < 17 |
      string c := ; L int n := abs x ; L int lr = Kr ;
      for i to w - 1 do (c prus dig char (S (n ÷: lr)) ; n over lr) ;
      (n = L0 | (x ≥ L0 | "+" | "-") + c)) ;
d)  proc L real string = (L real x, int w, d, e) string :
      (d ≥ 0 ∧ e > 0 ∧ d + e + 4 ≤ w |
      L real g = L10 ↑ (w - d - e - 4) ; L real h = g × L.1 ;
      L real y := abs x ; int p := 0 ;
      while y ≥ g do (y times L.1 ; p plus 1) ;
      (y > L0 | while y < h do (y times L10 ; p minus 1)) ;
      (y + L.5 × L.1 ↑ d ≥ g | y := h ; p plus 1) ;
      L dec string (x ≥ 0 | y | -y), w - e - 2, d) +
      "10" + int string (p, e + 1, 10)) ;
e)  proc L dec string = (L real x, int w, d) string :
      (abs x < L10 ↑ (w - d - 2) ∧ d ≥ 0 ∧ d + 2 ≤ w | string s:=  ;
      L real y := (abs x + L.5 × L.1 ↑ d) × L.1 ↑ (w - d - 2) ;
      for i to w - 2 do s plus dig char (int c = entier S (ytimes L10) ;
      y minus K c ; c)) ;
      (x ≥ 0 | "+" | "-") + s[1 : w - d - 2] + "." + s[w - d - 1 : ]) ;
f)  proc % dig char = (int x) char : ("0123456789abcdef" [x + 1] ) ;
```

{In connection with 10.5.2.1.c,d,e, see Table II.}

```
g)  proc % sign supp zero = (ref string c, int l, u) :
      for i from l + 1 to u while c[i] = "0" do
      (c[i] := c[i - 1] ; c[i - 1] := ".") ;
h)  int L int width = (int c := 1 ;
      while  L10 ↑ (c - 1) < L.1 × L max int do c plus 1 ; c) ;
i)  int L real width= -entier S (L ln (L small real) / L ln (L10)) ;
j)  int L exp width = 1 + entier S
      (L ln (L ln (L max real) / L ln (L10)) / L ln (L10)) ;
k)  proc % L real conv = ( L real x) string :
      (string s := L real string (x, L real width + L exp width + 4,
      L real width - 1, L exp width) ; sign supp zero (s, L real width + 4,
      L real width + L exp width + 3) ; s) ;
l)  proc % nextplc = (file file) : (opened of file |
      (line ended (file) | new line (file)) ;
      (page ended (file) | new page (file)) ;
```

*L int string:*

$$\underbrace{+\overbrace{\mathrm{DDDDDDDDDD}}^{w-1}}_{w}$$

*L dec string:*

$$\underbrace{\overbrace{+\mathrm{DDDDDD}}^{w-d-2}.\overbrace{\mathrm{DDDDDDDD}}^{d}}_{w}$$

*L real string:*

$$\underbrace{\overbrace{+\mathrm{DDDDDDDD}}^{w-d-e-4}.\overbrace{\mathrm{DDDDDD}}^{d}{}_{10}+\overbrace{\mathrm{DDD}}^{e}}_{w}$$

TABLE II: Display of the values of

*L int string*, *L dec string* and *L real string*

*frame*

[*1*] type (1 = integer, 2 = real fixed, 3 = real floating,
4 = complex fixed, 5 = complex floating, 6 = string,
7 = integer choice, 8 = boolean)

[*2*] radix (2, 4, 8, 10 or 16)

[*3*] sign (0 = no sign frame, 1 = sign frame '+', 2 = sign frame '-')

[*4*] number of digits before point; if type = 1 then *w-1*, else if type = 2 or 4 then *w-d-2* else if type = 3 or 5 then *w-d-e-4*, or, if type = 6, then number of characters in string (0, when variable)

[*5*] number of digits after point; if type = 2, 3, 4 or 5 then *d*

[*6*] sign of exponent; if type = 3 or 5 then as [*3*]

[*7*] number of digits of exponent; if type = 3 or 5 then *e*

[*8*], ...,[*14*] as [*1*], ..., [*7*] when *frame*[*1*] = 4 or 5

TABLE III: Significance of the elements of *frame*

10.5.2.2. Formatless input

{For formatless input, *read* and *get* can be used. The elements of the given value of the mode specified by [] union (*intype*, proc (*file*)) are treated one after the other; if an element is a layout procedure, then it is called with the file as its parameter; otherwise, it is straightened (10.5.0.2), and to the resulting names values are assigned, input from the given file as follows:

aa) If the name refers to a value whose mode is specified by *L int*, then, first the file is searched for the first character that is not a space (giving new lines and pages as necessary); next the largest string is read from the file that could be indited under control of some picture or the form *n(k2)d* or *+n(k1)"."n(k2)d*; this string is converted to an integer by *L string int*.

bb) If the name refers to a value whose mode is specified by *L real*, then, first the file is searched for the first character that is not a space; next the largest string is read from the file that could be indited under control of a picture of the form *+n(k1)"."n(k2)d* or *n(k2)d* followed by *.n(k3)d* or *s*. possible followed by *n(k4)"."en(k5)"." +* *n(k6)"."n(k7)d* or *n(k4)"." en(k6)"."n(k7)d*; this string is converted to a real number by *L string real*.

cc) If the name refers to a value whose mode is specified by *L compl*, then, first a real number is input as in bb and assigned to the real part; next the file is searched for the first character that is not a space; next a plus-i-times is expected; finally, a real number is input and assigned to the imaginary part.

dd) If the name refers to a value whose mode is specified by [] *char*, then, if both upper- and lowerstate of the value are one then as many characters are read as the value has elements; if not both states are one, then characters are read from the line under control of the terminatorstring referenced by the file (5.5.1.jj, 10.5.1.mm); the string with those characters as its elements is then the resulting value.

ee) If the name refers to a value whose mode is specified by *char*, then, first, if the line is full a new line is given, and, if the page is full, a new page is given; next the character is read from the file.

ff) If the name refers to a value whose mode is specified by *bool*, then, first the file is searched for the first character that is not a space; then a character is read; if this character is flip (flop), then the resulting value is true (false); if the character is neither flip nor flop, then the further elaboration is undefined. }

a)
```
proc read = ([] union (intype, proc (file)) x) :
  get (stand in, x) ;
```

b)
```
proc get = (file file, [1 : ] union (intype, proc (file)) x) :
  begin intype it ; proc (file) pf ; char k ; priority ! = 8 ;
        for i to upb x do
        (it ::= x[i] ; pf ::= x[i] | pf (file) |
        [1 : ] ref simplout y = straightin it ;
        op ? = (string s) bool :
        (outside (file) | false |  get string (file, k) ;
        char in string (k, loc int, s) |
        true | backspace (file) ; false) ;
        op ! = (string s, char c) char :
         (get string (file, k) ; char in string (k, loc int, s) | k |
          char sugg := c ; (char error of file (sugg) | sugg | c)) ;
        proc skip spaces = : (while (nextplc (file) ; ? ".") do skip) ;
        proc read dig = string :
          (string t :=     ; while ? "0123456789" do t plus k ; t) ;
        proc read num = string :
          (char t := (?"+-" | k | "+") ; while ? "."  do skip ;
          ".100123456789"!"0" + read dig) ;
        proc read real = string :
          (string t := (skip spaces ; read num) ; (? "." | t plus "."
          + read dig ; (? "10" | t plus "10" + read num) ; t)
          for j to upb y do
          (ref bool bb ; ref char cc ; ref string ss ;
          (‡ (ref L int ii ; (ii ::= y[j] |
            val ii := L string int (read num, 10))) ‡) ;
          (‡ (ref L real xx ; (xx ::= y[j]
            val xx := L string real (read real))) ‡) ;
```

10.5.2.2. continued 2

