4.2.2. Examples

```
s := p[0] := n := n+1+s
n := n+1
A := B/C−v−q×S
S[v,k+2] := 3−arctan(s×zeta)
V := Q>Y∧Z
```

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is **Boolean,** the expression must likewise be **Boolean.** If the type is **real** or **integer,** the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type, the transfer function is understood to yield a result equivalent to

entier(E+0.5)

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3. Go To Statements

4.3.1. Syntax

```
⟨go to statement⟩ ::= go to ⟨designational expression⟩
```

4.3.2. Examples

```
go to 8
go to exit [n+1]
go to Town[if y<0 then N else N+1]
go to if Ab<c then 17 else q[if w<0 then 2 else n]
```

4.3.3. Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5. Go to an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy Statements

4.4.1. Syntax

```
⟨dummy statement⟩ ::= ⟨empty⟩
```

4.4.2. Examples

```
L:
begin ... ; John: end
```

4.4.3. Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5. Conditional Statements

4.5.1. Syntax

```
⟨if clause⟩ ::= if ⟨Boolean expression⟩ then
⟨unconditional statement⟩ ::= ⟨basic statement⟩|
    ⟨compound statement⟩|⟨block⟩
⟨if statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩ ::= ⟨if statement⟩|⟨if statement⟩ else
    ⟨statement⟩|⟨if clause⟩⟨for statement⟩|
    ⟨label⟩ : ⟨conditional statement⟩
```

4.5.2. Examples

```
if x>0 then n := n+1
if v>u then V: q := n+m else go to R
if s<0∨P≦Q then AA: begin if q<v then a := v/s
            else y := 2×a end
          else if v>s then a := v−q else if v>s−1
            then go to S
```

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

```
if B1 then S1 else if B2 then S2 else S3  ;  S4
```

and

```
if B1 then S1 else if B2 then S2 else if B3 then S3  ;  S4
```

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e. the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

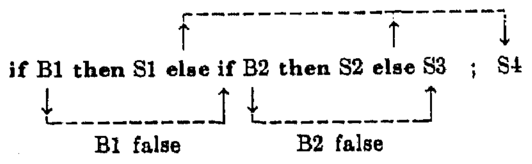
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:


if B1 **then** S1 **else if** B2 **then** S2 **else** S3 ; S4
B1 false
B2 false

4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. FOR STATEMENTS

4.6.1. Syntax

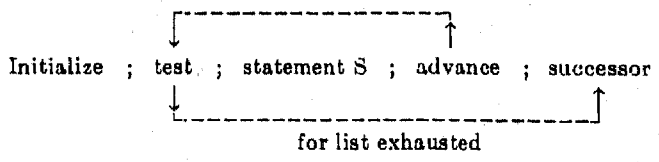
```
⟨for list element⟩ ::= ⟨arithmetic expression⟩|
    ⟨arithmetic expression⟩ step ⟨arithmetic expression⟩ until
    ⟨arithmetic expression⟩|⟨arithmetic expression⟩ while
    ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩|⟨for list⟩ , ⟨for list element⟩
⟨for clause⟩ ::= for ⟨variable⟩ := ⟨for list⟩ do
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩|
    ⟨label⟩:⟨for statement⟩
```

4.6.2. Examples

```
for q := 1 step s until n do A[q] := B[q]
for k := 1, V1×2 while V1<N do
        for j := I+G, L, 1 step 1 until N, C+D do
            A[k,j] := B[k,j]
```

4.6.3. Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element of the form A **step** B **until** C, where A, B, and C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```
    V := A  ;
L1: if (V−C)× sign(B)>0 then go to element exhausted;
    statement S  ;
    V := V+B  ;
    go to L1  ;
```

where V is the controlled variable of the for clause and *element exhausted* points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
L3: V := E  ;
    if ¬F then go to element exhausted  ;
    Statement S  ;
    go to L3  ;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. PROCEDURE STATEMENTS

4.7.1. Syntax

```
⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩|
    ⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩
```

```
⟨parameter delimiter⟩ ::= ,|)⟨letter string⟩:(
⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
    ⟨actual parameter list⟩⟨parameter delimiter⟩
    ⟨actual parameter⟩
⟨actual parameter part⟩ ::= ⟨empty⟩|
    (⟨actual parameter list⟩)
⟨procedure statement⟩ ::= ⟨procedure identifier⟩
    ⟨actual parameter part⟩
```

4.7.2. Examples

Spur (*A*)Order: (7)Result to: (*V*)
Transpose (*W*,*v*+1)
Absmax(*A*,*N*,*M*,*Yy*,*I*,*K*)
Innerproduct(*A*[*t*,*P*,*u*],*B*[*P*],10,*P*,*Y*)

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2. Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4. Actual-formal correspondence

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator **own**. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

```
⟨declaration⟩ ::= ⟨type declaration⟩|⟨array declaration⟩|
    ⟨switch declaration⟩|⟨procedure declaration⟩
```

5.1. Type Declarations

5.1.1. Syntax

```
⟨type list⟩ ::= ⟨simple variable⟩|
    ⟨simple variable⟩ , ⟨type list⟩
⟨type⟩ ::= real | integer | Boolean
⟨local or own type⟩ ::= ⟨type⟩|own ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩
```

5.1.2. Examples

```
integer p,q,s
own Boolean Acryl,n
```

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. Array Declarations

5.2.1. Syntax

```
⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩|⟨bound pair list⟩,⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩]|
    ⟨array identifier⟩,⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩|⟨array list⟩,⟨array segment⟩
⟨array declaration⟩ ::= array ⟨array list⟩|⟨local or own type⟩
    array ⟨array list⟩
```

5.2.2. Examples

```
array a, b, c[7:n,2:m], s[-2:10]
own integer array A[if c<0 then 2 else 1:20]
real array q[-7:-1]
```

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch Declarations

5.3.1. Syntax

```
⟨switch list⟩ ::= ⟨designational expression⟩|
    ⟨switch list⟩,⟨designational expression⟩
⟨switch declaration⟩ ::= switch ⟨switch identifier⟩:= ⟨switch list⟩
```

5.3.2. Examples

```
switch S := S1,S2,Q[m], if v>−5 then S3 else S4
switch Q :=p1,w
```

5.3.3. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ... , obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4. Procedure Declarations

5.4.1. Syntax

```
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩|
        ⟨formal parameter list⟩⟨parameter delimiter⟩
        ⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩ ::= ⟨identifier⟩|⟨identifier list⟩,⟨identifier⟩
⟨value part⟩ ::= value⟨identifier list⟩ ; |⟨empty⟩
⟨specifier⟩ ::= string|⟨type⟩|array|⟨type⟩array|label|switch|
    procedure|⟨type⟩procedure
⟨specification part⟩ ::= ⟨empty⟩|⟨specifier⟩⟨identifier list⟩ ; |
    ⟨specification part⟩⟨specifier⟩⟨identifier list⟩ ;
⟨procedure heading⟩ ::= ⟨procedure identifier⟩
    ⟨formal parameter part⟩ ; ⟨value part⟩⟨specification part⟩
⟨procedure body⟩ ::= ⟨statement⟩|⟨code⟩
⟨procedure declaration⟩ ::=
    procedure ⟨procedure heading⟩⟨procedure body⟩|
    ⟨type⟩ procedure ⟨procedure heading⟩⟨procedure body⟩
```

5.4.2. Examples (see also the examples at the end of the report)

```
procedure Spur(a)Order:(n)Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s+a[k,k]
end

procedure Transpose(a)Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
      for k := 1+i step 1 until n do
      begin w := a[i,k] ;
            a[i,k] := a[k,i] ;
            a[k,i] := w
      end
end Transpose

integer procedure Step (u) ; real u ;
Step := if 0≦u∧u≦1 then 1 else 0

procedure Absmax(a)size:(n,m)Result:(y)Subscripts:(i,k);
comment The absolute greatest element of the matrix a,
    of size n by m is transferred to y, and the subscripts of this
    element to i and k ;
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q])>y then begin y := abs(a[p,q]) ; i := p ;
  k := q
end end Absmax

procedure Innerproduct(a,b)Order:(k,p)Result:(y) ; value k ;
integer k,p ; real y,a,b ;
begin real s ;
s := 0 ;
for p := 1 step 1 until k do s := s+a×b ;
y := s
end Innerproduct
```

5.4.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language

Examples of Procedure Declarations:

EXAMPLE 1.