```
        (⸸ (ref L compl zz ; (zz ::= y[j] | get (file, re of zz) ;
          skip spaces ; "⌊"!"⌊" ; get (file, im of zz)))⸸) ;
        (bb ::= y[j] | skip spaces ; val bb :=
        (? "1" | true | "0" ! "0" = "1")) ;
        (cc ::= y[j] | nextplc (file) ; get string (file, cc)) ;
        (ss ::= y[j] |: lws ss ∧ ups ss | get string (file, ss) |
        string t := ; while  (line ended (file) | false |:
        ? term of file | backspace (file) ; false | true)do t plus k ;
        val ss := t[:at(lws ss | lwb ss | lwb t - upb t + upb ss)] ) ;
    end : skip)) end ;
```

c) 
```
proc L string int = ( [1 :      ] char x, int r) L int :
  (r > 1 ∧ r < 17 | L int n := L0 ; L int lr = Kr ; int w = upb x ;
  for i from 2 to w do n := n × lr + K (int d = char dig (x[i]) ;
  (d < r | d)) ; (x[1] = "+" | n |: x[1] = "-" | -n)) ;
```

d) 
```
proc L string real = (string x) L real :
  (int e ; (char in string ("₁₀", e, x) |
  L string dec (x[1 : e - 1]) × L10 ↑ string int (x[e + 1:], 10)|
  L string dec (x))) ;
```

e) 
```
proc L string dec = ([1 :       ] char x) L real : (int w = upb x ;
  L real r := L0 ; int p ; (char in string (".", p, x) |
  [1 : w - 2] char s = x[2 : p - 1] + x[p + 1:] ;
  for i to w - 2 do r := L w × r +
  K (int d = char dig (s[i]) ; (d < 10 | d)) ;
  (x[1] = "+" | r |: x[1] = "-" | -r) × L.1 ↑ (w - p) |
  L string dec (x + "."))) ;
```

f) 
```
proc % char dig = (char x) int :
  (int i ; (char in string (x, i, "0123456789abcdef") | i - 1)) ;
```

g) 
```
proc % outside = (file f) bool : line ended (f) ∨ page ended (f) ∨
  file ended (f) ;
```

10.5.3. Formatted transput

{For the significance of formats see format-denotations (5.5).}

a) 
```
proc format = (file file, tamrof tamrof) :
  (forp of file := 1 ; format of file := collection list pack
  ("(" + F F of tamrof + ")", loc int := 1)) ;
```

```
b)  proc % collection list pack = (string s, ref int p) string :
      (string t := collection (s, p) ;
      while s[p] = "," do t plus "," + collection (s, p) ;
      p plus 1 ; t) ;
c)  proc % collection = (string s, ref int p) string :
      (int n, q ; string f := (p plus 1 ; insertion (s, p)) ;
      q := p ; replicator (s, p, n) ;
      (s[p] = "(" | string t = collection list packs (s, p) ;
      to n do f plus t | p := q ; f plus picture (s, p, loc[1 : 14] int)) ;
      f + insertion (s, p)) ;
d)  proc % insertion = (strings, ref int p) string :
      (int q = p ; skip insertion (s, p) ; s[q : p - 1]) ;
e)  proc % skip insertion = (string s, ref int p) :
      while (p > upb s | false |: skip align (s, p) | true |
      skip lit (s, p)) do skip ;
f)  proc % skip align = (string s, ref int p) bool :
      (int q = p ; replicator (s, p, loc int) ;
      (char in string (s[p], loc int, "x y p l k") |
      p plus 1 ; true | p := q ; false)) ;
g)  proc % replicator = (string s, ref int p, n) :
      (string t := ; while char in string
      (s[p], loc int, "0123456789") do (t plus s[p] ; p plus 1) ;
      n := (t = "" | 1 | string int ("+" + t, 10))) ;
h)  proc % skip lit = (string s, ref int p) bool :
      (int q = p ; replicator (s, p, loc int) ;
      (s[p] = """" | while (s[p plus 1] = """" | s[p plus 1] = """" |
      true) do skip ; true | p := q ; false)) ;
i)  proc % picture  = (string format, ref int p,
      ref[] int frame) string :
    begin int n ; int po = p ;
          op ? = (string s) bool :
             (skip insertion (format, p) ; p > upb format | false |
             int q = p ; replicator (format, p, n) ;
             (format[p] = "s" | p plus 1) ;
             (char in string (format [p], loc int, s) |
             p plus 1 ; true | p := q ; false)) ;
```

```
        proc intreal pattern = (ref[1 : 7] int frame) bool :
          ((num mould (frame[2 : 4]    ) | frame[1]:=  1 ; l) ;
          (? "." |: num mould (frame[3 : 5]    ) | frame[1] := 2 ; l) ;
          (? "e" |: num mould (frame[5 : 7]    ) | frame[1]: := 3 ; l) ;
          false. l : true) ;
        proc num mould = (ref[1 : 3] int frame) bool :
          ((? "r" | frame[1] :=n) ; (? "z" | frame[3] plus n) ;
           (? "+" | frame[2] := 1 |: ? "-" | frame[2] := 2) ;
          while ? "dz" do frame[3] plus n ;
          format[p] = "," v format[p] = "i" v format[p] = ")") ;
        proc string mould = (ref[] int frame) bool : (? "t" | true |
          while ? "a" do frame[4]plus n ; format[p] = "," v
          format[p] = ")") ;
          for i to 14 do frame[i] := 0 ; frame[2] := 10 ;
          (intreal pattern (frame[1 : 7]) | (? "i" |
          frame[1] plus 2 ; intreal pattern (frame[8 : 14])) ; end) ;
          (string mould (frame)| frame[1] := 6 ; end) ;
          (? "b" | frame[1] := 8 |: ? "c"   frame[1] := 7 |
          frame[1] := 0 ; end) ;
         (format[p] = "(" |
          while ? "(," do skip lit (format, p) ; p plus 1) ;
      end: skip insertion (format, p) ; format [po : p - 1]
  end ;
```

{In connection with 10.5.3.i. see Table III.}

10.5.3.1. Formatted output

```
a)  proc outf = (file file, tamrof tamrof,[] outtype x) :
      (format (file, tamrof) ; out (file, x)) ;
b)  proc out = (file file, [1 :] outtype x) :
      begin string format = format of file; ref int p = forp of file ;
            for k to upb x do
            ([1 :] simplout y = straightout x[k] ; int q, j := 0 ;
            [1 : 14]int frame ;
```

```
        rep : j plus 1 ; step :
        while (do insertion (file, format, p) ; p > upb format |
        false | format[p] = ",") do p plus 1 ; (j > upb y | end) ;
        (p > upb format | (¬format end of file | p := 1 ) ; step) ;
        q := p ; picture (format, q, frame) ;
        (frame[1] | int, real, real, compl, compl, string, intch,
        bool) ;
int:    (⫞ (L int i ; (i ::= y[j] |
        edit L int (file, i, format, p, frame) ; rep)) ⫟) ; incomp ;
real:   (⫞ (L real x ; (x ::= y[j] |
        edit L real (file, x, format,p, frame) ; rep)) ⫟) ;
        (⫞ (L int i ; (i ::= y[j] |
        edit L real (file, i, format, p, frame); rep)) ⫟) ; incomp ;
compl:  (⫞ (L compl z ; (z ::= y[j] |
        edit L compl (file, z, format, p, frame) ; rep)) ⫟) ;
        (⫞ (L real x ; (x ::= y[j] |
        edit L compl (file, x, format, p, frame) ; rep)) ⫟) ;
        (⫞ (L int i ; (i ::= y[j] |
        edit L compl (file, i, format, p, frame) ; rep)) ⫟) ; incomp ;
string: ([1 : frame[4]] char s ; char c ;
        (s ::= y[j] |: frame[4] = 0 | put (file, s) |
        edit string (file, s, format, p, frame) ; rep) ;
        (c ::= y[j] |
        edit string (file, c, format, p, frame) ; rep)) ; incomp ;
intch:  (int i ; (i ::= y[j] |
        edit choice (file, i, format, p) ; rep)) ; incomp ;
bool:   (bool b ; b ::= y[j] |
        edit bool (file, b, format, p) ; rep)) ;
incomp: (value error of file | rep | put (file, y[j]) ; undefined) ;
end : skip)
end ;
```

c)
```
proc % edit L int = (file f, L int i, string format,
  ref int p, []int fr) :
  edit string (f, L int string (i, fr[4] + 1, fr[2]), format, p, fr) ;
```

d)
```
proc % edit L real = (file f, L real x, string format,
  ref int p, [] int fr) :
  edit string (f, stringed L real (x, fr), format, p, fr) ;
```

```
e)  proc % stringed real = (L real x, [] int fr) string :
      (fr[1] = 2 | L dec string (x, fr[4] + fr[5] + 2, fr[5])
      L real string (x, fr[4] + fr[5] + fr[7] + 4, fr[5], fr[7])) ;
f)  proc % edit L compl = (file f, L compl z, [] int fr) :
      edit string (f, ([1 : 14] int g := fr ; g  [1]  minus 2 ;
      stringed L real (re z, g[1 : 7]) + "⊥" + stringed L real
      (im z, g[8 : 14])), format, p, fr) ;
g)  proc % edit string = (file f, string x, format,
      ref int p, [] int frame) :
      begin int p1:=  1, n ; bool supp ; string s := x ;
            op ? = (string s) bool:
              (do insertion (file, format, p) ; p > upb format |
              false | int q = p ; replicator (format, p, n) ;
              (supp := format[p] = "s" | p plus 1) ;
              (char in string (format[p], loc int, s) |
              p plus 1 ; true | p := q ; false)) ;
            proc copy = : ((supp | put string (f, s[p1])) ; p1 plus 1) ;
            proc intreal mould = :
              (? "r" ; sign mould (frame[3]) ; int mould ;
              (? "." | copy ; int mould |: s[p1] = "." | p1 plus 1) ;
              (? "e" | copy ; sign mould (frame[6]) ; int mould)) ;
            proc sign mould = (int sign) : (sign = 0 | p1 plus 1 |
               s[p1] := (s[p1] = "+" | (sign | "+", "⊥") | "-") ;
              (? "z" | sign supp zero (s, p1, p1 + n) | n := 0) ;
              to n + 1 do copy ; p plus 1) ;
            proc int mould = :
              (l : (? "z" | bool zs := true ; to n do
              (s[p1] = "0" ∧ zs | put string (file, "⊥") ;
              p1 plus 1 | zs := false ; copy) ; l) ;
              (? "d" | to n do copy ; l)) ;
            proc string mould = :
              while ? "a" do to n do copy ;
      tes: (frame[1] = 6 | string mould |: intreal mould ;
              frame[1] > 3 | p plus 1 ; copy ; intreal mould)
      end ;
```

10.5.3.1. continued 3