```
procedure euler (fct, sum, eps, tim)  ;  value eps, tim  ;
integer tim  ;  real procedure fct  ;  real sum, eps  ;
comment euler computes the sum of fct(i) for i from zero up to
infinity by means of a suitabley refined euler transformation. The
summation is stopped as soon as tim times in succession the abso-
lute value of the terms of the transformed series are found to be
less than eps. Hence, one should provide a function fct with one
integer argument, an upper bound eps, and an integer tim. The
output is the sum sum. euler is particularly efficient in the case
of a slowly convergent or divergent alternating series  ;
begin integer i, k, n, t  ;  array m[0:15]  ;  real mn, mp, ds  ;
i := n := t := 0  ;  m[0] := fct(0)  ;  sum := m[0]/2  ;
nextterm: i := i+1  ;  mn := fct(i)  ;
          for k := 0 step 1 until n do
              begin mp := (mn+m[k])/2  ;  m[k] := mn  ;
                mn := mp end means  ;
          if (abs(mn)<abs(m[n]))∧(n<15) then
              begin ds := mn/2  ;  n := n+1  ;  m[n] :=
                mn end accept
          else ds := mn  ;
          sum := sum + ds  ;
          if abs(ds)<eps then t := t+1 else t := 0  ;
          if t<tim then go to nextterm
end euler
```

EXAMPLE 2.[8]

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi)  ;  value x,y  ;
integer n  ;  Boolean fi  ;  real x,eps,eta,xE  ;  array
y,yE  ;  procedure FKT  ;
comment: RK integrates the system y_k'=f_k(x,y_1,y_2,...,y_n)
(k=1,2,...,n) of differential equations with the method of Runge-
Kutta with automatic search for appropriate length of integration
step. Parameters are: The initial values x and y[k] for x and the un-
known functions y_k(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e.
the set of functions f_k. The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre
sents the solution at x=xE. The Boolean variable fi, which must
always be given the value true for an isolated or first entry into
RK. If however the functions y must be available at several mesh-
points x_0, x_1, ..., x_n, then the procedure must be called repeat-
edly (with x=x_k, xE=x_{k+1}, for k=0, 1, ..., n-1) and then the
later calls may occur with fi=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=f_k(x,y[1], y[2], ..., y[n])
for x and the actual y's. A procedure comp enters as a nonlocal
identifier  ;
b·gin
  array z,y1,y2,y3[1:n]  ;  real x1,x2,x3,H  ;  Boolean out  ;
  integer k,j  ;  own real s,Hs  ;
  procedure  RK1ST(x,y,h,xe,ye)  ;  real  x,h,xe  ;  array
          y,ye  ;
    comment: RK1ST integrates one single RUNGE-KUTTA
          with initial values x,y[k] which yields the output
          parameters xe=x+h and ye[k], the latter being the
          solution at xe. Important: the parameters n, FKT, z
          enter RK1ST as nonlocal entities  ;
    begin
      array w[1:n], a[1:5]  ;  integer k,j  ;
      a[1] := a[2] := a[5] := h/2  ;  a[3] := a[4] := h  ;
      xe := x  ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k]  ;
      for j := 1 step 1 until 4 do
      begin
        FKT(xe,w,n,z)  ;
        xe := x+a[j]  ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k]+a[j]×z[k]  ;
          ye[k] := ye[k] + a[j+1]×z[k]/3
```

[8] This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step-by-step integration of differential equations in an automatic computing machine, [*Proc. Camb. Phil. Soc.* *47* (1951), 96]; and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, [*Fysiograf. Sällsk. Lund, Förhd.* *20*, 11 (1950), 136–152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

```
        end k
      end ;
    end RK1ST   ;
Begin of program:
      if fi then begin H := xE−x   ;   s := 0 end else H := Hs   ;
      out := false   ;
AA: if (x+2.01×H−xE>0)≡(H>0) then
      begin Hs := H   ;   out := true   ;   H := (xE−x)/2
        end if   ;
      RK1ST (x,y,2×H,x1,y1)   ;
BB: RK1ST (x,y,H,x2,y2)   ;   RK1ST(x2,y2,H,x3,y3)   ;
      for k := 1 step 1 until n do
            if comp(y1[k],y3[k],eta)>eps then go to CC   ;
      comment: comp(a,bc,) is a function designator, the value
        of which is the absolute value of the difference of the
        mantissae of a and b, after the exponents of these quan-
        tities have been made equal to the largest of the exponents
        of the originally given parameters a,b,c   ;
      x := x3   ;   if out then go to DD   ;
      for k := 1 step 1 until n do y[k] := y3[k]   ;
      if s=5 then begin s := 0   ;   H := 2×H end if   ;
      s := s+1   ;   go to AA   ;
CC: H := 0.5×H   ;   out := false   ;   x1 := x2   ;
      for k := 1 step 1 until n do y1[k] := y2[k]   ;
      go to BB   ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words [in typewritten copy; boldface in printed copy—Ed.] have been collected at the beginning.

The examples have been ignored in compiling the index.

END OF THE REPORT