```
h)  proc % edit choice = (file f, int c, string format, ref int p) :
      (c > 0 | do insertion (f, format, p) ; p plus 2 ;
      to c - 1 do (skip lit (format p) ; format[p] = ","
      p plus 1 | undefined) ;
      do lit (f, format, p) ;
      while format[p] ≠ ")" do (p plus 1 ; skip lit (format, p)) ;
      p plus 1 | undefined) ;
i)  proc % edit bool = (file f, bool b, string format, ref int p) :
      (do insertion (f, format, p) ; (format[p + 1] = "(" |
      p plus 2 ; (b | do lit (f, format, p) ; p plus 1 ; skip lit
      (format, p) | skip lit (format, p) ; p plus 1 ; do lit (f, format, p) |
      put string (f, (b | "1" | "0"))) ; p plus 1) ;
j)  proc % do insertion = (file f, string s, ref int p) :
      while (p > upb s | false |: do align (f, s, p) | true |
      do lit (f, s, p)) do skip ;
k)  proc % do align = (file f, string s, ref int p) bool:
      (int q = p ; int n ; replicator (s, p, n) ;
      (s[p] = "x" | to n do space (f) ; l |:
      s[p] = "y" | to n do backspace (f) ; l |:
      s[p] = "p" | to n do new page (f) ; l |:
      s[p] = "l" | to n do new line (f) ; l |:
      s[p] = "k" | char of bfile of f := n ; l) ; p := q ; false.
      l : p plus 1 ; true) ;
l)  proc % do lit = (file f, string s, ref int p) bool :
      (int q = p ; int n ; replicator (s, p, n) ; (s[p] = """"" |
      while (s[p plus 1] = """"" | s[p plus 1] = """"" | true) do
      put string (f, s[p]) ; true | p := q ; false)) ;
```

10.5.3.2. Formatted input

```
a)  proc inf = (file file, tamrof tamrof, [] intype x) :
      (format (file, tamrof) ; in (file, x)) ;
b)  proc in = (file file, [1 :] intype x) :
    begin string format = format of file ; ref int p = forp of file ;
          for k to upb x do
          ([1 :] ref simplout y = straightin x[k] ; int q, j := 0 ;
          [1 : 14] int frame ;
```

```
          rep : j plus 1 ; step :
          while ( exp insertion (file, format, p) ; p > upb format|
          false | format[p] = ",") do p plus 1 ; (j ≥ upb y | end) ;
          (p > upb format | ( format end of file | p := 1) ; step) ;
          q := p ; picture (format, q, frame) ;
          (frame[1] | int, real, real, compl, compl, string, intch, bool) ;
  int:    (⌈ (ref L int ii ; (ii ::= y[j] |
          indit L int (file, ii, format, p, frame) ; rep)) ⌉) ; incomp ;
  real:   (⌈ (ref L real xx ; (xx ::= y[j] |
          indit L real (file, xx, format, p, frame) ; rep)) ⌉) ; incomp ;
  compl:  (⌈ (ref L compl zz ; (zz ::= y[j] |
          indit L compl (file, zz, format, p, frame) ; rep)) ⌉) ; incomp ;
  string: (ref string ss ; ref char cc ; [1 : frame[4]] char t ;
          (frame[4] = 0 |: ss ::= y[j] | get (file, ss) ; rep | incomp ;
          indit string (file, t, format, p, frame) ;
          (ss ::= y[j] | val ss := t ; rep|:  cc ::= y[j] |
          val cc := t[1] ; rep)) ; incomp ;
  intch:  (ref int ii ; (ii ::= y[j] |
          indit choice (file, ii, format, p) ; rep)) ; incomp ;
  bool:   (ref bool bb ; (bb ::= y[j] |
          indit bool (file, bb, format, p) ; rep)) ;
  incomp: (value error of file | rep | undefined) ;
  end : skip)
  end ;
```

c) 
```
proc % indit L int =
  (file f, ref L int i, string format, ref int p, [] int fr) :
  (string t ; indit string (f, t, format, p, fr) ;
  i := L string int (t, fr[2])) ;
```

d) 
```
proc % indit L real =
  (file f, ref L real x, string format, ref int p, [] int fr) :
  (string t ; indit string (f, t, format, p, fr) ;
  x := L string real (t)) ;
```

e) 
```
proc % indit L compl =
  (file f, ref L compl z, string format, ref int p, [] int fr) :
  (string t ; int i ; indit string (f, t, format, p, fr) ;
  z := (char in string ("⊥", i, t) |
  (L string real (t[1 : i - 1]) ⊥ L string real (t[i + 1 : ])))) ;
```

```
f)  proc % indit string =
      (file f, ref string t, string format, ref int p, [] int frame) :
      begin int n ; bool supp ; char k ; string x := ;
            op ? = (string s) bool :
              (exp insertion (format, p) ; p > upb format | false |
              int q = p ; replicator (format, p, n) ;
              (supp := format[p] = "s" | p plus 1) ;
              (char in string (format[p], loc int, s) |
              p plus 1 ; true | p := q ; false)) ;
              priority ! = 8 ;
            op ! = (string s, char c) string :
              (char in string (next, loc int, s) | (supp | "" | k)|
              char sugg := c ; (char error (sugg) | sugg | c)) ;
            proc next = char : (get string (f, k) ; k) ;
            proc intreal mould = :
              (? "r" ; sign mould (frame[3]) ; int mould ;
              (? "." | x plus "." ! "." ; int mould ;
              (? "e" | x plus "10" ! "10" ; sign mould (frame[6]) ;
              int mould)) ;
            proc sign mould = (int sign) : (sign = 0 | x plus "+" |
              int j ; (¬ ? "z" | n := 0) ; for i to n + 1
              while next = "." do j := i ;
              x plus (sign = 1 | " +-" ! "+" | : k = "-" | k |:
              backspace (f) ; j > 0 | j minus 1 ; "+" | "+" ! "+") ;
              for i from j + 1 to n + 1 do x plus "0123456789"!"0") ;
            proc int mould = : (l :
              (? "z" | int j ; for i to n while next = "." do j := i ;
              backspace (f) ;
              from j to n do x plus "0123456789" ! "0" ; l) ;
              (? "d" | to n do x plus "0123456789" ! "0" ; l)) ;
            proc string mould = while? "a" do to n do x plus
              (supp | "." | next) ;
   tis:     (frame[1] = 6 | string mould |: intreal mould ; frame[1] > 3 |
            "|" ! "|" ; intreal mould) ;
            t := x ;
   end ;
```

```
g)  proc % indit choice =
      (file f, ref int c, string format, ref int p) :
      (exp insertion (f, format, p) ; p plus 2 ; c := 1 ;
      while  ask lit (f, format, p) do
      (c plus 1 ; format[p] = "," | p plus 1 | undefined) ;
      while format[p] ≠ ")" do (p plus 1 ; skip lit (format, p)) ;
      p plus 1 ; exp insertion (f, format, p)) ;
h)  proc % indit bool =
      (file f, ref bool b, string format, ref int p) :
      (exp insertion (f, format, p) ; (format[p + 1] = "(" |
      p plus 2 ; (b := ask lit (f, format, p) |
      p plus 1 ; skip lit (format, p) |:
      p plus 1 ;  ask lit (f, format, p) | undefined) |
      char k ; get string (f, k) ; b := (k = "1" | true |
      k = "0" | false)) ;
      p plus 1 ; exp insertion (f, format, p)) ;
i)  proc % exp insertion = (file f, string s, ref int p) :
      while (p > upb s | false |: do align (f, s, p) | true |
      exp lit (f, s, p)) do skip ;
j)  proc % exp lit = (file f, string s, ref int p) bool :
      (int q = p ; int n ; replicator (s, p, n) ;
      (s[p]   """"" | int r = p ; to n do (p := r ;
      while (s[p plus 1] = """"" | s[p plus 1] | """"" | true) do
      (char k ; get string (f, k) ; k ≠ s[p] | undefined)) ; true |
      p := q ; false)) ;
k)  proc % ask lit = (file f, string s, ref int p) bool :
     (int c = char of f ; int n ; replicator (s, p, n) ;
      (s[p] = """"" | int r = p ; to n do (p := r ;
      while (s[p plus 1] = """"" | s[p plus 1] = """"" | true) do
      (char k ; get string (f, k) ; k ≠ s[p] | l)) ; true.
      l : while (s[p plus 1] = """"" | s[p plus 1] = """"" | true) do skip ;
      char of f := c ; false)) ;
```

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

10.5.4. Binary transput

```
a)  proc % to bin = (file f, simplout x) [] int :
      c a value of mode 'row-of-integral' whose lower bound is one,
        and whose upper bound depends on the value of 'f' and on the
        mode of the value of 'x' ; furthermore,
        x = from bin (f, x, to bin (f, x)) c ;
b)  proc % from bin = (file f, simplout v, [] int y) simplout :
      c a value, if one exists, of the mode of the actual parameter
        corresponding to v, such that
        y = to bin (f, from bin (f, v, y)) c ;
```

{On some channels a more straightforward way of transput is available. Some properties of this binary transput depend on the particular implementation, others can be deduced from 10.5.4. }

10.5.4.1. Binary output

```
a)  proc write bin = ([] outtype x) : put bin (stand back, x) ;
b)  proc put bin = (file file, [1 :] outtype x) :
      if bin possible[chan of file] ∧ opened of file
      then if¬ set possible[chan of file] thef state def of file
        then (state get of file ∨¬ state bin of file | undefined)
        else state def of file := state bin of file := true ;
             state get of file := false
        fi ;
        for k to upb x do
        ([1 :] simplout y = straightout x[k] ;
        for j to upb y do
        ([1 :] int bin = to bin (file, y[j]); bfile b = bfile of file ;
        ref int p = page of b, l = line of b, c = char of b ;
        for i to upb bin do (next plc (file) ;
        book of b[p, l, c] := bin[i] ; c plus 1 ;
        (p = lpage of b ∧ l = lline of b |
          (c > lchar of b | lchar of b := c) |
          lpage of b := p ; lline of b := l ; lchar of b := c))))
      else undefined
      fi ;
```

10.5.4.2. Binary input

```
a)  proc read bin = ([] intype x) : get bin (stand back, x) ;
b)  proc get bin = (file file, [1 :] intype x) :
      if bin possible[chan of file] ∧ opened of file
      then if¬ set possible[chan of file] thef state def of file
        then (¬state get of file ∨¬ state bin of file | undefined)
        else state def of file := state bin of file :=
              state get of file := true
        fi ;
        for k to upb x do
        ([1 :] ref simplout y = straightin x[k] ;
        for j to upb y do
        ([1 :] int bin := to bin (file, y[j]); bfile b = bfile of file ;
        fcr i to upb bin do (next plc (file) ; check plc (file) ;
        bin[i] := bool of b[page of b, line of b, char of b] ;
          char of b plus 1) ;
        (⊦ (ref L int ii ; (ii ::= y[j] |
          val ii ::= from bin (file, ii, bin))) ⊣) ;
        (⊦ (ref L real xx ; (xx ::= y[j] |
          val xx ::= from bin (file, xx, bin))) ⊣) ;
        (⊦ (ref L compl zz ; (zz ::= y[j] |
          val zz ::= from bin (file, zz, bin))) ⊣) ;
        (ref string ss ; (ss ::= y[j] |
          val ss ::= from bin (file, ss, bin))) ;
        (ref char cc ; (cc ::= y[j] |
          val cc ::= from bin (file, cc, bin))) ;
        (ref bool bb ; (bb ::= y[j] |
          val bb ::= from bin (file, bb, bin))) ))
      else undefined
      fi ;
```

{But Eeyore wasn't listening. He was taking the balloon out, and putting it back again, as happy as could be. ...

Winnie-the-Pooh, A.A. Milne. }

## 11. Examples

### 11.1. Complex square root

A declaration in which *compsqrt* is a procedure-with-[complex]-parameter-[complex]-mode-identifier (here [complex] stands for structured-with-real-field-letter-r-letter-e-and-real-field-letter-i-letter-m.) :

a) <u>proc</u> *compsqrt* = *(*<u>compl</u> *z)* <u>compl</u> : <u>c</u> *the square root whose real part is nonnegative of the complex number z* <u>c</u>

b) <u>begin</u> <u>real</u> *x* = <u>re</u> *z, y* = <u>im</u> *z ;*

c) <u>real</u> *rp = sqrt ((*<u>abs</u> *x + sqrt (x ↑ 2 + y ↑ 2)) / 2) ;*

d) <u>real</u> *ip = (rp = 0 | 0 | y / (2 × rp)) ;*

e) *(x ≥ 0 | rp* <u>⊥</u> *ip |* <u>abs</u> *ip* <u>⊥</u> *(y ≥ 0 | rp | −rp))*

f) <u>end</u>

[complex]-calls {8.6.2} using *compsqrt*:

g) *compsqrt (w)*

h) *compssqrt (−3.14)*

i) *compsqrt (−1)*

11.2. Innerproduct1

A declaration in which *innerproduct1* is a procedure-with-integral-parameter-and-procedure-with-integral-parameter-real-parameter-and-procedure-with-integral-parameter-real-parameter-real-mode-identifier:

a) *proc innerproduct1 = (int n, proc (int) real x, y) real :*
*comment the innerproduct of two vectors, each with n components, x(i), y(i), i = 1, ..., n, where x and y are arbitrary mappings from integer to real number comment*

b) *begin long real s := long 0 ;*

c) *for i to n do s plus leng x(i) × leng y(i) ;*

d) *short s*

e) *end*

Real-calls {8.6.2} using *innerproduct1*:

f) *innerproduct1 (m, (int j) real : x1[j], (int j) real : y1[j])*

g) *innerproduct1 (n, nsin, ncos)*

11.3. Innerproduct2

A declaration in which *innerproduct2* is a procedure-with-reference-to-row-of-real-parameter-and-reference-to-row-of-real-parameter-real-mode-identifier:

a) *proc innerproduct2 = (ref[1 :] real a ; ref[1 : upb a] real b) real) :*
*c the innerproduct of two vectors a and b with equal number of elements c*

b) *begin long real s := long 0 ;*

c) *for i to upb a do s plus leng a[i] × leng b[i] ;*

d) *short s*

e) *end*

Real-calls using *innerproduct2*:

f) *innerproduct2 (x1, y1)*

g) *innerproduct2 (y2[2], y2[, 3])*

11.4. Innerproduct3

A declaration in which *innerproduct3* is a procedure-with-reference-to-integral-parameter-and-integral-parameter-and-procedure-real-parameter-and-procedure-real-parameter-real-mode-identifier:

```
a)  proc innerproduct3 = (ref int i, int n, proc real xi, yi) real :
      comment the innerproduct of two vectors whose n elements are the
      values of the expressions xi and yi and which depend, in general,
      on the value of i comment
b)  begin long real s := long 0 ;
c)  for k to n do (i := k ; s plus leng xi × leng yi) ;
d)  short s
e)  end
```

A real-call using *innerproduct3*:

```
f)  innerproduct3 (j, 8, x1[j], y1[j + 1])
```

11.5. Largest element

A declaration in which *absmax* is a procedure-with-reference-to-row-of-row-of-real-parameter-and-reference-to-real-parameter-and-reference-to-integral-parameter-and-reference-to-integral-parameter-mode-identifier:

```
a)  proc absmax = (ref[1 :, 1 :] real a,
b)  c result c ref real y, c subscripts c ref int i, k) :
      comment the absolute value of the element of greatest absolute value
      of the matrix a is assigned to y, and the subscripts of this element
      to i and k comment
c)  begin y := -1 ;
d)  for p to 1 upb a do for q to 2 upb a do
e)  if abs a[p, q] > y then y := abs a[i := p, k := q] fi
f)  end
```

Void-calls {8.6.2} using *absmax*:

```
g)  absmax (x2, x, i, j)
h)  absmax (x2, x, loc int, loc int)
```

## 11.6. Euler summation

```
a)  proc euler = (proc (int) real f, real eps, int tim) real:
      comment the sum for i from 1 to infinity of f(i), computed by means
      of a suitably refined Euler transformation. The summation is
      terminated when the absolute values of the terms of the transformed
      series are found to be less than eps tim times in succession. This
      transformation is particularly efficient in the case of a slowly
      convergent or divergent alternating series comment
b)  begin int n := 1, t; real mn, ds := eps; [1 : 16] real m ;
c)         real sum := (m[1] := f(1)) / 2 ;
d)         for i from 2 while (t := (abs ds < eps | t + 1 | 1)) ≤ tim do
e)           begin mn := f(i) ;
f)             for k to n do begin mn := ((ds := mn) + m[k]) / 2 ;
g)                                  m[k] := ds end;
h)             sum plus (ds := (abs mn < abs m[n] ∧ n < 16 |
i)                            n plus 1 ; m[n]:= mn ; mn / 2 | mn))
j)           end ;
k)         sum
l)  end
```

A call using *euler*:

```
m)  euler ((int i) real : (odd i | -1 / i | 1 / i), 1_10-5, 2)
```

## 11.7. The norm of a vector

```
a)  proc norm = (ref[1 :] real a) real :
      c the euclidean norm of the vector a c
b)     (long real s := long 0 ;
c)     for k to upb a do s plus leng a[k] ↑ 2 ;
d)     short long sqrt(s))
```

For a use of *norm* as a call, see 11.8.d.

11.8. Determinant of a matrix

```
a)  proc det = (ref[1 :, 1 :] real a, ref[1 : upb a] int p) real :
b)     if upb a = 2 upb a
c)     then int n = upb a ;
          comment the determinant of the square matrix a of order n by
          the method of Crout with row interchanges: a is replaced by its
          triangular decomposition l × u with all u[k, k] = 1. The vector p
          gives as output the pivotal row indices; the k-th pivot is chosen
          in the k-th column of l such that abs l[i, k] / row norm is
          maximal comment
d)          [1 : n] real v ; real d := 1, r := -1, s, pivot ;
e)          for i to n do v[i] := norm (a[i]) ;
f)          for k to n do
g)             begin int k1 = k - 1 ; ref int pk = p[k] ;
h)             ref[,] real al = a[, 1 : k1], au = a[1 : k1] ;
i)             ref[] real ak = a[k], ka = a[, k], apk = a[pk],
j)                alk = al[k], kau = au[, k] ;
k)             for i from k to n do
l)                begin ref real aik = ka[i] ;
m)                if (s := abs (aik minus innerproduct 2 (al[i], kau)) /
                     v[i]) > r
n)                   then r := s ; pk := i fi
o)                end ;
p)             v[pk] := v[k] ;pivot := ka[pk] ;
q)             for j to n do
r)                begin ref real akj = ak[j], apkj = apk[j] ;
s)                r := akj ; akj := if j ≤ k then apkj
t)                else (apkj - innerproduct2 (alk, au[, j])) / pivot fi ;
u)                if pk ≠ k then apkj := -r fi
v)                end ;
w)             d times pivot
x)             end ;
y)          d
z)     fi
```

A call using *det*:

aa) *det (y2, i1)*

11.9. Greatest common divisor

An example of a recursive procedure:

```
a) proc gcd = (int a, b) int :
     c the greatest common divisor of two integers c
b)      (b = 0 | abs a | gcd (b, a ÷: b))
```

A call using *gcd*:

```
c) gcd (n, 124)
```

11.10. Continued fraction

An example of a recursive operation:

```
a) op / = ([1 :] real a ; [1 : upb a] real b) real :
     comment the value of a/b is that of the continued fraction
     a1 / (b1 + a2 / (b2 + ... an / bn)...) comment
b)      (upb a = 1 | a[1] / b[1] | a[1] / (b[1] + a[2 :] / b[2 : 1]))
```

A formula using /:

```
c) x1 / y1
```

{The use of recursion may often be elegant rather than efficient as in 11.9 and 11.10. See, however, 11.11 and 11.14 for examples in which recursion is of the essence.}

## 11.11. Formula manipulation

```
a)  begin union form = (ref const, ref var, ref triple, ref call) ;
b)  struct const = (real value) ;
c)  struct var = (string name, real value) ;
d)  struct triple = (form left operand, int operator, form right operand) ;
e)  struct function = (ref var bound var, form body) ;
f)  struct call = (ref function function name, form parameter) ;
g)  int plus = 1, minus = 2, times = 3, by = 4, to = 5 ;
h)  const zero, one ; value of zero := 0 ; value of one  = 1 ;
i)  op = = (form a, ref const b) bool :
      (ref const ec ; (ec ::= a | val ec :=: b | false)) ;
j)  op + = (form a, b) form :
      (a = zero | b |: b = zero | a | triple := (a, plus, b)) ;
k)  op - = (form a, b) form : (b = zero | a | triple := (a, minus, b)) ;
l)  op × = (form a, b) form :
      (a = zero ∨ b = zero | zero |: a = one | b |: b = one | a |
                                   triple := (a, times, b)) ;
m)  op / = (form a, b) form :
      (a = zero ∧    b = zero  | zero |: b = one | a | triple := (a, by,b)) ;
n)  op ↑ = (form a, ref const b) form :
      (a = one ∨ (b :=: zero) | one |: b :=: one | a | triple := (a, to, b)) ;

o)  proc derivative of = (form e, c with respect to c ref var x) form :
p)  begin ref const ec ; ref var ev ; ref triple et ; ref call ef ;
q)    case ec, ev, et, ef ::= e in
r)          zero comma
s)          (val ev :=: x | one | zero) comma
t)          (form u = left operand of et, v = right operand of et,
u)          udash = derivative of (u, c with respect to c x),
v)          vdash = derivative of (v, c with respect to c x) ;
w)          case operator of et in
x)            udash + vdash, udash - vdash,
y)            u × vdash + udash × v, (udash - et × vdash) / v,
z)            (ec ::= v | v × u ↑ (const c ;
aa)           value of c := value of ec - 1 ; c) × udash) esac) comma
```

11.11. continued

```
ab)          ref function f = function name of ef ;
ac)          form g = parameter of ef ;
ad)          ref var y = bound var of f ;
ae)          function fdash := (y, derivative of (body of f, y)) ;
af)          (call := (fdash, g)) × derivative of (g, x)
ag)      esac
ah)    end c derivative c ;
ai)    proc value of = (form e) real :
aj)      begin ref const ec ; ref var ev ; ref triple et ; ref call ef ;
ak)        case ec, ev, et, ef ::= e in
al)            value of ec comma
am)            value of ev comma
an)            (real u = value of (left operand of et),
ao)                  v = value of (right operand of et) ;
ap)            case operator of et in
aq)                 u + v, u - v, u × v, u / v, exp (v × ln(u)) esac)
ar)            comma
as)            ref function f = function name of ef ;
at)            value of bound var of f := value of (parameter of ef) ;
au)            value of (body of f)
av)        esac
aw)      end c value of c ;
ax)    form f, g ; var a := ("a", skip), b := ("b", skip), x := ("x", skip) ;
ay)    start here :
az)    read ((value of a, value of b, value of x)) ;
ba)    f := a + x / (b + x) ; g := (f + one) / (f - one) ;
bb)    print ((value of a, value of b, value of x,
               value of (derivative of (g, c with respect to c x))))
bc)    end
```

11.12. Information retrieval

```
a)   begin c authors and titles enquiry system c
            struct book = (string title, ref book next),
                   auth = (string name, ref auth next, ref book book) ;
b)          ref book book ; ref auth auth, first auth := nil, last auth ;
c)          string name, title ; int i ; file input, output ;
d)          open (input,,remote in) ; create (output, remote out) ;
e)          put (output, (reset,
f)            "to.enter.a.new.author,.type.""author"",.
              a.space,.and.his.name." , new line,
g)            "to.enter.a.new.book,.type.""book"",.a.space,.the.name.
              of.the.author,.a.new.line.and.the.title." , new line,
h)            "for.a.listing.of.the.books.by.an.author,.type.""list"",.
              a.space,.and.his.name." , new line,
i)            "to.find.the.author.of.a.book,.type.""find"",.
              a.new.line.and.the.title." , new line,
j)            "to.end,.type.""end""." , new line)) ;
k)          proc update = expr if val first auth :=: nil
l)          then auth := first auth := last auth := auth :=(name, nil, nil)
m)          else auth := first auth ; while val auth :≠: nil do
n)                  (name = name of auth | known | auth := next of auth) ;
o)                last auth := next of last auth := auth :=
                    auth := (name, nil, nil) ;
p)          known: skip fi ;
q)   client: inf (input, fc ("author","book","list","find","end",""),
               x30al, 80al, f, i) ;
r)            case i in author, publ., list, find, end, error esac ;
s)   author: in (input, name) ; update ; client ;
t)   publ:    in (input, (name, title)) ; update ;
u)            if val book of auth :=: nil
v)            then book of auth := book := title, nil)
w)            else book := book of auth ; while val next of book :≠: nil do
x)                  (title = title of book | client | book := next of book) ;
y)                  (title ≠ title of book | next of book := book :=
                    (title, nil))
z)            fi ; client ;
```

11.12. continued

```
aa)  list:  in (input, name) ; update ;
ab)         outf (output, fp"author:."30allf, name) ;
ac)         if val (book := book of auth) :=: nil
ad)         then put (output, "no.publications")
ae)         else while val book :≠: nil do
af)           begin if line number (output) = max line[remote out]
ag)                 then outf (output, f41k"continued.on.next.page"p
                                  "author:."30a41k"continued"llf, name)
ah)                 fi ; outf (output, f80alf, title of book) ;
ai)                 book := next of book
aj)           end
ak)         fi ; client ;
al)  find:  in (input, (loc string, title)) ; auth := first auth ;
am)         while val auth :≠: nil do
an)           begin book := book of auth ; while val book :≠: nil do
ao)                 if title = title of book
ap)                 then outf(output, fl"author:."30af,
                                 name of auth) ; client
aq)                 else book := next of book
ar)                 fi ; auth := next of auth
as)           end ; outf (output, fll"unknown"f,) ;
at)         client ;
au)  end:   put (output, (new page, "signed.off")) ;
av)         close(input,) ; close (output,).
aw)  error: put (output, (new line, "mistake,.try.again.")) ;
ax)         new line (input) ; client
ay)  end
```

11.13. Cooperating sequential processes

```
begin int nmb magazine slots, nmb producers, nmb consumers ;
          read ((nmb magazine slots, nmb producers, nmb consumers)) ;
          [1 : nmb producers]file infile, [1 : nmb consumers]file outfile ;
        for i to nmb producers do
          open (infile[i], inchannel[i]) ;
          c the multiple values inchannel and outchannel are
          defined in a surrounding range c
        for i to nmb consumers do
          open (outfile[i], outchannel[i]) ;
        mode page = [1 : 60, 1 : 132] char ;
        [1 : nmb magazine slots] ref page magazine ;
        int c pointers of a cyclic magazine c
          index := 1, exdex := 1,
            c general semaphores c
          full slots := 0, free slots := nmb magazine slots,
            c binary semaphores c
          in buffer busy := 1, out buffer busy := 1 ;
        proc par call = (proc (int) p, int n) :
          c calls n incarnations of p in parallel c
          (n > 1 | par (p (n), par call (p, n - 1)) | p (1)) ;
        proc producer = (int i) :
           do (page page ; get (infile[i], page) ;
          down free slots ; down in buffer busy ;
          magazine[index] := page ;
          index modb nmb magazine slots plus 1 ;
          up full slots ; up in buffer busy)  ;
        proc consumer = (int i) :
           do (page page ; down full slots ;
          down out buffer busy ; page := magazine[exdex] ;
          exdex modb nmb magazine slots plus 1 ;
          up free slots ; up out buffer busy ;
          put (outfile[i], page))  ;
        par (par call (producer, nmb producers),
             par call (consumer, nmb consumers)
end
```

11.14. Towers of Hanoi

```
a) begin proc p = (int me, de, ma) :
b)         if ma > 0
c)         then p (me, 6-me-de, ma - 1) ;
d)           out (stand out, (me, de, ma)) ;
e)           c move from peg 'me' to 'peg 'de' piece number 'ma' c
f)           p (6-me-de, de, ma - 1)
g)         end ;
h)           for k to 7 do
i)           (outf (stand out, fl"k="2zdl,
                       n (max int) (2(2(4(3(d)x)x)x)l) f, k) ;
j)           p (1,2,k))
k) end
```

```
K=  1
121

K=  2
131 122 321

K=  3
121 132 231 123  311 322 121

K=  4
131 122 321 133  211 232 131 124  321 312 211 323  131 122 321

K=  5
121 132 231 123  311 322 121 134  231 212 311 233  121 132 231 125
311 322 121 313  231 212 311 324  121 132 231 123  311 322 121

K=  6
131 122 321 133  211 232 131 124  321 312 211 323  131 122 321 135
211 232 131 213  321 312 211 234  131 122 321 133  211 232 131 126
321 312 211 323  131 122 321 314  211 232 131 213  321 312 211 325
131 122 321 133  211 232 131 124  321 312 211 323  131 122 321

K=  7
121 132 231 123  311 322 121 134  231 212 311 233  121 132 231 125
311 322 121 313  231 212 311 324  121 132 231 123  311 322 121 136
231 212 311 233  121 132 231 214  311 322 121 313  231 212 311 235
121 132 231 123  311 322 121 134  231 212 311 233  121 132 231 127
311 322 121 313  231 212 311 324  121 132 231 123  311 322 121 315
231 212 311 233  121 132 231 214  311 322 121 313  231 212 311 326
121 132 231 123  311 322 121 134  231 212 311 233  121 132 231 125
311 322 121 313  231 212 311 324  121 132 231 123  311 322 121
```

## 12. Glossary

Given below are the locations of the first, and sometimes other, instructive appearances of a number of words which, in Chapters 1 up to 10 of this Report, have a specific technical meaning. A word appearing in different grammatical forms (e.g. "contain", "contains", "contained", "containing") is given once, usually as infinitive (e.g. "contain").

action 2.2, 2.2.5
ALGOL 68 program 4.4
apostrophe 1.1.6.c
applied occurrence 4.1.2.a
appoint 6.0.2.a
a priori value 5.1.0.2.b
arithmetic value 2.2.3.1.a
assign 2.2.2.l, 8.3.1.2.c
asterisk 1.1.2.a
automaton 1.1.1.a
backfile 10.5.1.aa, cc
balance 6
blind alley 1.1.2.d
case clause 9.4.b, c, d, e
channel 10.5.1.aa, bb
character 2.2.3.1.a, f
close a file 10.5.1.ii
coercend 8
collateral 2.2.5, 6.2.2.a
colon 1.1.2.a
comma 1.1.2.a
compatible 5.5.1.dd, nn
compile 2.3.c
component of 2.2.2.h
composite 3.1.2.d
computer 1.1.1.a
constant 5
constituent 1.1.6.e
contain 1.1.6.b
conversion-key 5.5.1.ff
copy 2.2.4.1.a
create a file 10.5.1.gg
defining occurrence 2.2.2.c
4.10.2.a
denote 1.1.6.c
deproceduring 8.2, 8.2.2
dereferencing 8.2, 8.2.1
describe 2.2.3.3.b
descriptor 2.2.3.3.a
direct constituent 1.1.6.e
direct production 1.1.2.c
divided by 2.2.3.1.c
edit 5.5.1.ll
elaborate collaterally 6.2.2.a
elaboration 1.1.6.h, 6.0.2.a
element 2.2.2.k
elementary 2.2.5
encountered 4.4.4.b
end of file 10.5.1.cc
English language 1.1.1.b
envelope 1.1.6.j
environment enquiry 10.1
equivalent to 2.2.2.h
establish a file 10.5.1.gg
expect 5.5.1.gg
extended language 1.1.1.a
extension 1.1.7
external object 2.2.1
false 2.2.3.1.e
field 2.2.2.k

12 continued

12 continued

12 continued



{Denn eben, wo Begriffe fehlen,
Da stellt ein Wort zur rechten Zeit sich ein.
Faust, J.W. von Goethe.}

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM