Informal Introduction to ALGOL 68

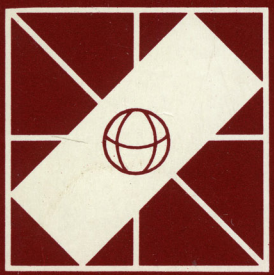
C. H. Lindsey, S. G. van Der Meulen

lindsey
v.d. meulen

informal introduction to ALGOL 68

informal introduction to ALGOL 68

by
c.h. lindsey and
s.g. van der meulen



IFIP

REVISED

north-holland

INFORMAL INTRODUCTION TO ALGOL 68
REVISED EDITION

INFORMAL INTRODUCTION to ALGOL 68

C. H. LINDSEY
Department of Computer Science, University of Manchester

S. G. van der MEULEN
Department of Informatics, University of Utrecht

Revised Edition

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM · NEW YORK · OXFORD

North-Holland ISBN 0 7204 0726 5

First edition 1971
Revised reprint 1973
Second completely revised edition 1977
Revised reprint 1980, second printing 1981

Published by:
NORTH-HOLLAND PUBLISHING COMPANY,
AMSTERDAM, NEW YORK, OXFORD.

Sole distributors for the U.S.A. and Canada:

Elsevier/North-Holland Inc.
52, Vanderbilt Avenue
New York, NY 10017

Library of Congress Cataloging in Publication Data

Lindsey, C.H.
Informal introduction to ALGOL 68.

Authors' names in reverse order in previous ed.
Includes index.
1. ALGOL (Computer program language)
2. Electronic digital computers--Programming.
I. Meulen, S.G. van der, joint author. II. Title.
QA76.73.A24M47 1977 001.6'424 78-148532
ISBN 0-7204-0504-1
ISBN 0-7204-0726-5 pbk.

PRINTED IN THE NETHERLANDS

To the uninitiated reader

ACKNOWLEDGEMENTS

The Authors wish to thank Prof. Dr. A. van Wijngaarden and the other authors of the ALGOL 68 Report, and also the many members of WG 2.1 and others in the computing community who read this Introduction in draft form and in its first edition, for their encouragement and helpful criticisms of the text. The Mathematical Centre, Amsterdam are particularly thanked for their assistance in reproducing the drafts of the first edition. The authors also wish to acknowledge the official support accorded to this work by IFIP TC–2 and WG–2.1.

PREFACE

Publication of this volume represents a major step in making accessible to the international computing community the operational content of the "Revised Report on the Algorithmic Language ALGOL 68". The Report itself is, of course, an unparalleled accomplishment in the literature of programming languages *as a defining document.* The complexity of the topic, however, necessarily disqualifies such a completely rigorous treatment from consideration as a general pedagogical device; hence the present volume.

As noted by the authors, this book "is not – and is not intended to be – a primer for the programming novice". This is as it should be. The Revised Report addresses those who must understand every nuance of the language; primarily language designers and compiler implementers. If implementations of ALGOL 68 are to be other than academic exercises, there is a pressing need to acquaint the set of people who write computer programs as a routine part of their daily lives with the essential elements of the language. This book can do just that.

The algorithmic language, ALGOL 68, stands as a major product of the International Federation for Information Processing, an organization now in its second decade, counting among its members the national information processing societies from thirty-four countries spread over all six of the world's inhabited continents. Among the principal activities of the Federation is the work of Technical Committee 2, Programming Languages, and its Working Group 2.1, ALGOL. The initial effort of WG 2.1 stemmed from the development of the algorithmic language ALGOL 60, and since 1964 under the Chairmanships of Prof. Dr. W.L. van der Poel, Prof. Dr. M. Paul and currently Prof. J.E.L. Peck, this group of international experts, including the authors of the present work, has been engaged in the design and development of ALGOL 68.

While the present book is wholly the work of the two authors, it has been extensively reviewed in manuscript by the Working Group and has the status of a working paper within the group. It follows that the accuracy with which it represents ALGOL 68 is far higher than might be apparent from its cover. Every effort has been made to ensure that the book contains a comprehensive, accurate and readable introduction to the language. Having had the opportunity to observe this effort closely from my position as Chairman of TC-2, I can assure the reader that the authors have been successful in this endeavor.

T.B. Steel, Jr.
New York, May 20, 1976

FOREWORD

The algorithmic language ALGOL 68 was designed by Working Group 2.1 of the International Federation for Information Processing, and formally defined in a Report* published early in 1969. The first edition of the present text was a companion volume to that Report.

Since that time the language, in whole or in part, has been implemented on a variety of computers and substantial experience has been gained of its use. One leading computer manufacturer has released an implementation in virtual complete agreement with the current official definition, and it should only be a matter of time before others follow.

The experience of implementation and use led to many proposals for changes to the language and these were incorporated in the text of a Revised Report approved at the Los Angeles meeting of the Working Group in 1973. The "Revised Report on the Algorithmic Language ALGOL 68"† (hereafter referred to as simply "the Report") is now the official, rigorous and final definition of the machine-independent programming language ALGOL 68.

This "Informal Introduction to ALGOL 68" seeks to describe, rather than to define, the revised language. If you have some difficulty in understanding the Report, it is our hope that you will find our informal treatment more palatable, even though this may have been achieved at the expense of rigour. It is the companion volume referred to in Section R.0.1.1 of the Report. (We shall always precede our references to the Report with such an R. All other references are to the present text.)

This introduction, however, is not – and is not intended to be – a primer for the programming novice. The "user" to whom we address ourselves is assumed to be a "programmer", i.e. someone who is able to write, or at least to read, a text in some machine-independent programming language which is on a level not too much below, for instance, ALGOL 60. Our aim has been to describe the whole of ALGOL 68, and this Introduction may also therefore have some merit as a work of reference – provided it is always understood that the official Report is the final arbiter in all cases of doubt.

* A. van Wijngaarden (ed.), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Report on the algorithmic language ALGOL 68, *Numer. Math.* **14** (1969) 79-218; also in *Kibernetika* **6** (1969) and **7** (1970).

† A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker, Revised report on the algorithmic language ALGOL 68, *Acta Informat.* **5** (1975) parts 1-3 (reprints published by Springer, Berlin, and also as Mathematical Centre Tract 50 by the Mathematisch Centrum, Amsterdam); also in SIGPLAN Notices **12** (5) (1977).

TABLE OF CONTENTS

0. VERY INFORMAL INTRODUCTION TO ALGOL 68

Contents

0.0. Aims and methods

Since ALGOL 68 is a highly recursively structured language, it is quite impossible to describe it until it has been described. So that you can read this Introduction without tying your own mental processes into a recursive knot, it has been laid out to a certain pattern, which we ask you to follow. Please, therefore, start by reading once or twice "Very Informal Introduction", in which we try to give a broad survey of what is in this language – mainly by the way of small examples and plain explanations. After that, we shall tell you what to do next.

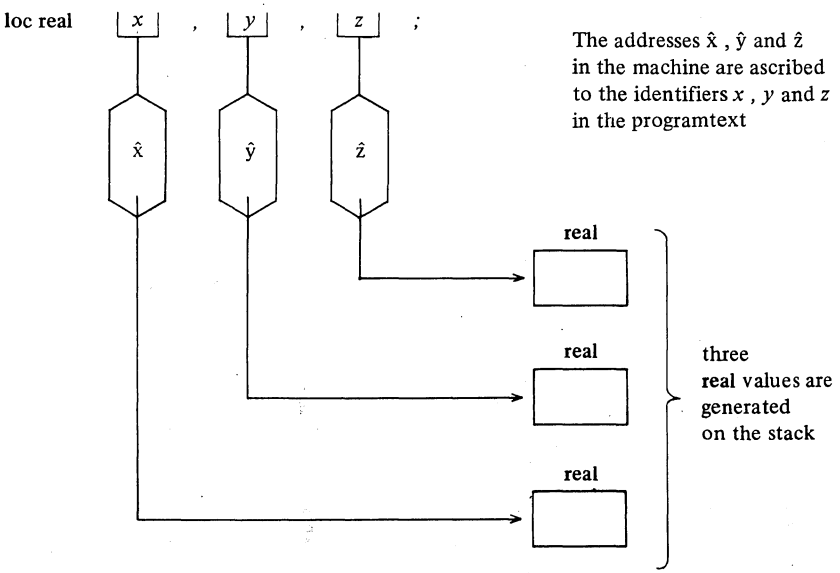
if *you think you know it all already*
then *read (what to do next)* **comment** *Section 0.14* **comment**
fi

0.1. A simple program

(E1)
```
begin   loc real x , y , z ;
        read (x) ; read (y) ;
        z := ( x + y ) / 2 – sqrt ( x × y ) ;
        print (z)
end
```

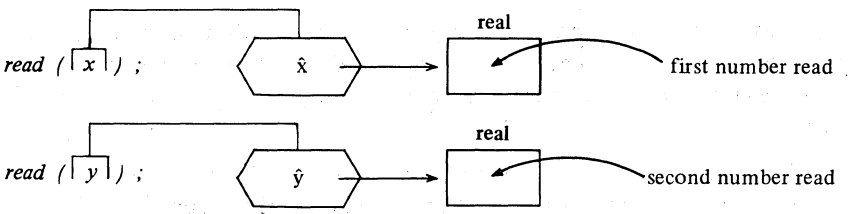
This piece of text represents a program, and as such it defines a sequence of actions to be performed by a computer. This sequence of actions is termed "the elaboration of the program". We shall briefly outline the elaboration of E1 :

1. Three identifiers *x* , *y* and *z* for **real** variables are declared. That is to say that somewhere in the memory of our computer, in the "stack", three locations for **real** values are reserved. These values are referred to by the names x̂ , ŷ and ẑ (i.e. by their "addresses") which are also entities (values) in the computer. It thus appears that a 'variable' consists of a value associated with the name which refers to it. You may consider the identifiers *x* , *y* and *z* as the representatives in the programtext of the names x̂ , ŷ and ẑ.

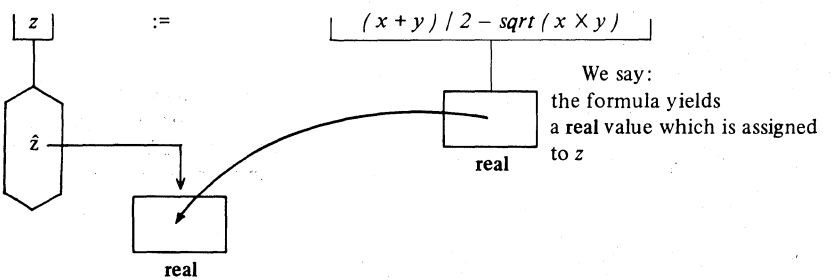


The names x̂ , ŷ and ẑ "refer to" **real** values (see also Section 0.8).

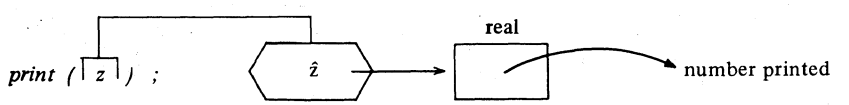
2. Next, from a punched tape, or punched card or some other medium, two numbers are read in and assigned to x and y. That is to say, these numbers are converted into the proper bit-patterns in the private internal number-system of the computer, which subsequently are stored in the locations corresponding to the names $\hat{x}$ and $\hat{y}$:



3. Then, the difference between the arithmetic and geometric mean of these values is calculated and assigned to z, so that we find this difference in the location corresponding to $\hat{z}$:



4. Finally, this value, referred to by $\hat{z}$, is reconverted into humanly recognizable graphics and is printed by some device:



Summing up we have:

1. Three variable-declarations.
2. Two input statements.

3. An assignation. To the left of the becomes-symbol := we find the destination *z*, and to the right a 'formula' whose value is to be assigned to that destination.

4. An output statement.

The three variable-declarations in the first line are separated by and-also-symbols, represented by commas. This means that they are elaborated "collaterally", which is a technical term stating that the order of their elaboration is not prescribed.

The collateral declaration in the first line, the input statements in the second, the assignation in the third, and the output statement in the last line are separated by go-on-symbols (represented by semicolons). This means that they are elaborated "serially". This again is a technical term stating that the order in which these phrases are elaborated is explicitly prescribed to be the textual order: one after the other. They form a 'serial-clause'.

The piece of text E1 might very well be part of a larger program. The meaning of the identifiers *x* , *y* and *z* (and consequently the existence of the names x̂ , ŷ and ẑ and of the **real** values referred to by them) is, however, local: i.e. the "reach" of their declarations is limited to the serial-clause between **begin** and **end**, which delimit a 'closed-clause'. If outside this closed-clause (or inside another closed-clause contained within it) other identifiers *x, y* and *z* should be declared, then these have nothing to do with those declared in example E1. We say, therefore, that the original meaning of *x* , *y* and *z* applies only to that part of the program text.

The standard input procedure *read* accepts as actual parameter not only the name of a **real** value, but also (amongst many others) a 'row-display' like *(x,y)*.

Consequently, line 2 may be replaced by:

read ((x,y)) ; Observe that we again give precisely one actual-parameter to *read*; instead of one name, one row-display of names.

Instead of the two phrases 3 and 4 we might write, in one statement:

print (z := (x + y)/2 – sqrt (x × y))

Instead of the **begin** and **end** we may write *(* and *)*. Thus E1* below is, at least in its effect, completely equivalent to E1:

(E1*)
```
(  loc real x , y , z ; read ( (x,y) ) ;
   print (z := (x + y)/2 – sqrt (x × y) )
)
```

And now we discover that *z* became superfluous, so that we can write:

(E1**)
```
(  loc real x , y ; read ( (x,y) ) ;
   print ( (x + y)/2 – sqrt (x × y) )
)
```

0.2. The primitive modes, denotations

The **real** in E1 specifies a mode (i.e. it specifies that x̂ , ŷ and ẑ will refer to values which belong to a certain class). An infinite number of distinct modes (disjoint classes of values) is provided in this language. They are, however, derived from the primitive modes (which form the basis of the entire mode system): boolean (**bool**), integral (**int**), real (**real**), character (**char**) and a few others (see 0.11).

For these primitive modes we have 'denotations': symbols or sequences of symbols yielding a specific value in such a mode.

0.2.1. bool

There are two boolean values, denoted by the symbols **true** and **false**.

If (in E2 below) *C* is a piece of program yielding a boolean value, then this value is either **true** or **false**. In a 'conditional-clause' like:

(E2) **if** *C* **then** *Ctrue* **else** *Cfalse* **fi**

first of all *C* is elaborated, being the condition between **if** and **then**. The then-part *Ctrue* between **then** and **else** is elaborated only if the condition yielded **true**. The else-part *Cfalse* between **else** and **fi** is elaborated only if the condition yielded **false**.

The symbols **if** and **then**, **then** and **else**, **else** and **fi** enclose clauses in which new identifiers of local reach may be declared as in a closed-clause (they form pairs of brackets, so to speak). Observe that the whole conditional-clause is enclosed between **if** and **fi**. The else-part may be absent, in which case the then-part is closed by **fi**; the then-clause must always be there.

For E2 we may also write:

(E2*) *(C | Ctrue | Cfalse)*

which may be a fine notation to use in a formula, where either the value of *Ctrue* or the value of *Cfalse* is to be yielded (see 0.5.2).

0.2.2. **int**

There will be many integral values, denoted by:

0 1 2 3 4 5 6 7 8 9 10 11 12 - - - - - - 2147483647 - - -

How far can we go? The Report does not answer this question. It depends entirely on the implementation. It is, however, prescribed that we can always know the largest integral value by an "environment enquiry" *max int*, which is a standard identifier yielding the largest integral value in a specific implementation (see also 0.11.2).

Observe that there is no sign preceding an integral-denotation. Of course you may write:

+1 −37 −1000 +534711 −513617 etc.

but then the + and the − are monadic-operators applied to the value denoted.

Let (in E3 below) *I* be a piece of program yielding an integral value and let *S* be another piece of program; let *I1, I2, I3, - - - -, Ik* be certain phrases. In a 'case-clause' like:

(E3) **case** *I* **in** *I1, I2, I3, - - - -, Ik*
out *S*
esac

first of all *I* is elaborated. If its yield is less than *1* (i.e. *0* or *−1* or *−2* etc) or greater than *k* (i.e. *k + 1* etc), then the out-part *S* will be elaborated; if, to the contrary, the yield of *I* is one of the values *1* or *2* or *3* or - - - - or *k*, then the corresponding phrase in the in-part will be elaborated.

The symbols **case** and **in** and also **out** and **esac** enclose clauses in which new identifiers of local reach may be declared as in a closed-clause or conditional-clause. The whole case-clause is enclosed between **case** and **esac**. The out-part (in E3 **out** *S*) may be absent, in which case the in-part is closed by **esac**; the in-part must always be there containing at least two phrases.

For E3 we may also write:

(E3*) *(I | I1, I2, I3, - - - -, Ik | S)*

0.2.3. **real**

There are many real values, which can be denoted in many styles:

3.1415927 or $31415927_{10}-7$ or $0.31415927_{10}+1$ or $.31415927_{10}1$

instead of the symbol $_{10}$ you may also use *e* : *31415927e–7*

0.9 .9 9.0 100.0 0.0 $1_{10}2$ $10_{10}10$ $1_{10}-10$ $1_{10}0$ --

The class of real values in the finite memory of a concrete computer is finite by necessity; it is an implementation dependent image of the mathematical concept "real number system". The largest real value in a certain implementation can be obtained by the environmental enquiry *max real* and the smallest real value which can be usefully compared with *1.0* from *small real.* (See also 0.11).

Observe again that there is no sign preceding a real-denotation. Of course you may write:

+1.0 $-37_{10}-4$ $-31415927_{10}-7$ etc.

but then the + and – preceding the real-denotation are monadic-operators applied to the value denoted.

0.2.4. char

There is a prescribed minimal set of graphics in which we find all (small) letters, the digits and some other tokens. The class of character values is at least this minimal set; specific implementations, however, may extend it.

A character-denotation consists of the character denoted between two quote-symbols:

"a" "b" "c" --- --- --- "x" "y" "z"
"0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
"." "$_{10}$*" "," "(" ")" "+" "–" --- ---*

Specific character-denotations are:

" " or *"_"*	representing the space-symbol	(see 0.4.3)
""""	representing the quote-symbol itself	(see 5.1.1.1)

0.2.5. bool, int, real and char

(D1)

```
loc bool p, q;                    bool p, q;
loc int  i, j, k, m, n;    or     int  i, j, k, m, n;
loc real a, b, x, y;              real a, b, x, y;
loc char c;                       char c;
```

In these declarations boolean variables are "ascribed" to the identifiers *p* and *q*; correspondingly integral variables are ascribed to *i*, *j*, *k*, *m* and *n*,

real variables to *a* , *b* , *x* and *y* and a character variable to *c*. That is to say these identifiers are made to yield the names p̂ , q̂ , î , j , k̂ , m̂ , n̂ , â , b̂ , x̂ , ŷ , and ĉ referring to boolean- , integral- , real- or character-values respectively. **loc bool** , **loc int** , **loc real** and **loc char** are "local generators" and signify that the lifetime of the values generated is restricted. However, since **loc** is the default situation, this symbol may be omitted here thus preserving some similarity to ALGOL60.

In this Informal Introduction, identifiers will occasionally occur out of context from their declarations. Unless otherwise specified, these identifiers will be assumed to have been declared as listed in D1 (or D2, D3, - - hereafter). The complete set is listed in Appendix 2.

For boolean values operators are defined: ∨ or **or**, ∧ or **and**, ¬ or **not**, yielding boolean values. Boolean values can be compared =, ≠ or /=; the result of a comparison is also a boolean value. The monadic-operator **abs**, however, when applied to a boolean value, yields an **int** : **abs** *p* is *1* if *p* is **true** , **abs** *p* is *0* if *p* is **false**. This can be expressed concisely as the conditional-clause *(p | 1 | 0)*.

Many operators are defined for integral and real values: + , − , × or * , / , ÷ or % or **over** (implying integral division), **mod** or ÷× (for the modulo operation), ↑ or ** (for raising to the power) , etc. The result is an integral value when both operands are integral (except division which always yields a real value); in all other cases the result is real.

Integral and real values can be compared: < , ≤ or <= , = , ≠ or /= , ≥ or >= , > ; the result is a boolean value.

The monadic operator **abs** , when applied to an **int** or a **real** *ir* , yields *(ir < 0 | −ir | ir)*.

The monadic operators **round** and **entier** (integral part of) serve to transfer a **real** into an **int**. The transfer of an **int** into a **real** is implicit in the language (no operator is needed to control this transfer); you may write:

x := *i*

but you must write:

i := **round** *x* or *i* := **entier** *x*

Each character value corresponds to an integral value; no two different characters correspond to the same **int**; the actual correspondence is to be defined by the implementation. The **int** corresponding to a **char** is obtained by applying the monadic operator **abs** (**abs** *c* yields the integral value corresponding to *c*), and the converse operation by **repr**.

Character values can be compared as if they were integral values and by the same operators; in fact their **abs** values are then compared.

Character values are the materials from which **strings** are composed (see Section 0.4.3).

0.3. Loops

Suppose we want to input many pairs of numbers x and y and we want to do the algorithm E1 that many times. Let the input start with an integral number greater than zero, which fixes the number of pairs following. Then–in a very old-fashioned way–the program might be:

```
(E4)    begin   loc int n ; read (n) ; loc int count := n ;
                loc real x , y ;
        again : read ( (x,y) ) ;
                print ( (x+y)/2 – sqrt (x × y) ) ;
                count := count – 1 ;
                if count > 0 then goto again fi
        end
```

1. The reason we declared *n* was to hold the number of pairs; it is quite natural to read this *n* immediately after its declaration. In this language it is allowable to put statements between declarations. The counting will be done via the identifier *count*; the counter is initialized at its very declaration.

3. The identifier *again* defines a 'label' signposting a point in the program where we want to **go to** from elsewhere. Labels are only allowed beyond the declarations (i.e. it is not allowed to write a label in a serial-clause where a declaration follows).

5. The counter is decreased by 1. Operations of this kind occur so often in the practice of programming that we have got special operators for them; they combine subtraction (or addition, multiplication, division etc.) with assignation. We thus may write:

count **minusab** *1* or *count* –:= *1*

6. We may combine the phrases 5 and 6 into one, also omitting the redundant go-to-symbol **goto**:

if *(count* –:= *1)* > *0* **then** *again* **fi**

which can be abbreviated into:

((count –:= *1)* > *0* | *again)*

The use of labels and **gotos** leads in many cases to badly structured programs and should be avoided wherever it is possible. In this language we have alternative constructs to structure the program in such cases, avoiding the **goto** entirely.

In particular, a cycle like E4 can be put in a more concise form in which the counting will be done behind the screens. The result is a much safer and also more transparent program:

```
(E4*)   begin  loc int n ; read (n) ;
               to n
               do loc real x , y ; read ( (x, y) ) ;
                  print ( (x+y)/2 – sqrt (x × y) )
               od
        end
```

1. Where the counting is done behind the screens we do not have to declare a *count*.
2. The serial-clause following, between **do** and **od**, will be repeated *n* times.
3, 4, 5. Between **do** and **od** we find a serial-clause, establishing a reach in which, consequently, variables of local scope can be generated (here ascribed to *x* and *y*). This construct **to** *n* **do** - - - - **od** is a particular case of a much more general construction (see 0.6).

0.4. The creation of new modes, multiples

One of the interesting features of this language is the possibility of deriving new modes from the primitive ones, as many as you need. The method whereby new modes are created is such that they can, in their turn, be used to create further modes in a perfectly systematic manner. Some of these derived modes, such as **string** and **compl** (complex) are standard in the language (i.e. being declared in the standard-prelude, they are permanently built in).

In this section, we shall briefly outline the construction of multiple values (multiples). Other constructions (procedures, structures, references, unions and further derived modes) will be outlined in following sections. You will find a more systematic treatment in Chapters 1 and 2.

0.4.1. Multiples

Suppose you want to use a row of *n* **reals** named *u* , then you may declare, for instance:

```
(E5)   loc [1 : n] real u ;   The lower-bound of the row is 1 ,
                              the upper-bound of the row is the value of n.
```

Now you have at your disposal *n* **real** values on the stack:

$u[1], u[2], \text{---}, u[n]$

In fact you have got more than this, you have got *n* "subnames":

$\hat{u}[1], \hat{u}[2], \text{---}, \hat{u}[n]$

to which you can assign new values as in the case of simple variables:

$u[i] := u[i] + x$
$u[i] := u[n-i]/u[j]$ etc.

If you want to use a square matrix of *n* x *n* **reals** (a row-row-of-real) named *a* , then you may declare, for instance:

(E6) **loc** [*1*:*n*,*1*:*n*] **real** *a* ;

Now you have at your disposal *n* x *n* **real** values on the stack:

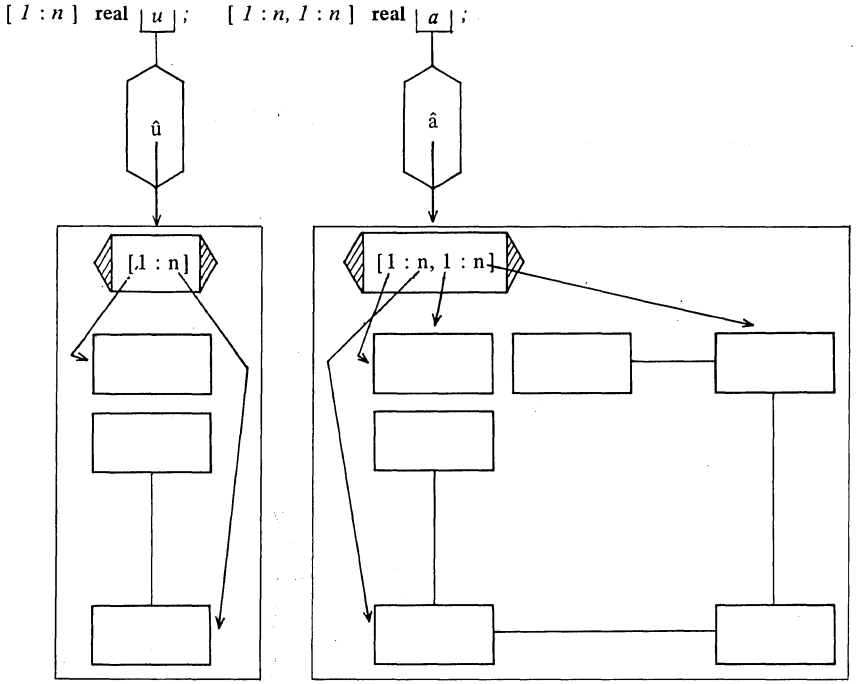
$a[1,1], a[1,2], \text{---}, a[1,n], a[2,1], a[2,2], \text{---}, \text{------}, a[n,n]$

In fact you have got *n* x *n* subnames to which you can assign new **real** values:

$a[i,j] := a[i,k] \times a[k,j]$
$a[i,j] := a[j,i] / (i+j)$ etc.

(D2) [*1*:*n*] **real** *x1* , *y1* ; [*1*:*n*] **int** *i1* ; Observe how the optional **loc** has been omitted as we shall often do in the sequel
[*1*:*m*,*1*:*n*] **real** *x2* ; [*1*:*m*,*1*:*n*] **int** *i2* ;
[*1*:*n*,*1*:*n*] **real** *y2* ;

By E4 and E5 we declared the identifiers *u* and *a* to yield the names û and â respectively (see also 0.1). The name û refers to a [ ] **real**, a row-of-real, the name â refers to a [,] **real**, a row-row-of-real. That is to say that *u* yields a [ ] **real** variable and *a* yields a [,] **real** variable in much the same way as, for instance, *x* yields a **real** variable and *p* a **bool** variable.



Now the question arises as to whether we may assign, for instance:

u := x1
a := y2

the answer is yes, and it does exactly what you should expect it to do:

$$u[1] := x1[1], u[2] := x1[2], \text{---}, u[n] := x1[n]$$

and:

$$a[1,1] := y2[1,1], a[1,2] := y2[1,2], \text{---}, \text{--- ---}, a[n,n] := y2[n,n]$$

provided, of course, that the bounds to the left and to the right of the becomes-symbol are equal. Also an input-statement like:

read (y2)

does what you would expect:

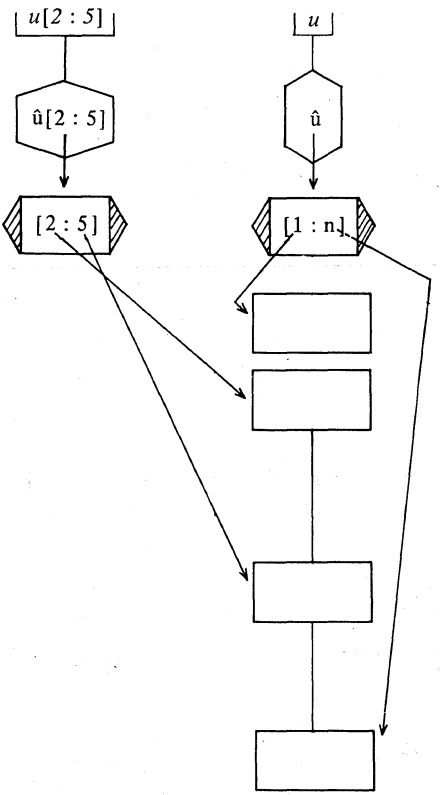
$$read\ (\ (y2[1,1], y2[1,2], \text{---}, y2[1,n], \text{---}, \text{--- ---}, y2[n,n]\ )\ )$$

Where û is a name referring to the whole of the multiple value:

$u[1], u[2], \text{- - -}, u[n]$

the 'slice' $u[2{:}5]$ yields a subname referring to a part (a slice) of that multiple value, namely:

$(\, u[2], u[3], u[4], u[5] \,)$



In much the same way $a[i,\ ]$ yields a subname referring to the multiple value:

$(\, a[i,1], a[i,2], \text{- - -}, a[i,n] \,)$

Therefore, even assignations like:

```
u[2:5] := x1[n-3:n];
a[i, ] := u ;
a[i, ] := y2[ ,j]
```

etc. do exactly what they suggest. For further discussion see 1.5.1 and 5.5.1.3.

Moreover, operators acting upon multiple values and slices may also be defined (see, for example, 8.5), so that we can then write clauses like:

```
u := x1 + y1 ;
a := y2 × a
```

0.4.2. New mode indications

Once you have decided to create a new mode, you may want to give this new class of values a distinguishing mark (we do not say "name", because that is a technical term in this language with a very specific meaning, see 0.1 and 0.8). You may define a new 'mode-indication' by declaring:

(E7) **mode vector** = [*1*:*n*] **real** ;
mode matrix = [*1*:*n*,*1*:*n*] **real** ;

And now, in the context of these mode-declarations:

(E5*) **loc vector** *u* ; is equivalent to E5 : **loc** [*1*:*n*] **real** *u* ;

(E6*) **loc matrix** *a* ; is equivalent to E6 : **loc** [*1*:*n*,*1*:*n*] **real** *a* ;

0.4.3. Multiples with flexible bounds, **strings**

In E5 and E6 the bounds of the variables (the lower-bounds *1*, and the upper-bounds *n*) are fixed at the elaboration of the declaration and cannot be changed afterwards, but on some occasions you might wish to do just that, although this may be expensive in some implementations (the storage allocation at run time is more complicated than in the case of fixed bounds).

Variables whose bounds may be changed after the declaration of the multiple are termed "flexible" (**flex**):

(E8) **loc flex** [*m* : *n*] **real** *fu* ; the bounds of *fu* are initially [*m* : *n*] but may be changed later.

Flexible bounds are particularly useful in the case of **strings**, which are built into the standard-declarations of the language (which is why one may expect a reasonably efficient implementation of, in particular, this flexible bound application). A **string** is a row-of-character with flexible bounds:

(D3) **mode string** = **flex** [*1*: *0*] **char** ;
loc string *s* ;

The variable-declaration **loc string** s (which is equivalent to **loc flex** [*1:0*] **char** *s*) declares *s* to yield a flexible name ŝ referring to a row-of-character, which is empty to begin with. As soon as you assign to *s*, the bounds get the new values required:

(E9) *s* := *"the upper-bound becomes 26";*
s := *"the flexible upper-bound now becomes 39" ;*
s := " " the **string** is empty again

By the way, you have just met three string-denotations, the lower-bound of a string-denotation is always 1. Observe that the denotation for the empty string is "", and for the space is " " (or may also be " _ ").

For **string**, being a built-in mode, several operators are defined, such as $<$, $\leqslant$, = , ≠ , $\geqslant$, $>$ (to compare them) and + and x (to concatenate them).

If the value of *s* is *"this is the begin"*, then the outcome of the assignation:

(E10) *s* := *s* + *" and this is the end."*

or (which is the same):

(E10*) *s* +:= *" and this is the end."*

may speak for itself.

0.5. The value of a unit

So far we have met identifiers and denotations as the objects in this language that yield a value of some mode. They are, however, only specific cases of a 'unit'. In this section we shall show you over some units and pay some attention to the values they may yield. The complete and systematic treatment will be found in Chapter 5.

0.5.1. The value of a formula

A formula is a unit. For example:

x + y , (x + y) / 2 , (x + y)/2 − sqrt (x × y)

A formula defines a (more or less compound) computation, which usually yields (i.e. delivers on the stack) a value of some mode. We then say: "the formula yields (upon elaboration) the value".

In the example above, we met:

identifiers *x* and *y*, units yielding variables,

a denotation *2*, a unit yielding a constant;

and another kind of unit *sqrt (x × y)*, which is the 'call' of a procedure returning a value: the square root of the value of the formula $x \times y$ (for procedure calls see Section 0.7).

Different operators may have different priorities. The implied bracketing in the example above is:

```
( x + y ) / 2  -  sqrt ( x × y )
----------       ------------
left-operand     right-operand
```

The 'left-operand' and the 'right-operand' of the subtraction are elaborated collaterally. That is to say that there is no prescribed order for getting the value of the left-operand and getting the value of the right-operand (see also 0.1). The same applies again to the elaboration of the left- and the right-operand in the formula $(x + y) / 2$.

Hence, an implementor is perfectly entitled to optimize the elaboration of formulae. For example:

y1 [**round** *sqrt(a × b)*] + *i1* [**round** *sqrt(a × b)*] × *x1* [**round** *sqrt(a × b)*]

The implementor may elaborate **round** *sqrt (a × b)* only once (for reasons explained in 3.7.1), and elaborate the multiplication *i1* [*k*] × *x1* [*k*] before accessing *y1* [*k*]; that is to say that there is no prescribed order of elaboration "from left to right".

0.5.2. The value of a conditional clause

A conditional-clause is a unit:

if *p* **then** $31415927_{10}-7$ **else** $27182818_{10}-7$ **fi**

yields upon elaboration the **real** value of its then-part or of its else-part, depending upon the value of its condition. You may assign it to a **real** variable, or use it in a formula:

$x := 1 / (p \mid 31415927_{10}-7 \mid 27182818_{10}-7)$

If a conditional-clause yields a variable then you are perfectly entitled to use it in a destination (to the left of a becomes-symbol):

if *p* **then** *x* **else** *y* **fi** := *3.1415927*

or:

(p | x | y) := *3.1415927*

0.5.3. The value of a serial clause

A serial-clause (see 0.1) may also yield a value, although it is not (yet) a unit. It may be used, however, to make a unit. For example, a closed-clause, which is a unit (see 0.5.4), contains a serial-clause, and a conditional-clause (another unit) may contain several of them.

The value of a serial-clause, then, is the value of its final (completing) unit. The serial-clause:

read (x) ; x ≠ 0

(see E11 below) yields the value of its final unit *x ≠ 0*, which is a **bool** value. Now consider the program:

(E11)
```
begin
int num := 0 , pos := 0 , neg := 0 , real absum := 0 , x ;
while read (x) ; x ≠ 0
  do absum plusab     if num +:= 1 ; x > 0
                    then pos  +:= 1 ; +x
                     else neg  +:= 1 ; –x
                       fi
  od ;
print ( (num, pos, neg, absum ) )
end
```

Here we meet another form of a loop-clause (see 0.3 E4). In a loop of the form:

while *C* **do** *S* **od**

first of all the condition *C* will be elaborated, yielding a **bool** value; as long as this condition yields **true**, the serial-clause *S* between **do** and **od** will be elaborated, followed by a new elaboration of *C*. That is to say: depending on the **bool** lastly yielded by *C*, the elaboration of **while** *C* **do** *S* **od** results in

C or *C ; S ; C* or *C ; S ; C ; S ; C* or *C ; S ; C ; S ; C ; S ; C* etc.

3. Now, in E11 above, the construct after **while** is the serial-clause

read (x) ; x ≠ 0

yielding the **bool** value yielded by its last unit *x ≠ 0*. Hence, if the first number read is *0*, the clause between **do** and **od** will never be elaborated; if, on the contrary, this number is unequal to *0*, then the formula

4–7. *absum* **plusab** **if** *num* +:= *1* ; *x* > *0*
then *pos* +:= *1* ; +*x*
else *neg* +:= *1* ; –*x*
fi

will be elaborated.

The right operand of **plusab** is a conditional-clause, the condition of which is a serial-clause increasing the counter *num* by *1* and yielding the **bool** value *x* > 0. Depending on this condition the then-part *pos* +:= *1* ; +*x* or the else-part *neg* +:= *1* ; –*x* will be elaborated, yielding either +*x* or –*x* after having increased either *pos* or *neg* by *1*. That is to say: the serial-clause between **do** and **od** (here a formula) results in adding **abs** *x* to *absum*.
3. After that the serial-clause after **while** is revisited, repeating the story (looping the clause) as long as the number read is found to be unequal to *0*.

It may happen that a serial-clause does not yield a value, because its final unit does not leave a value on the stack (the root of this will be shown in 0.7). We then say: "this serial-clause yields **void**". For example:

real *x , y , z ; read ((x,y)) ; print (z := (x + y)/2 – sqrt (x × y))*

yields **void**, because the output statement *print* (although it delivers humanly recognizable graphics on some printing device) does not leave a value on the stack. If, however, we want this serial-clause to yield, for example, the name of the value printed (from which that value may then be obtained), then we simply make *z* its final unit:

real *x , y , z ; read ((x,y)) ; print (z := (x + y)/2 – sqrt (x × y)) ; z*

0.5.4. The value of a closed clause

A closed-clause (see 0.1) is a unit and as such it may be used in formulae and in assignations. The value of a closed-clause is that of its constituent serial-clause:

*x2[i, j] := (**real** x , y , z ; read ((x,y)) ;*
print (z := (x + y)/2 – sqrt (x × y)) ;
z)

The value of a closed-clause may also be a name, and then you are entitled to use it in a destination (see also 0.5.2); provided, of course, that the name yielded does not happen to be local to the clause:

*(**real** x , y , z ; read ((x,y)) ;*
print (z := (x + y)/2 – sqrt (x × y)) ;
*if z > 1 **then** m **else** n **fi**)* +:= *1*

Under certain circumstances it may be annoying to have to arrange a closed-clause in such a way as to deliver the value required, because there may be more than one candidate for the final unit. In such cases a 'completer' may help. A completer is the symbol **exit**; the unit preceding a completer is (by definition) a completing (final) unit of the closed-clause. You may take a completer as a suppositious close-symbol.

For example, suppose you want a closed-clause to read a block of *n* pairs of real numbers and to deliver **false** if there is no pair in the block in which the difference between the arithmetic and geometric mean is greater than *1*, but to deliver **true** and to print the first such pair if this is the case. This may be programmed as follows:

```
(E12)  p := ( real x , y , z ;
              to n do read ( (x,y) ) ; z := (x + y)/2 − sqrt (x × y) ;
                   if z > 1 then finish fi
              od ;
              false exit
              finish: print ( (x,y) ) ;
              true )
```

(For another example in which a completer is used, see E28* in 0.7.3).

0.5.5. The value of a constant

To conclude this bird's-eye view, we consider an extremely simple kind of unit, the constant. You may declare:

(E13) **real** *pi* = *3.1415927*

The thus declared identifier *pi* yields a **real** value (and not a **real** variable), in much the same way as the denotation *3.1415927* yields that value. You cannot assign to *pi* (it not being a variable): *pi* := *2.7182818* would be as nonsensical as is *3.1415927* := *2.7182818*. Beware of the slight notational distinction between E12 and:

(E13*) **real** *pivar* := *3.1415927*

This distinction would have been clearer if we had written:

(E13**) **loc real** *pivar* := *3.1415927*

To *pivar* you may assign any other **real** value (*pivar* being a **real** variable). For the notational matter, see 0.8.

Declarations of constants enable the programmer to enforce efficient compilation, for example in accessing the elements of multiple values. Compare E14 and E14* below:

```
(E14)   sw    +:= w[i] ;
        swx   +:= w[i] x x[i] ;
        swx2  +:= w[i] x x[i] x x[i] ;
        swx3  +:= w[i] x x[i] x x[i] x x[i] )
```

In E14 an element such as $w[i]$ and $x[i]$ has to be pulled many times out of a multiple value, which may be rather time consuming.

```
(E14*)  ( real  wi = w[i], xi = x[i] ; real wixi = wi x xi ; real wixi2 = wixi x xi ;
          sw    +:= wi ;
          swx   +:= wixi ;
          swx2  +:= wixi2 ;
          swx3  +:= wixi2 x xi )                    (see also E15)
```

In E14*, an element is never taken out of a multiple more than once. Of course, you could have achieved most of this efficiency equally well by declaring the proper local variables: **real** *wi* := *w*[*i*], *xi* := *x*[*i*]; etc, but then you would still have to go via the names when getting the values referred to (which may take longer in some implementations). In the form of a constant, you have the desired values most readily at hand.

The importance of the declaration of constants, however, is to be found at another level; in Chapter 1, we shall see that in all identity-declarations a constant is declared (see also 0.7 and 0.8), and this mechanism is nothing more nor less than the life-line of the formal – actual correspondence.

0.6. A more involved program

Before embarking upon routines and other new modes that actually make the new language, it is worthwhile to dwell on the subject of primitive declarations and multiple values for just one section more. Many of the (until now only) newly dressed features of the language will be found in full swing in the more involved example E15 below. Although it is the program that matters here, it may acquire a not too artificial setting in the following context:

Suppose the input starts with an integral number n, which fixes the number of pairs of measurements following. The first **real** f of each pair is a factor, accounting for environmental influences on the target measurement, which is the second **real** x of the pair. There may be other (not measured)

influences on x, which is why we are not particularly interested in the correlation of the two. We therefore confine ourselves to the computation of the mean, dispersion and momental skewness of the x's, weighted by the (more or less normal) distribution of the f's. Preceding the (f,x)-couples, but following n, we input another pair of **reals** *eps* and *ups*; all x's below *eps* or above *ups* are to be discarded as being certainly out of range (as a result of punching errors, for instance).

We briefly survey the program E15 (the numbers in the margin refer, as usual, to the linenumbers in the program text):

1- - - . In any place in a program text (except within identifiers and denotations) comments may be inserted. A comment begins and ends with the comment-symbol **comment**. Alternatively **co**, ¢ or # can be used in matched pairs:

¢ *this was the beginning of a comment and this is the end*:¢

3, 9, 24-25 **for** *i* **to** *n* **do** (as well as **for** *i* **to** *n* **while** *C* **do** etc.) are specific cases of the general construction of a loop-clause:

for *i* **from** *start* **by** *step* **to** *finish* **while** *condition* **do** *doclause* **od**

The integral counter *i* is implicitly local to the construction (it has nothing to do with a possibly declared other *i*). If the counter *i* does not occur in the *doclause* or in the condition, then you may omit **for** *i*. If *start* is *1*, you may omit **from** *start*. If *step* is *1*, you may omit **by** step. If you do not want to finish at a certain value of *i*, you may omit **to** *finish* (see also E11 *). If it is **true**, you may omit **while** *condition*.

11-13 The construction :

```
if C1 then C1true
elif C2 then C2true
elif C3 then C3true
     else C3false
fi
```

or, in the abbreviated notation:

```
( C1 | C1true
|: C2 | C2true
|: C3 | C3true
     | C3false
)
```

is shorthand for the nested conditional-clauses:

```
if C1 then C1true
     else if C2 then C2true
               else if C3 then C3true
                          else C3false
                    fi
          fi
fi
```

38–46. The statement *print* is a very accommodating output carrier. It accepts almost everything you may invent to output,

be it a lay-out procedure like:	*new line*	,
or a **string** denotation like:	*"number of measurements: "*	
or a variable like:	*n , below , above*	,
or a unit like:	*sqrt (varf)*	,
or a row-display of them all.		

1–46. In the example E15 we are very strict about the use of variables and constants: we never use a variable when a constant suffices (i.e. when we do not assign to it). Of course, you could declare all identifiers to yield variables; in some implementations, however, a constant might be slightly more efficient. Pay also some attention to the use of and-also-symbols (collateral elaboration of declaration and row-displays) and go-on-symbols (serial elaboration).

```
(E15)  begin
 1)    int  n ;          read (n) ;              ¢ number of measurements ¢
 2)    real  eps , ups ; read ( (eps, ups) ) ;   ¢ eps ⩽ x[i] ⩽ ups ¢
 3)    [1: n]real f , x ;  for i to n
 4)                        do read ( (¢factor¢ f[i] , ¢measurement¢ x[i]) ) od ;
 5)    int  below := 0, ¢ number of measurements rejected: too small ¢
 6)         above := 0, ¢ number of measurements rejected: too large ¢
 7)    real sf    := 0, ¢ sum factors accepted ¢
 8)         sf2   := 0; ¢ sum squared factors accepted ¢
 9)    for i to n
10)    do   real xi = x[i];
11)         sf   +:=   if xi < eps  then  below +:= 1 ; f[i] := 0
12)                  elif xi ⩽ ups  then  f[i]
13)                                 else  above +:= 1 ; f[i] := 0
14)                  fi ;
15)         sf2 +:= f[i] ↑ 2
16)    od ;
17)    int  m = n – below – above ;
18)    real af    = sf/m ,              ¢ mean factor ¢
19)         varf  = sf2/m – af x af ;   ¢ variance of the factors ¢
20)    real sw    := 0, ¢ total weight ¢
21)         swx   := 0, ¢ sum weighted measurements ¢
22)         swx2  := 0, ¢ sum squared weighted measurements ¢
23)         swx3  := 0; ¢ sum cubed weighted measurements ¢
```

```
24)     for i to n
25)     while real fi = f[i]; fi ≠ 0
26)     do      real xi = x[i], wi = exp (-( (fi-af)↑2)/(2 × varf) ) ;
27)             real wixi , wixi2 ;
28)             sw   +:= wi ;
29)             swx  +:= (wixi  := wi × xi) ;
30)             swx2 +:= (wixi2 := wixi × xi) ;
31)             swx3 +:= wixi2 × xi
32)     od ;
33)             ¢ first      , second          , third          moment about 0 ¢
34)     real    ax   = swx/w , ax2 = swx2/w , ax3 = swx3/w ;
35)     real    varx = ax2 - ax↑2 ;
36)     real    sdx  = sqrt (varx) ;
37)     real    skx  = (ax3 - 3 × ax × ax2 + 2 × ax ↑ 3)/(2 × varx × sdx) ;
38)     print ( ( newline ,
39)     "number of measurements: " , n , "below: " , below , "above: " , above ,
40)             newline ,
41)             newline ,
42)     "mean factor: " , af , "dispersion: " , sqrt (varf) ,
43)             newline ,
44)             newline ,
45)     "normal mean: " , ax , "dispersion: " , sdx , "skewness: " , skx
46)         ) )
        end
```

0.7. **Routines and procedures**

A concept of fundamental importance in programming is the "routine", a unit that can be activated from different places in the program, under different circumstances and in different incarnations when elaborated recursively. Moreover, routines may have a provision for formal-parameters, to which the actual-parameters are then supplied when the routine is activated.

Routines, and also their names, may be ascribed to identifiers; we then speak about 'procedures'. In this language, a routine may also be ascribed to an operator. Procedures and operators are to be distinguished in that they are activated differently. Procedures are activated by "calling" them, and operators by applying them in formulae. In this section we shall consider procedures; for operators see 0.10.

0.7.1. Procedures without parameters

With the aid of the symbol **proc** we can derive new modes from already defined ones (as we did with the aid of the symbols [and] in 0.4). We thus obtain one of many possible **proc** modes, the simplest of which is the **proc void** (without parameters, not returning any value).

Suppose we want to turn the algorithm E1* into a procedure. This algorithm is defined by a unit (a closed-clause); we declare it as a **proc void** in the following way:

```
(E16)  proc p = void: ( real x , y , z ; read ( (x, y) ) ;
                        print ( z := (x + y) / 2 – sqrt (x × y) )
                      ) ;
```

The right hand side of this identity-declaration is a routine-text. It yields a corresponding routine – a value of mode **proc void** – which is now ascribed to the identifier *p*. The unit to the right of **void**: is not, of course, elaborated at this stage.

The procedure *p* does something for you: it reads two numbers, does some computation with them and finally prints the result. However, it does not, and it cannot, return any value; *p* is declared to be a procedure returning **void**.

For those who are accustomed to the "procedure body" (a well known concept in some other programming languages), the alternative writing:

```
(E16*) proc p = void: begin real x , y , z ;
                            read ( (x, y) ) ;
                            print ( z := (x + y) / 2 – sqrt (x × y) )
                      end ;
```

may be more familiar.

Within the context of the declaration E16, we can call this procedure. Consider:

```
(E17)  begin  int n ; read (n) ; to n do p od end
```

Between **do** and **od** we find *p*. By virtue of its declaration, *p* is a unit of the mode **proc void**. The elaboration of a unit of the mode **proc void** is (in this syntactic position) the elaboration of its unit. Therefore, the piece of program above is, at least in its effect, equivalent to E4 (see 0.3).

We often want a procedure to return a value. For example, *p* could do just a little more and return the value printed. Inside the closed-clause this value is referred to by ẑ. We already know how to get a closed-clause to yield a

specific value (0.5.4). Now, by prefacing it with **real**: , we arrive at the declaration of a **proc real**:

```
(E18)  proc pz = real: ( real x , y , z ; read ( (x, y) ) ;
                         print ( z := (x + y) / 2 - sqrt (x × y) ) ;
                         z ) ;
```

The mode **proc real** is derived from **real** as, for instance, was [*1*:*n*] **real** .

Within the context of E19, it is not difficult to understand the effect of the following piece of program:

```
(E19)  begin  int n ; read (n) ;
              int less := 0 , morequal := 0 ;
              to n do ( print ( newline ) ; pz < 1 | less | morequal ) +:= 1 od ;
              print ( ( newline, newline , less, morequal) )
       end
```

The *pz* in the formula *pz* < *1* is a unit of the mode **proc real**. The elaboration of a **proc real** unit is (in this syntactic position) the elaboration of the routine it yields; that routine returns a **real** value, and consequently *pz* yields that **real** value.

0.7.2. Procedures with parameters

Even more important than procedures without, are procedures with formal-parameters. The actual-parameters are then supplied when the procedure is called, as we have already done on several occasions when calling the standard procedures *read* , *print* and *sqrt* .

We now declare a procedure *d* with two **real** parameters returning a **real** value, a **proc** (**real** , **real**) **real**:

```
(E20)  proc d = ( real a , b ) real:   (a + b) / 2 - sqrt (a × b) ;
                 | 1   || 2  |
                 |-----------|
                 |-----------3-----------|
```

1. These are the formal-parameters *a* and *b* which are both specified to be of mode **real**. The actual-parameters have to match this mode. There are, however, certain facilities in this respect. If, for instance, a **ref real** is supplied, then its **real** value will be taken; if an **int** is supplied, then it will be "widened" into a **real**.
2. The prefix **real**: requires that the routine is to return a **real** value, when called.

3. The whole sequence of symbols to the right of the = is a 'routine-text'. It yields a routine, which requires two **real** parameters. The elaboration of a call in which actual-parameters are then supplied to match these formal-parameters, is the subject matter of 1.2.3 and 4.2.2.2. It will suffice, here, to state that *a* and *b*, as formally declared in our example E20, will be **real** constants in the routine (when, for instance, *x* and *y* are supplied as actual-parameters, then the identity-declarations **real** *a* = *x*, *b* = *y* ; will be elaborated). That is to say, the actual-parameters are elaborated once, to supply their **real** values. In other programming languages this phenomenon may be known as "call by value" (see also 0.8.2 and 0.8.3).

Within the context of the declaration E20, the algorithm E1, for example, could be programmed in the following way:

```
(E21)  begin  real x , y , z ; read ( (x, y) ) ;
              print ( z := d(x, y) )
       end
```

The assignation E12 (see 0.5.4) could look like:

```
(E12*) p := ( real x , y , z ;
              to n do read ( (x, y) ) ;
                    if d(x, y) > 1 then finish fi
                 od ;
              false exit
              finish : print ( (x, y) ) ;
              true
            )
```

An example of a procedure with a parameter but not returning a value is:

```
(E22)  proc skip = (int n) void:
                    ( real x ; to n do read (x) od ) ;
```

which skips *n* numbers on the input tape.

0.7.3. Examples of procedure declarations

We have four kinds of procedures:

1) without parameters, not returning any value ;
2) without parameters, returning a value ;
3) with parameters, not returning any value ;
4) with parameters, returning a value .

Examples:

```
proc skiptozero =          void: ( loc real x := 1 ;
                                   while x ≠ 0 do read (x) od )

¢ which was a proc void ¢ ;
proc nexttozero =          real: ( skiptozero ;
                                   loc real x ; read (x) ; x )

¢ which was a proc real ¢ ;
proc skipto     = ( real a) void: ( loc real x := a−1 ;
                                   while x ≠ a do read (x) od )

¢ which was a proc ( real ) void ¢ ;
proc nextto     = (real a) real: ( skipto (a) ;
                                   loc real x ; read (x) ; x )

¢ which was a proc(real) real ¢ ;
```

In the standard prelude of the language, we find declarations for the **proc** *(real)* **reals** *sqrt , exp , ln* (the natural logarithm) , *cos , arccos , sin , arcsin , tan , arctan* . Moreover there is a **proc real** *random,* which returns the next pseudo-random **real** value from a uniformly distributed sequence on the interval [0,1) (i.e. $0 \leqslant random < 1$). Finally we find in the standard prelude an identity-declaration **real** *pi* = **c** *a* **real** *value close to* π **c** . (See 1.3.2 for the significance of the special comment-symbol **c**).

To these we subjoin:

```
(D4)  proc ncos = (int i) real: cos ( 2 × pi × i/n ) ; ¢ a proc ( int ) real ¢
      proc nsin = (int i) real: sin ( 2 × pi × i/n ) ; ¢ a proc ( int ) real ¢
      real e = c a real value close to the base of natural logarithms,
                 i.e. 2.718281828459045 c ;
```

Another example of a **proc(int)real** declaration is:

```
(E23)  proc fac = ( int n) real:
                    if n > nmaxfac
                  then faclarge (n)
                   else int f := 1 ;
                        for i from 2 to n do f x:= i od ; f
                   fi ;
```

where *nmaxfac* is an implementation dependent integral constant which is related to *maxint* in the following way:

```
(E23*)  int nmaxfac = ( int n , f := 1 ;
                        for i while f ⩽ maxint ÷ i do n := i ; f × := n od ; n ) ;
```

and *faclarge* is another **proc(int)real:**

```
(E23**) proc faclarge = ( int n) real:
                        c depending on nmaxfac :
                          stirlings formula with correction factor ,
                          sqrt (2 x pi x n ) x ( n/e ) ↑ n x corr (n)
                          or some series expansion for 1/gamma (n) ;
                          see "Handbook of Mathematical Functions"
                          edited by Milton Abramowitz and I. E. Stegun
                          Sections 6.1.37/38 and 6.1.34 c ;
```

The **proc(int)real** *fac* , as declared in E23, attempts to return the exact **real** equivalent of $n!$ as long as this value remains $\leqslant$ *maxint* (the critical value of n is yielded by the constant *nmaxfac*); otherwise, a **proc(int)real** *faclarge* is called to give a reasonable approximation.

Formal-parameters may be of any mode and procedures may return a value of any mode. The time has not yet come for the more arresting examples, which is why we confine ourselves to two simple ones. Both of them will put in another appearance in 0.8.2, because their efficiency can be improved. The starred example numbers refer to the unstarred numbers in Section 0.8.2.

```
(E27*) proc maxindex = ( [ ] real a) int:
                       ( int j := lwb a
                         for i from j + 1 to upb a
                         do if a[i] > a[j] then j := i fi od ; j ) ;
```

In E27* we see the declaration of a procedure with a 'row of real' parameter, returning an **int**, a **proc(** [ ] **real)int** ; *maxindex* returns the index of the maximal element in a given row (if there are more "maximal elements", then the lowest of their indices is returned).

The monadic-operators **upb** and **lwb** return the values of the upper-bound and lower-bound respectively; being declared in the standard-prelude of the language, they are permanently built in and are applicable to all kinds of multiples. If the multiple has several subscripts, they are dyadic-operators (so that, for instance, *2* **upb** *x2* returns the second upper-bound of the row-row-of-real *x2*).

The **proc(char,string)bool** *match* , declared below, returns **true** if the character ascribed to the formal-parameter *c* occurs in the given **string**; if not, then the value returned by *match* will be **false**. In this procedure we make use of a completer (see 0.5.4):

```
(E28*) proc match = ( char c , string s) bool:
                    ( for i from lwb s to upb s
                      do if c = s[i] then yes fi od ; no: false exit
                                          yes:          true ) ;
```

0.8. The creation of new modes, names and values referred to

0.8.1. Variable declarations revisited

It is time to reconsider the variable-declaration:

(E24) **loc real** *x* ;

because it is not so innocent as it looks.

You might already have suspected this, knowing what its elaboration achieves (see 0.1):

> a location for a **real** value is reserved in the memory
> of our computer (on the stack);
> this **real** value is referred to by a name x̂ (i.e. its address);
> this name x̂ is now ascribed to the identifier *x*.

In the program text, the identifier *x* thus represents a **real** variable, which is a **real** value associated with the name referring to it. What is "variable", of course, is not the name (the location, the address) but the value referred to. The name cannot be changed by the program, it has been ascribed to *x* and this relation is an indissoluble alliance. Nevertheless, a "name" is a value as well, and consequently it must have a mode. The mode, then, of the name x̂ is **ref real** ('reference to real').

The symbol **ref** plays a role in declarations as do the symbols [and] and **proc** (and some others which we shall meet soon): they assist in the creation of new modes. We now come to the unmasking of E24. Consider the identity-declaration:

(E24*) **ref real** *x* = **loc real** ;

which expresses more precisely what happens during the elaboration of E24:

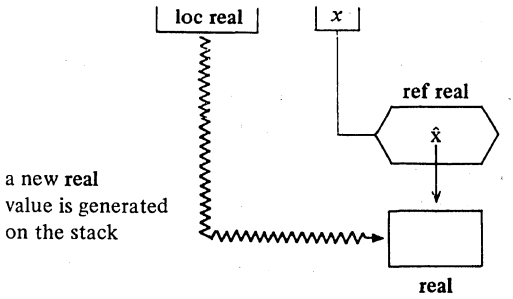
1) **loc real** is a **loc** generator which yields a **ref real**
2) this **ref real** value is ascribed to the identifier *x*.

Thus the effect of E24* is the same as that of E24 and, indeed, every variable-declaration has an equivalent identity-declaration hiding behind it.

The point to remember is the hidden fact that, on declaring a **real** variable, two values are involved:

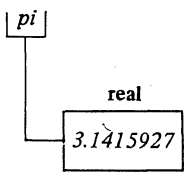
1) a **real** which is variable.
2) a **ref real** which is constantly yielded by the identifier,

Getting ahead of Section 1.2.2.3 we may depict a variable-declaration happening as follows:

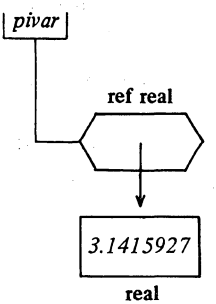


which is the same as the situation depicted in 0.1. For a more systematic treatment see Chapter 1.

The result of the declarations E13 and E13* (see 0.5.5) can be depicted as follows:



To *pi* no other value can be assigned, *pi* being a constant.



To *pivar* any other value can be assigned, *pivar* being a variable.

Finally, we give some declarations in their three equivalent forms. The first indicates most clearly what is actually happening, although in practice one will always use one of the others (which are also more similar to declarations in other programming languages).

identity-declaration :		variable-declaration :	
ref bool	*p* = **loc bool** ;	**loc bool** *p* ;	**bool** *p* ;
ref int	*n* = **loc int** ;	**loc int** *n* ;	**int** *n* ;
ref real	*x* = **loc real** ,	**loc real** *x* , *y* ;	**real** *x* , *y* ;
	y = **loc real** ;		
ref char	*c* = **loc char** ;	**loc char** *c* ;	**char** *c* ;
ref [ ] **real**	*x1* = **loc** [1 : *n*] **real** ;	**loc** [*1* : *n*] **real** *x1* ;	[*1* : *n*] **real** *x1* ;

0.8.2. Procedures, values and references

One of the implications of the extremely fruitful concept of a reference ("name") as a value in this language is that we have been given a quite natural way of declaring (amongst many other very useful constructions):

I variable procedures,
II formal-parameters which refer to, rather than yield, the values which are topical ("actual") when the procedure will be called,
III procedures returning a name (a reference to a value).

We discuss briefly these three applications of the concept of a name by giving some examples.

I)

By declaring:

(E25) **mode fun = proc (real) real ;**
loc fun *f ;*

the identifier *f* is declared to be a variable **fun** (to yield the name referring to a routine of the mode **proc(real)real**).

Within the context of this declaration, we may assign any **fun** routine to this **fun** variable. For example:

f := *ln*

Now the call:

y := *f*(*x*) is the call *y* := *ln*(*x*)

while after the assignation:

f := (**real** *a*) **real**: *a* × *ln*(*a*) − *a*

the same call becomes:

y := *x* × *ln*(*x*) − *x*

II)

Even more important is the specification of a formal-parameter to be a name:

(E26) **proc** *read fun* = (**ref real** *a*) **real**: (*read*(*a*) ; *a* := *f*(*a*)) ;

E26 declares the formal-parameter to be a **ref real** (the name of a **real**) to which, consequently, a **real** value can be assigned. This happens two times in the procedure: first the value of the number read is assigned to *a* (by the call

of the procedure *read*), and then the *f* of that value is assigned to *a*. If we now call this procedure, for example:

y := *read fun(x)*--*y*

then the formal – actual correspondence results in:

ref real *a* = *x* ;

i.e. the formal name â is made to refer to the same **real** value as the actual name x̂. The call *read fun(x)* thus results in assigning to *x* the *f* of the value read. Which **fun** is then applied depends (in the context of E25) on the **fun** assigned to *f*.

In other programming languages, a **ref** parameter may be known as an "output parameter". In contradistinction to the more or less domesticated term "call by value" in ALGOL 60 (see 0.7.2), this could be termed "call by reference". For an equivalent of the ALGOL 60 term "call by name" see 0.8.3.

We are now in a position to improve on the examples at the end of 0.7.3. There, the value parameters imply that a copy of the actual rows is to be passed to the routine. Unfortunately, not all implementors have been able to avoid actually making this copy. We now show how to avoid the problem altogether.

(E27)
```
proc maxindex = ( ref [ ] real a) int :
                ( int j := lwb a ;
                  for i from j + 1 to upb a
                  do if a[i] > a[j] then j := i fi od ;
                  j ) ;
```

Now, in the call *maxindex(x1)*, only the name x̂1 is given to the procedure: **ref**[ ] **real** *a* = *x1*. The access to the [ ] **real** referred to by x̂1 thus runs via the formal name â to that very [ ] **real** and not (as was the case in 0.7.3) to a copy of it.

(E28)
```
proc match = ( char c , ref string s ) bool:
             ( for i from lwb s to upb s
               do if c = s[i] then yes fi ; no : false exit
                                  yes:       true ) ;
```

Now, in the call *match("?", text)*, in the context of the declaration **string** *text*, the character "?" is copied onto the stack, but the name tex̂t is given to the procedure instead of a copy of its value.

III)

An example of a procedure returning a name is:

(D5) **proc** *xory* = **ref real**: **if** *random* < *0.5* **then** *x* **else** *y* **fi** ;

If we now assign:

xory := *3.1415927*

then it depends on the value returned by *random* to what destination we actually assigned.

A more substantial example is:

```
(E29)   proc maxelmnt = (ref [ ] real a) ref real:
                        ( int j := lwb a ;
                          for i from j + 1 to upb a
                          do if a[i] > a[j] then j := i fi od ;
                          a[j] ) ;
```

Compare E29 with E27. In E29, â[*j*] is a name referring to a maximal element of the actual row, when the procedure is called. If you want to assign a new value to the maximal element of, for instance, the row *x1*, then you could do it by a call of *maxindex* :

x1 [*maxindex (x1)*] := *y*

but also, and more directly, by a call of *maxelmnt* :

maxelmnt (x1) := *y*

0.8.3. Procedures as formal parameters

Compare:

```
(E30)   proc choice1 = (      real a) void: ( a < 0.5 | x | y ) +:= a ;
        choice1 (random) ;
```

with:

```
(E31)   proc choice2 = ( proc real a ) void: ( a < 0.5 | x | y ) +:= a ;
        choice2 (random) ;
```

In the call *choice1 (random)*, the identity-declaration **real** *a* = *random* will be elaborated. Hence, the **proc real** *random* is called only once and its value (<0.5 or $\geqslant 0.5$) is added to *x* or to *y*.

In the call *choice2 (random),* the identity-declaration **proc real** *a = random* will be elaborated. Hence, the **proc real** *random* is ascribed to *a* inside the routine and will be called once in the conditional-clause and again as the right operand of the +:= (i.e. *random* will be called twice in E31).

The construction E31 is similar to what in ALGOL 60 is known as "call by name"; it has, nevertheless, nothing to do with the concept of a name in ALGOL 68 (in which it is an application of the principle that any mode may occur in a formal-parameter, in particular also a **proc real**).

Compare also:

(E30*) *choice1 (x – y + ncos(k))*

and:

(E31*) *choice2 (* **real**: *x – y + ncos(k))* ¢ see D4 in 0.7.3 ¢

In E30*, the formula *x – y + ncos(k)* will be elaborated once in the elaboration of the identity-declaration **real** *a = x – y + ncos(k).* Depending on the condition, this value (inside the routine ascribed to *a*) will be added to *x* or to *y,* so that the result of the call will be:

x +:= a i.e. *x +:=* ¢ the value of ¢ *x – y + ncos(k)*

or

y +:= a i.e. *y +:=* ¢ the value of ¢ *x – y + ncos(k)*

In E31*, on the contrary, the formula *x – y + ncos(k)* appears in a routine-text (the mode of which is **proc real**) and it is the routine yielded by **real**: *x – y + ncos(k)* which is ascribed to the **proc real** identifier *a* at the elaboration of the procedure-call. Hence, the formula will be elaborated twice (once in the conditional-clause and again as the right operand of the +:=). The two successive elaborations, however, yield the same value, which is why the two calls E30* and E31* have the same result (though E30* is the more efficient one).

However, compare now:

(E30**) *choice1 (x – y + ncos(k +:= 1))*

and:

(E31**) *choice2 (* **real**: *x – y + ncos (k +:= 1))*

Each elaboration of the formula *x – y + ncos(k +:= 1)* has a side effect on *k* (*k := k + 1*) and therefore the two calls will not have the same result; in E30** the formula being elaborated once, and in E31** twice.

Another construction in which we find a procedure as formal-parameter (in ALGOL 60 circles known as "Jensen's device") is:

(E32)
```
proc sigma = (int a , b , proc(int)real fun) real :
                ( real value := 0 ;
                  for i from a to b do value +:= fun(i) od ;
                  value ) ;
```

calls of which may be:

```
y := sigma ( 1 , n, (int j) real: x1 [j] ) ;
y := sigma ( −m, +m , ncos )
```

It is worth your while to find out why we have to give the **proc(int)real** routine-text *(**int** j)**real**: x1 [j]* as actual-parameter in the first call, and why *ncos* would do in the second call.

0.8.4. Pointers (variable names)

Until now, names have always appeared as being constantly yielded by identifiers. By:

```
loc real x ;
```

or:

```
ref real x = loc real ;
```

we declared x to yield constantly a **ref real** (the name of a **real**). Only the **real** value referred to could be changed and never the name $\hat{x}$.

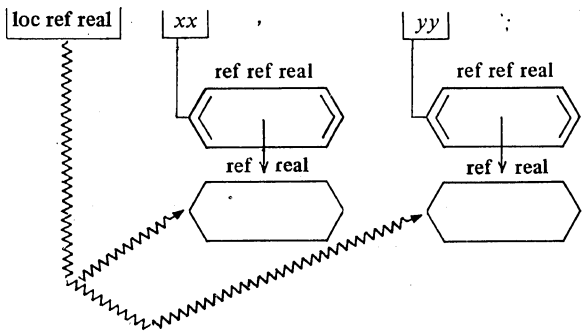
You may, however, also want to declare identifiers to yield variable names, i.e. references to names. It may be clear immediately that such an identifier will then yield a reference-to-reference. We thus declare:

(D6)
```
ref real xx , yy ;
```

or equivalently

(D6*)
```
ref ref real xx = loc ref real ,
                  yy = loc ref real ;
```

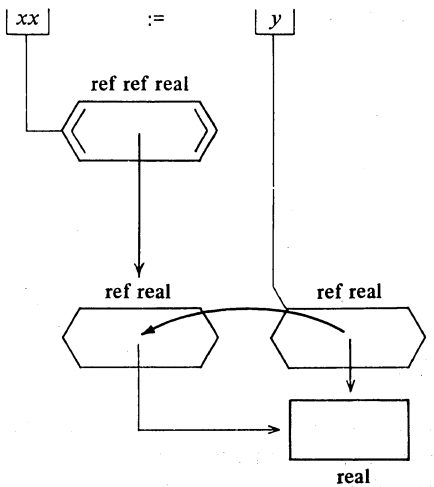
This collateral-declaration generates on the stack two **ref reals** (two locations for names of **reals**), which simply means that via such values on the stack you can refer to other values (**reals**):



Consider now:

(E33) $xx := y$

After this assignation, the value of xx (i.e. the **ref real** referred to by $x\hat{x}$) is the name $\hat{y}$ (i.e. the **ref real** yielded by y):



Observe that an assignation always takes place at the highest level possible. Now the call:

$y := sqrt(xx)$

results in: $y := sqrt(y)$, while after the assignation:

(E33*) $xx := x$

the same call results in

y := sqrt(x)

In computer oriented programming this is known as "indirect addressing" (in a certain location in the memory one finds the address of another value). In some other programming languages (in particular in assembly languages) a **ref ref** may be known as a "pointer" (i.e. by D6, *xx* and *yy* yield "pointers" to **real** variables, addresses of addresses of **real** values).

0.8.5. Identity relators, the cast, coercion

Where names are values in this language and may be manipulated as all other values (see 0.8.4), you may want to ask whether two names of the same mode are the same name. Neither the equals-symbol =, nor its negation ≠ can serve this purpose, because they are operators defined to compare values of certain modes only (and these values are mostly not names). To compare names, we have the identity-relators :=: (or **is**) and :≠: (or :/=: or **isnt**). For example, after the assignation:

y := x

it is most certainly **true** that

y = x

but it is also **true** that nevertheless:

(E34) *y* **isnt** *x* or *y* :≠: *x*

because *y* and *x* yield different names (references to different locations on the stack).

(E35) *y* **is** *xory* or *y* :=: *xory*

however, is **true** or **false** depending on the value lastly yielded by *random* (see D5 in 0.8.2) and independently of whether *y = x* or *y ≠ x* .

Observe also that after the assignations:

xx := y ; yy := xx

or, in one phrase:

yy := xx := y

nevertheless:

yy :≠: *xx* or *yy* **isnt** *xx*

because *yy* and *xx* yield different names (the mode of which is **ref ref real**); reconsider E34 and its motivation.

One might now be disappointed that after the assignation *yy* := *xx* := *y*, nevertheless *yy* **isnt** *xx*. In the nature of things it must be possible to get an answer to whether names assigned to different pointers are the same. But then the right question must be asked, and this question lies one level of reference below the question *yy* :=: *xx* .

We could go down one level in reference by declaring, for instance, the **proc(ref ref real)ref real**

proc *the name assigned to* = (**ref ref real** *aa*) **ref real**: *aa* ;

and then we get undoubtedly the proper answer when we call this procedure to the left and to the right of the identity-relator:

the name assigned to (xx) :=: *the name assigned to (yy)*

The **ref real**: in front of the reference-to-reference-to-real-unit *aa* "coerces" this unit to yield its reference-to-real value. Fortunately, it is not necessary to declare such a monstrosity of a procedure. You may write:

(E36) **ref real** *(xx)* :=: **ref real** *(yy)*

The technical term for **ref real** *(xx)* is 'cast'. A cast, in general, coerces its body to yield a value of the mode it dictates (if possible).

The cast may also be used to assign a value to the name assigned to a pointer:

(E37) **ref real** *(xx)* := *y*

comes, in the context of E33*, to the same as:

x := *y*

You might ask now why we did not meet the cast at a much earlier stage; why, for example, we did not have to write:

a := **real** *(b)* , which, by the way, is correct ALGOL 68

The answer is that in all current situations where it is clear from the context what you want, your computer will be so kind as to coerce your units to your will. As a matter of fact, "coercion" is the technical term for the provision that:

when no ambiguities make trouble,
your units will be implicitly coerced to the mode you
apparently require.

The time has not yet come to discuss all the slings and arrows of coercion. Here we set a pointer to 1.1.6 and another one to 5.1.0.

0.9. Structures and other new modes

Besides the multiple, you will find in this language another system that gives you control of a collection of values, and that is in the form of a structured value (or "structure" for short). The individual values in a multiple are its "elements", the individual values in a structure are its "fields". The elements in a multiple are all of the same mode, [*1:n*] **real**, [*1:80*] **char** , [*1:5*] **proc void** etc.; the fields of a structure on the contrary may be of different modes, although there are very useful (even standard) applications where the field modes are the same.

0.9.1. **compl**ex values, vectors etc.

By declaring:

(D7) **mode compl** = **struct** (**real** *re* , **real** *im*) ;
¢ another built in mode like **string** ¢

or

mode compl = **struct** (**real** *re* , *im*) ;
¢ an obvious contraction ¢

a new mode **compl** (complex) is defined consisting of two **real** values; one of them is selected by the field-selector *re*, the other by *im*. Although field-selectors look the same as identifiers, you must not confuse them.

Now the variable-declarations:

(D8) **compl** *w* , *z* ;

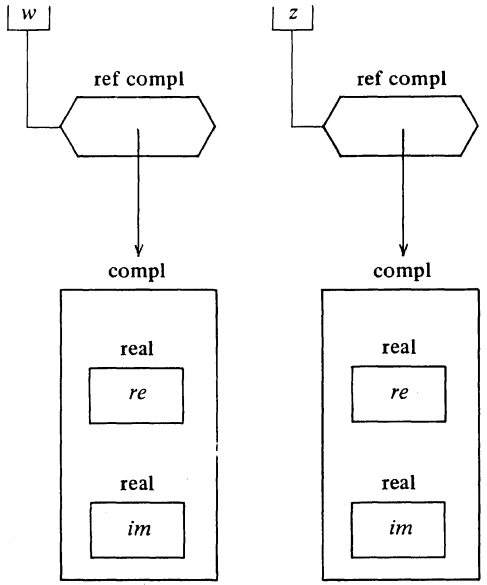
ascribe to *w* and *z* the names ŵ and ẑ which refer to structures as defined in D7 (i.e. *w* and *z* are **compl** variables).

The fields of *w* and *z* (the real and imaginary part of these complex variables) may be selected as follows:

(E38) *x* := *re* **of** *z* ; *y* := *im* **of** *z* ;

The *re* **of** and *im* **of** select the field in much the same way as, for instance, [*i*] , [*i,j*] etc. select the element in a multiple. There is, however, an important distinction: the selection of a field may be done at compile time, whilst the selection of an element in a multiple usually involves computation and, consequently, can then only be done at run time.

After the declarations D8, the identifiers *w* and *z* yield **ref compls**; the situation may be depicted as follows:



The result of the assignation:

$z := w$

should be obvious. You may, however, also assign:

(E39) $z := (x , y)$

which amounts to:

re **of** $z := x$, *im* **of** $z := y$

The (x, y) in this context is a structure-display, the counterpart of a row-display.

The mode **compl** is built into the standard declarations of the language, and operators = , ≠ , + , – , × , / , ↑ , and a monadic operator **conj** are declared for it with the meanings to be expected. Moreover, we have (monadic) operators **re** , **im** , **abs** and **arg** which return a **real** when applied to a **compl**, and an operator ⊥ (or **i** or +×), which may be pronounced as plus-i-times, which makes a **compl** of two **real** operands:

$z := x \perp y$ or $z := x +\times y$

amounts to:

z := (x, y)

(see also 0.10.4).

As from **int** to **real**, there is automatic widening from **real** to **compl** (and via **real**, from **int** to **compl**). The assignation:

z := x := i := 1

results in *i = 1 , x = 1.0 , z = (1.0 , 0.0)*

Other examples of new modes defined by structures with fields of the same mode might be:

(D9)
```
mode vec      = struct ( real xcoord , ycoord , zcoord ) ;
mode rational = struct ( int numerator , denominator ) ;
vec v1 , v2 , v3 ;
rational r1 , r2 , r3 ;
```

These modes, however, are not built into the standard declarations. If you want to use them for new kinds of operands, then you have to declare operators for them; this can easily be done (see 0.10.7, 8.4.1 and 8.4.2).

0.9.2. Structures with mixed mode fields, chains etc.

The really interesting feature of structured modes is, however, that you can collect values of different modes into them. For example:

(E40)
```
mode book = struct ( string text , int index ) ;
     book revised report on the algorithmic language algol 68;
```

Now the field:

text **of** *revised report on the algorithmic language algol 68*

contains the **string**: *"may be difficult for the uninitiated reader"* [see R 0.1.1], and:

index **of** *revised report on the algorithmic language algol 68*

might be the point where you really got stuck.

There is, however, another

book *informal introduction to algol 68 revised edition*

which also contains the **string** denoted, but in quite another context. May this other **book** help you to proceed at the *index* **of** referred to above.

An important implication of the concept of mixed mode fields is that you can make a field refer to another structure of that same mode. In this way chains (lists, queues, etc.) can be defined in a most natural manner. For example:

(E41)
```
mode volume = struct ( string text , int index ,
                       ref volume companion , next ) ;
volume report , informal introduction ,
       revised report , informal introduction revised ;
```

The assignations:

(E42)
```
companion of report := informal introduction ;
companion of revised report := informal introduction revised ;
```

have been made. Moreover we can report that:

(E43)
```
next of report := revised report ;
next of informal introduction := informal introduction revised ;
```

There will be no *next* **of** *revised report,* neither a *companion* **of** *informal introduction* nor **of** *informal introduction revised* nor a *next* **of** *informal introduction revised.* To express this, we assign:

(E44)
```
next of revised report :=
companion of informal introduction :=
companion of informal introduction revised :=
next of informal introduction revised := nil ;
```

where **nil** is the same as "a reference to no value at all".

We may speak about:

```
text of companion of revised report
```

which is the text under your very eyes at this moment.

It is important to comprehend the mode of the *companion* and *next* fields of, for example, *revised report* and *informal introduction revised.* To these identifiers **volume** variables have been ascribed and their mode is, consequently, **ref volume.** Likewise, and this is the important point, *companion* **of** *revised report* yields a **ref volume** variable, and so its mode is **ref ref volume**; hence, *companion* **of** *revised report* is a "pointer" as are *companion* **of** *report, next* **of** *report* and *next* **of** *informal introduction.*

From this it follows that in E42 and E43 names have been assigned to pointers rather than (non **ref**) values – you will not find the *text* **of** *informal introduction revised* in the *revised report,* but via the pointer *companion* **of**

revised report you find a reference to that text. A more complete discussion of this rather delicate matter will be found in Sections 1.4.3 and 5.4.2.

To conclude, we consider an example in which names will be assigned to **ref** fields.

(E45)
```
mode card = struct ( int value , string colour , picture ,
                     ref card next ) ;
mode deck= struct ( int number , ref card one ) ;
[0:51] card card ;
deck myhand , yourhand ;
```

In E45 we declared 52 **card**s and two **deck**s, each of which may refer to a certain **card** by its field *one* **of** *myhand* (*one* **of** *yourhand*). Which **card**s you and I then have in our hands may now follow from the fields *next* **of** *card*[*i*]; *number* **of** *yourhand* may be the number of **card**s you have in your hand.

Leaving the value, colour and picture of the **card**s for what they are, we may assign:

(E46)
```
for i from 0 to 51 do next of card[i] := card [ (i + 1) mod 52] od ;
yourhand := ( 52, card[0] ) ; myhand := ( 0 , nil )
```

where **mod** is the standard operator yielding the value of the left-operand modulo the right-operand. We thus arranged our 52 **card**s in a circular chain and you have them all in your hand.

You may now remove a **card**, say *card*[*37*], by:

```
next of card [36] := card [38]
```

and give it to me:

```
one of myhand := card [37] ;
next of card [37] := card [37]        ¢ we made card [37]
                                        selfreferring ¢
```

which you could have done in one statement:

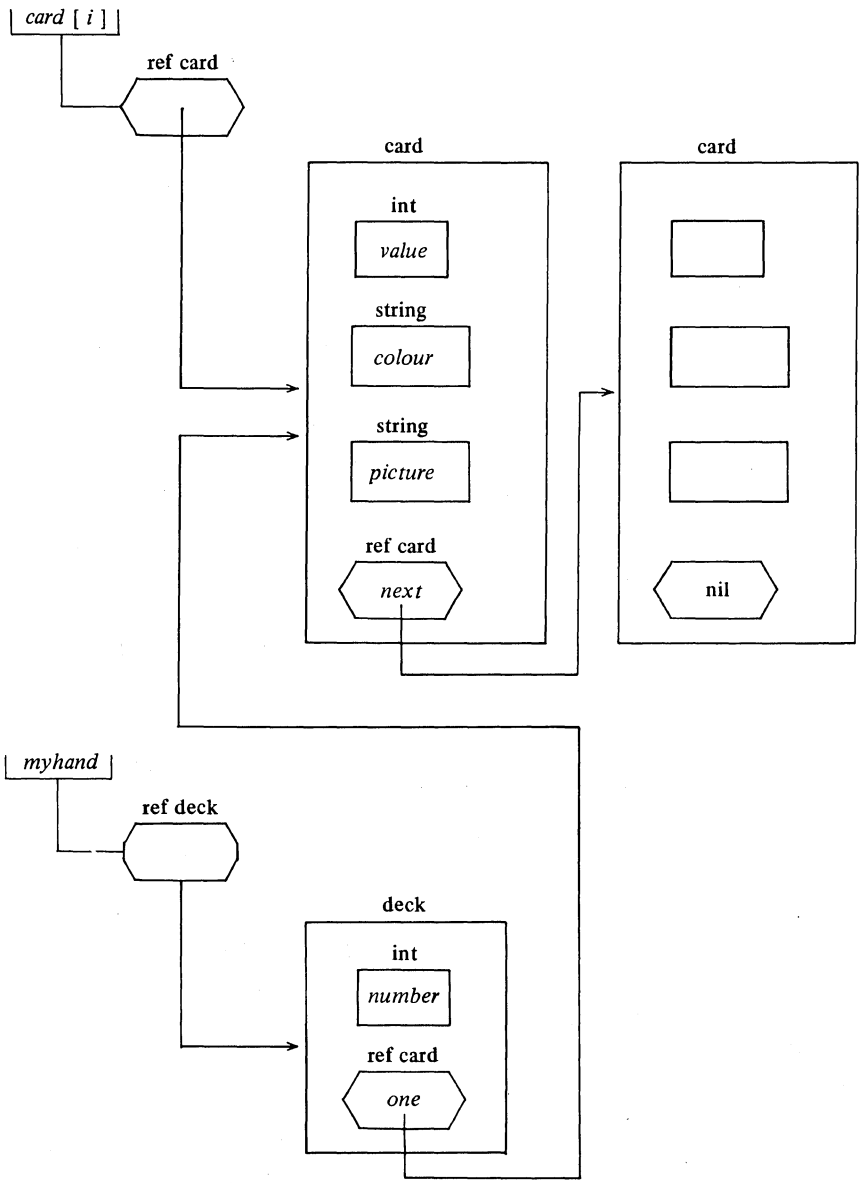
```
one of myhand := next of card [37] := card [37]
```

You may give me another **card**:

```
next of card [28] := card [30] ;
next of card [37] := card [29] ;
next of card [29] := card [37]
```

We may administer these two events by:

```
number of yourhand −:= 2 ;
number of myhand   +:= 2
```


card [i]
ref card
card
int
value
string
colour
string
picture
ref card
next
card
nil
myhand
ref deck
deck
int
number
ref card
one

0.10. Routines and operators

A routine may be ascribed to an operator (see 0.7). An operator is represented by a symbol, such as:

```
+ – x / ÷ ↑ ↓ ∨ ∧ ¬ ⊥ < ⩽ = ≠ ⩾ >
<= >= +x /x
+:= –:= x:= /:=  etc.
```

or by a bold word (representing a symbol), such as:

over up down or and not i lt le eq ne ge gt
plusab minusab timesab divab upb lwb re im etc.

You can make as many operators as you need:

plus minus times div pow
nor impl parl perp a b c m n o p etc.

We have to distinguish between two kinds of operators:

1) monadic: the routine has one formal-parameter;
a monadic-operator is always applied to the operand to its right (prefix notation) ,

2) dyadic: the routine has two formal-parameters ;
a dyadic-operator is always applied to the operand to its left (the left-operand) and to its right (the right-operand), (infix notation).

The monadic operators always have a higher priority than all the dyadic ones; for the latter nine priority levels are provided. The priority of a newly defined operator is declared by a priority-declaration:

prio o = *3 ;*

An operator will normally be used to return a value. The natural use of operators is in formulae. Routines yielded by operators are therefore always routines with either one or two formal-parameters, usually returning a value. Operation-declarations look like procedure-declarations; the only difference lies in the use of **op** instead of **proc.**

In this section we shall confine ourselves to taking over some of the operation-declarations from the standard-prelude [R 10.2] . They may speak for themselves and reading them is one of the ways of learning the language. At this point (or at a point a little further on in this section), you might

decide to try some close reading in R 10.2.3. Anyhow, the examples we give here illustrate how to declare and use your own operators.

0.10.1. Operations on boolean operands

R 10.2.3.2

```
op abs  = (bool a) int:   if a then 1 else 0 fi;
op ¬    = (bool a) bool:  if a then false else true fi;
op ∨    = (bool a, b) bool: if a then true else b fi;
op ∧    = (bool a, b) bool: if a then b else false fi;
op =    = (bool a, b) bool: (a ∧ b) ∨ (¬a ∧ ¬b);
op ≠    = (bool a, b) bool: ¬(a = b);
```

These declarations express neither more nor less than the fundamental truth tables of elementary boolean algebra. You might subjoin in your own library (it is not in the standard-prelude):

```
prio impl = 5;
  op impl = (bool a, b) bool: ¬(a ∧ ¬b);
```

Pay some attention to the definition of equality of two **boolean** operands: the first occurrence of the symbol = is the operator to be defined; the second occurrence is the is-defined-as-symbol which is part of all identity- and operation-declarations.

0.10.2. Formulae

Routines ascribed to operators are activated by the formulae in which those operators occur. Compare, for instance:

(E47)
```
op   ∨  = (bool a, b) bool: if a then true else b fi;
proc or = (bool a, b) bool: if a then true else b fi;
```

Both ∨ and *or* yield (different instances of) the same routine. Therefore, the formula $p \vee q$ returns the same value as the call *or(p,q)*.

There is, however, a very important distinction between the elaboration of a formula and that of a call. In a procedure-call it is the procedure-identifier (irrespective of the actual-parameters) which appoints the routine to be activated. In a formula, the mode of the operands, as well as the operator itself, is taken into account. In a procedure-call your computer will be so kind as to coerce (if possible) your actual-parameters until they match the

modes required by the formal-parameters (**int**s may be widened into **real**s and **real**s into **compl**s etc.). In formulae this kindness is restricted. There may be many different occurrences of the same operator token, yielding different routines depending on the (then necessarily different) modes of the formal-parameters. Therefore, the modes of the actual operands have a firm vote in the election of one of the routines nominated under the same operator.

Moreover, the same symbol may occur as a monadic- as well as a dyadic-operator, even in the same formula. It is always immediately clear from the context which of these two possibilities applies.

Let there be declared:

(E48.0) **proc** *eq* = (**compl** *a, b*) **bool**: **abs** *a* = **abs** *b* ;
(E48.1) **op** = = (**compl** *a, b*) **bool**: (*re* **of** *a* = *re* **of** *b*) ∧ (*im* **of** *a* = *im* **of** *b*) ;
(E48.2) **op** = = (**real** *a, b*) **bool**: $a \leqslant b \wedge a \geqslant b$;
(E48.3) **op** = = (**int** *a, b*) **bool**: $a \leqslant b \wedge a \geqslant b$;
(E48.4) **op** = = (**bool** *a, b*) **bool**: $(a \wedge b) \vee (\neg a \wedge \neg b)$;
(E48.5) **op** ¬ = (**bool** *a*) **bool**: (*a* | **false** | **true**) ;
prio ¬ = 3 ;
(E48.6) **op** . ¬ = (**bool** *a, b*) **bool**: $a \wedge \neg b$;

Now:

$w = z$	invokes E48.1
$x = y$	invokes E48.2
$i = j$	invokes E48.3
$p = q$	invokes E48.4

but:

eq(w,z), *eq(x,y)*, *eq(i, j)* — all three call E48.0 — because *w* and *z* are **ref compl**s; *x* and *y* will be widened; *i* and *j* will be widened — to **compl**

however:

eq(p,q) is undefined, because a **bool** cannot be coerced to **compl**

but:

*eq(***abs** *p* , **abs** *q)* calls E48.0 , because **abs** *p* and **abs** *q* yield **int**s which will be widened to **compl**

and in:

¬p ¬¬q	the first	occurring ¬ is monadic, and invokes E48.5
or	the second	occurring ¬ is dyadic , and invokes E48.6
(¬p) ¬(¬q)	the third	occurring ¬ is monadic, and invokes E48.5

Apart from owning a routine, operators also have a certain priority which determines the parsing of the formula in which they occur. By inserting brackets in a formula, a different parsing can be obtained from that required by the "natural" priorities; in fact, priorities serve to avoid brackets.

Take, for instance, the formula *(a ∧ b) ∨ (¬a ∧ ¬b)*. The operators ∧ and ∨ have different priorities (**prio** ∧ = *3* , ∨ = *2*); consequently, the brackets are superfluous: *a ∧ b ∨ ¬a ∧ ¬b* yields the same value, though in a less transparent manner.

The monadic ¬ is of higher priority than any dyadic-operator and therefore the brackets in, for instance *¬(a ∧ ¬b)* are essential.

0.10.3. Operations on arithmetic operands, the standard prelude

Maybe a certain amazement will fall upon you in Section 10.2.3 of the Report. You will find there declarations such as:

[R 10.2.3.3.i] **op** += *(***int** *a, b)* **int**: *a − − b ;*

[R 10.2.3.3.m] **op** ÷= *(***int** *a, b)* **int**:
if *b ≠ 0*
then int *q* := *0* , *r* := **abs** *a ;*
while *(r −:= **abs** b) ⩾ 0* **do** *q* +:= *1* **od** *;*
if *(a < 0* **and** *b ⩾ 0)* **or** *(a ⩾ 0* **and** *b < 0)*
then *−q* **else** *q* **fi**
fi *;*

and even worse, because you will also meet tokens not belonging to the language such as **L** , **P** , **Q** , **R** , **E** etc. which provide a kind of shorthand for the standard-prelude only.

These definitions of arithmetic operations (the meaning of which will be known to every programmer) have nothing to do with machine efficiency. To the contrary, their justification lies in the fact that they "fix the semantics" by defining all operations in terms of a certain minimal set of primitive operations. Agreement with the choice of this set and whether you like this method or not is a matter of taste (maybe even of philosophy); it has nothing to do, however, with the language defined. This is entirely a problem of how to define a language.

Moreover, a routine defines a series of actions in a computer and it is explicitly stated in the Report [R2.1.4.1.a] that any of these actions may be

replaced by any other action which causes the same effect. Consequently, an implementor is perfectly free to supply means for generating (more) efficient machine code, whenever he is able to do so. In particular, for the routines occurring in the standard-prelude (and also in the library-preludes, see 0.10.7), he can generate efficient machine code himself, taking advantage of every specific machine feature (most machines will have single commands for addition and integer division and, unless he is a maniac, your implementor will not follow the routines in the standard-prelude to the letter).

0.10.4. Operations on **compl**ex operands

```
mode compl = struct ( real re , im ) ;
```

[R 10.2.3.7]

```
op   ⊥   = (real a, b) compl:  ( a , b ) ;
op   re  = (compl a) real:     re of a ;
op   im  = (compl a) real:     im of a ;
op  abs  = (compl a) real:     sqrt ( re a ↑ 2 + im a ↑ 2 ) ;
op  arg  = (compl a) real:
                   if real re = re a , im = im a ;
                      re ≠ 0 or im ≠ 0
                   then   if abs re > abs im
                          then arctan ( im/re ) +
                               pi/2 × ( im < 0 | sign re – 1 | 1 – Sign re )
                          else –arctan ( re/im ) + pi/2 × sign im
                          fi
                   fi ;

op conj  = (compl a) compl:    re a ⊥ – im a ;
op  .=   = (compl a, b) bool:  re a = re b ∧ im a = im b ;
op   ≠   = (compl a, b) bool:  ¬( a = b ) ;
op   +   = (compl a) compl:    a ;
op   –   = (compl a) compl:    – re a ⊥ – im a ;
op   +   = (compl a, b) compl: ( re a + re b ) ⊥ ( im a + im b ) ;
op   –   = (compl a, b) compl: ( re a – re b ) ⊥ ( im a – im b ) ;
op   ×   = (compl a, b) compl: ( re a × re b – im a × im b ) ⊥
                               ( re a × im b + im a × re b ) ;
```

```
op   /   =(compl a, b) compl: (   real d = re( b × conj b );
                                  compl n = a × conj b;
                                    ( re n / d ) ⊥ ( im n / d ) );

op   ↑   =(compl a, int b) compl:
                                  ( compl p := 1;
                                    to abs b do p := p × a od;
                                    ( b ⩾ 0 | p | 1/p )        );
```

We could subjoin another operator to this set, which the authors seem to have forgotten: the monadic **i** (or ⊥ which is, however, a less appropriate representation in this case):

```
(D10)  op i = (int a) compl: ( 0 , a );
       op i = (real a) compl: ( 0 , a );
```

Instead of $x \perp y$ (or x **i** y) you could then also write $x +$ **i** y, which is closer still to the usual mathematical notation.

0.10.5. Operations combined with assignations

[R 10.2.3.11]

```
op +:=  = (ref real a , real b) ref real: a := a + b ; ¢ or plusab  ¢
op −:=  = (ref real a , real b) ref real: a := a − b ; ¢ or minusab ¢
op ×:=  = (ref real a , real b) ref real: a := a × b ; ¢ or timesab ¢
op /:=  = (ref real a , real b) ref real: a := a / b ; ¢ or divab   ¢
```

The first formal-parameter has to be **ref**, because we want to assign to it. The value returned has also been declared to be **ref**, and some consequences of this are shown in 6.3. These operators are declared for all arithmetic operands (**int** , **real** and **compl**).

0.10.6. Operations on **strings**

The standard mode **string** is defined as follows:

```
[R 10.2.2]   mode string = flex [1 : 0] char ;
```

The **flex** in front of the $[1 : 0]$ means that the bounds of a **string** may be reset by assignation (compare 5.5.4.1) – i.e. a **string** is a multiple which is allowed to "breathe" – initially a **string** is empty, which is expressed by $[1 : 0]$, the upper-bound being less than the lower-bound.

It is instructive to unravel the operations on **strings**. In doing so we meet

the at-symbol @ (or **at**) which arranges for the bounds of the actual **strings** to be considered (by "sliding" them) to have a lower-bound *1*.

[R 10.2.3.10]

```
op < =( string a , b ) bool:
        begin   int m = upb a[@1] ; n = upb b[@1] ; int c := 0 ;
                for i to if m < n then m else n fi
                while ( c := abs a[@1][i] - abs b[@1][i] ) = 0
                do skip od ;
                if c = 0 then m < n and n > 0 else c < 0 fi
        end ;
op ≤ =( string a , b ) bool: ¬( b < a )
op = =( string a , b ) bool: a ≤ b and b ≤ a
op ≠ =( string a , b ) bool: ¬ ( a = b )
op ≥ =( string a , b ) bool: b ≤ a
op > =( string a , b ) bool: b < a
op + =( string a , b ) string:
        begin int m = if int la = upb a[@1]; la < 0 then 0 else la fi ,
                      n = if int lb = upb b[@1]; lb < 0 then 0 else lb fi ;
              [1:m + n] char c ;
              c[1:m] := a[@1]; c[m + 1:m + n] := b[@1];
              c
        end ;
op x =( string a , int b ) string: ( string c ; to b do c := c + a od ; c ) ;
op x =( int a , string b ) string: b x a
```

0.10.7. The library prelude

The operation-declarations considered thus far belong to the standard-prelude of the language, i.e. they are built in. You will also find in the standard-prelude all environment enquiries and all declarations for formatless and formatted input and output (which therefore are also built into this language).

Nothing, however, prevents you from subjoining to this standard-prelude a set of home made declarations. Of course you are free to declare such new things in your own particular-program; but, as soon as you want to apply them in several programs, or you want to enable others to use them, or you have reason to expect that more efficiency may be acquired, then you can go to your implementor and ask him to make the whole set an as efficient as

possible extension of the standard-prelude, i.e. a 'library-prelude'. In that way an arbitrary number of problem oriented dialects may be defined. For some possible examples see 8.4 and 8.5.

The possibility of subjoining library-preludes to the standard-prelude contributes in no small measure to the flexibility of this language and this is one of the basic concepts of ALGOL 68.

You should, therefore, never accept an implementation in which library-preludes cannot be coded efficiently or in which the attachment of one or more library-prelude(s) to the standard-prelude cannot easily be done.

0.11. **bits** and **bytes**, **longs** and **shorts**

On most modern computers you will find, if not in the hardware then in the standard software, provision for the manipulation of single bits in a machine word, of parts of machine words (byte-addressing) and also for double and maybe even multilength arithmetic.

In a concrete computer, all instances of values of all modes will be stored as bit-patterns. Whether a specific bit-pattern may correspond to a value of a specific mode or not is mainly a matter of how the standard software may interpret that piece of binary information. That is to say the interpretation of bit-patterns and also the arrangement, size and structure of their locations in the memory is almost entirely a matter of software.

In this language, the possibility of considering (a part of) a machine word as a mere sequence of bits is reflected in the mode **bits** (0.11.1), the particular interpretation of a "byte" as a **char** (i.e. the interpretation of one or more machine words as representing a fixed sequence of **chars**) is reflected in the mode **bytes** (0.11.1) and the possibility of multilength arithmetic is reflected in the **long - - - long** modes (0.11.2). To what extent available hardware features will be used for these further modes and to what extent (and how) they will be simulated by software provisions is entirely a matter of implementation.

0.11.1. The modes **bits** and **bytes**

Both **bits** and **bytes** are defined to enable the programmer to take advantage of certain (hardware-) features of the machine on which the language is implemented. A **bits** will be something pretty close to a machine word and the environment enquiry *bits width* is then the wordlength. A **bytes** may be a memory unit in which a certain number of characters can efficiently be stored; a **bytes** may be considered as a string of fixed length *bytes width.*

Operators are defined for **bits** and **bytes** reflecting the most current machine operations.

For **bits** we have = and ≠ to compare them, and ∨ (disjunction), ∧ (conjunction), **shl** (shift left), **shr** (shift right), **abs** (from **bits** to **int**), **bin** (from **int** to **bits**), an operator **elem** (selects a certain bit from a **bits**) and some others.

For **bytes** we have the comparison operators as for **strings**, **elem** as for **bits** and a transfer from **string** to **bytes**.

(D11) **bits** *t* ; **bytes** *r* ;

declares *t* to be a **bits** variable, and *r* to be a **bytes** variable.

For **bits** we have a separate denotation, consisting of a sequence of digits *1* (the equivalent of **true**) and *0* (the equivalent of **false**), preceded by *2 r* to indicate that the sequence following is to be understood as a number in binary representation (radix 2). If we assign:

(E49) *t* := *2 r 1011100100001*

and bits width is, say, *32* then t̂ refers to a machineword (a **bits**):

2 r 00000000000000000001011100100001

The value of **abs** *t* is now *5921* (conversely, the value of *t* is **bin** *5921*).

If we now assign:

t := *t* ∧ *2 r 111111*

then the value of *t* becomes *2 r 100001.* Then, after the assignation:

t := *t* **shl** *3*

the value of *t* is *2 r 100001000,* so that *29* **elem** *t,* i.e. *(bitswidth−3)* **elem** *t,* is **true**, but *30* **elem** *t* is **false.**

If we want to consider the **bytes** *r* as a **string**, then we may apply a cast, e.g.

s **plusab string** *(r)*

Just to demonstrate **bits** and **bytes**, we consider the following example:

(E50)
```
proc compose string =
  ( bits select , ref [1: bits width] bytes phrase ) string:
    ( string s ; ¢ initialization is not necessary, because
                   the flexible bounds are set to 0 and 1 at
                   the declaration (see D3 in 0.4.3) ¢
      for i to bitswidth
      do if i elem select then s plusab string ( phrase[i] ) fi od ;
      s ) ;
```

0.11.2. The **long** and **short** modes

The prefices **long** and **short** play a role in the creation of new modes in much the same way as [ ] , **struct** , **proc** and **ref** do; the **long** or **short**, however, may stand only in front of **int** , **real** , **compl** , **bits** and **bytes** and of all long and short modes derived from these.

In the standard-prelude you will find environment enquiries such as:

int *int lengths* = **c** the number of different lengths of integers **c** ;
int *int shorths* = **c** the number of different shorths of integers **c** ;

stating to what extent the long and short feature is implemented for **int**s, and correspondingly for **real**s, **bits** and **bytes**.

Now, if we declare, for instance:

long long long long long long long int *iiiiiiiint* ;

but *int lengths* = *3* , then the value of our *iiiiiiiint* will be treated as if it had been declared:

long long int *iiiiiiiint* ;

Hence, *int lengths* = *3* means that your implementor will distinguish only three kinds of integral values: **int** , **long int** and **long long int** . The same applies to **real** (and, consequently, to **compl**), **bits** and **bytes**. The number of **long**s characterizes the degree of discrimination with which the value is kept in the computer.

In the language the prefices **long** and **short** also turn up in denotations:

iiiiiiiint := **long long long long long long long** *0*

In the standard-prelude you will also find the environment enquiries:

long int *long max int* = **c** the largest long integral value **c** ;
short int *short max int* = **c** the largest short integral value **c** ;
long long int *long long max int* = **c** the largest long long integral value **c** ;
etc.

short real *short max real* = c the largest short real value c ;
short real *short small real* = c the smallest short real value which can be usefully compared with **short** *1.0* c ;
long real *long max real* = c the largest long real value c ;
long real *long small real* = c the smallest long real value which can be usefully compared with **long** *1.0* c ;

etc.

For the arithmetic modes (**int** , **real** , **compl** and their **longs** and **shorts**) we have a monadic-operator **leng** which makes the operand one **longer**, and a monadic-operator **shorten** which takes away one **long** or adds one **short**. There is no automatic transfer between different **longs** and **shorts** of the same basic mode.

For example:

```
(E51)  proc inprod = ( ref [ ] real a , b ) real:
           ( long real value := long 0.0 ;
             for i
             from ( lwb a < lwb b | lwb b | lwb a )
                to ( upb a > upb b | upb b | upb a )
               do value +:= leng a[i] × leng b[i] od ;
             shorten value ) ;
```

If, in a call of E51, the bounds of the actual rows are not equal, then the routine will compute the innerproduct as if the rows had been supplemented with zero elements until their bounds matched.

0.12. Unions

United modes (**unions** for short) are brought into existence to enable the programmer to specify locations in which values of different modes can be stored, and to dispose of the names which refer to such accommodating locations. In particular, with the aid of **unions** you can define routines which accept actual-parameters and (or) return a value of one of several possible modes.

The mode-declaration:

```
(E52)  mode strint = union ( string , int ) ;
```

declares **strint** to be a new mode encompassing both the mode **string** and the mode **int**. It is important to know that this does not define a new kind of value; a new mode has been declared. The values in this mode, however, are still **strings** or **ints** (see also Section 1.6.1).

The variable-declaration:

(E53) **loc strint** *year ;*

ascribes to *year* the name of either a **string** or an **int.** Thus year is a **ref strint** identifier.

A **ref strint** is, most certainly, a new kind of value (i.e. it is neither a **ref string**, nor a **ref int**), it is a **ref union (string , int)**, but it does not refer to a "**strint**" because there is no such thing.

You may now assign to *year*:

(E54) *year* := *"1968"*

as on another occasion:

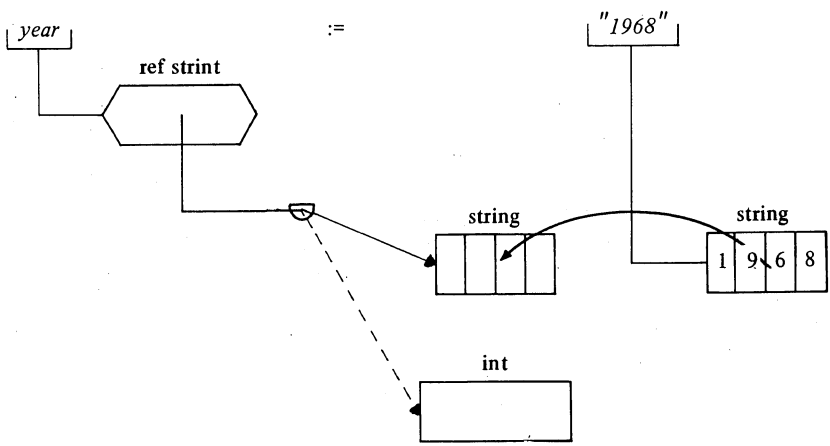
(E54*) *year* := *1968*

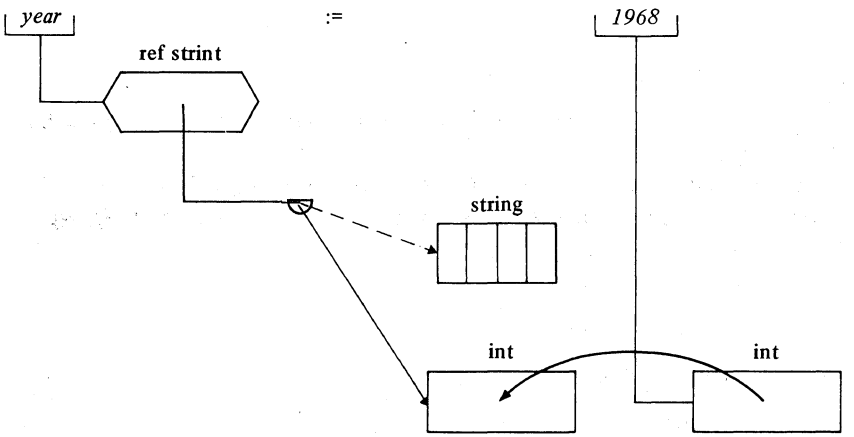
Observe, that in the context of:

(E55) **string** *text* , **int** *numb ;*

neither the assignation *text* := *1968* , nor the assignation *numb* := *"1968"*, is allowed.

The assignations E54 and E54* may be depicted as follows:





Once you have declared a united variable, you may want to ask the mode of the value assigned to it. It may be clear that this requires mode checking at run time. However, this is in fact the only situation where, in this language, run time mode checking is inevitable and for no other reason than that the programmer has explicitly asked for it.

In the context of E52 - - - E55 you may write:

(E56)
```
case year
   in ( string ) : true ,
      ( int ) :    false
esac
```

This unit is a 'conformity-clause'. It yields upon elaboration the value **true** if *year* refers to a **string** (which is the case, for instance, after the assignation E54): otherwise its value is **false** (which is the case after E54*).

The expressions *text* := *year* and *numb* := *year* are not correct assignations in this language, not even when the modes conform. Therefore we have a slightly more extended form of the conformity-clause:

(E57)
```
case year
   in ( string s ) : text := s ,
      ( int i ) :    numb := i
esac
```

If we want to know which of the two assignations has been elaborated we might declare:

```
(E58)   proc deliver = ( ref string t , ref int n , strint tn ) bool:
            case tn
              in ( string s ): ( t := s ; true ) ,
                 ( int i     ): ( n := i ; false )
            esac ;
        deliver ( text , numb , year )
```

The call *deliver (text, numb, year)* delivers the actual value of *year* to the right destination. The value of *deliver* is **true** or **false** depending upon the mood of the year.

An example of a procedure yielding a value of one of two possible modes may be (in the context of the declarations E23* and E23**, see 0.7.3):

```
(E59*)  mode intreal  = union ( int , real ) ;
```

```
(E59)   proc factorial = ( int n ) intreal:
                              if n > nmaxfac
                            then faclarge (n)
                                 ¢ in which case a real value is yielded ¢
                             else int f := 1 ;
                                 for i from 2 to n do f x:= i od ; f
                                 ¢ in which case an int value is yielded ¢
                              fi ;
```

The difference with E23 is that any call of the there declared **proc(int)real** *fac* yields a **real** (the **int** computed in the else-part is widened to **real** because such is required), so that you can not know whether the value returned was an exact factorial or not. A call of the **proc(int)intreal** *factorial,* on the contrary, yields an **int** or a **real** and you can find out which of the two was the case.

Beware, however, of a pitfall.

You cannot assign:

```
y := factorial (m)
```

nor:

```
i := factorial (m)
```

because an "**intreal**" can neither be assigned to a **ref real** nor to a **ref int.**

The way to achieve this, of course, is:

```
case factorial (m)
  in (real r): y := r ,
     (int n): i := n
esac
```

As a final example consider:

(E60)
```
mode numb = union ( int , real , compl ) ;
proc gamma = ( numb u ) numb:
    case u
      in ( int i ):
          if i > 0
       then case i
              in 1, 1 , 2 , 6 , 24 , 120 , 720 , 5040 ,
                 40320 , 362880 , 3628800 , 39916800 ,
                 479001600
            out faclarge (i-1)
           esac
      fi ,
      ( real r ): c algorithm for the gamma function with
                  a real argument r , yielding a real value c ,
      ( compl c ): c algorithm for the gamma function with
                   a compl argument c , yielding a compl value c
esac
```

0.13. Local and global generators, stack and heap

We know that by a variable-declaration like

loc real *x* ;

a **real** value is generated on the stack. To put it more precisely:

The "local-generator" **loc real** generates on the stack a new **real** value (its "side-effect" so to speak) and it yields upon elaboration the **ref real** name x̂ which is then ascribed to the identifier *x*.

A less concise way to formulate this happening is:

ref real *x* = **loc real** ;

in which we meet the local-generator as a unit.

An interesting example of the use of a local-generator outside a variable-declaration or an identity-declaration is given by the following phrases in which a triangular matrix is generated:

(E61)
```
loc [1: n] ref [ ] real triang ;
for i to n
do triang [i] := loc [1: i] real od ;
```

Observe that *triang* is declared to yield a singly subscripted row of names: *triang*[*i*] yields the name of a [ ] **real**; *triang*[*i,j*] is undefined. If you want to access the [*i,j*]th element of *triang*, you must write:

triang [*i*] [*j*]

You then access the *j*th element of the row referred to by the *i*th element of *triang* (for further discussion, see 5.7.2.E10).

The generator **loc real** is a local-generator because the **real** value it generates ceases to exist as soon as the range to which the value was **loc**al is completed (i.e. as soon as we leave that range). Not only the relation between the identifier *x* and the **ref real** ascribed to it ceases to hold, but also the **real** value it referred to vanishes as the stack contracts. We must know that a serial-clause is a local range when it contains at least one declaration – therefore the clause between **do** and **od** in E61 was not a local range and we could take the **ref** [ ] **real** value with us outside the **do** and **od.**

Now, what can we do when we want to take a value and its name outside a local range, for example when the serial-clause between **do** and **od** contains a declaration? Or, to put it differently, how can a location (a box) survive a contracting stack?

The solution in this language is the presupposition of the "heap": another storage allocation regime besides the stack, in which values may be generated which remain there as long as some name refers to them.

By a variable-declaration like:

heap real *hx ;*

or an identity-declaration:

ref real *hx* = **heap real** *;*

a **real** value is again generated, and the global, or "heap", generator **heap real** yields its name.

For example:

(E62)
```
loc [1:n] ref [ ] real zigzag ;
for i to n
do loc int length ; read (length) ;
   zigzag [i] := heap [1:length] real
od ;
```

Here the local-generator **loc**[*1 : length*] **real** would not work, because the stack will contract at the completion of the serial-clause between **do** and **od** which now is a local range because it contains the declaration **loc int** *length*;.

For an example of a heap variable-declaration consider:

(E63)
```
mode record = struct (string name, int date of birth,
                      real value, ref record next),
mode society = struct (ref record first, last);
loc society high;
```

Now assume a large file of records defining potential members for the *high* society. We want a procedure *try* to scan that file and try every record for acceptability on the ground of age and minimal value required. The procedure *try* has to yield **false** when the record under consideration is not acceptable; if it is (the rare cases) it must generate a seat in the **society** and yield **true**. A procedure always defines a local range, so we must apply a heap-generator:

```
proc try = (int date, real minimum) bool:
  if heap record new;
    read ( (name of new, date of birth of new,
            value of new ) );
    next of new := nil;
    date of birth of new < date and value of new > minimum
  then last of high := next of last of high := new;
       ¢ in which case new survives ¢
       true
  else false ¢ in which case new is thrown away ¢
  fi;
```

Now let us generate the founder member:

```
first of high := last of high :=
    heap record := ("Methuselah", 0, 1000.0, nil);
```

and examine the first hundred candidates, who are apparently required to be no more than 50 years younger than *Methuselah*, and to be not much lower in value than their predecessors.

```
to 100
do   if try ( (date of birth of first of high ) + 50,
              .95 x (value of last of high) )
     then print ( (name of last of high, newline) )
       fi
od
```

0.14. What to do next

The remainder of this Introduction contains, in a two-dimensional way, eight (or seven) chapters (please now refer to the table of contents). The eight

horizontal chapters are:

1. BASIC CONCEPTS
2. DECLARATIONS
3. CLAUSES
4. ROUTINES
5. UNITS
6. STANDARD PRELUDE
7. TRANSPUT
8. EXAMPLES

The seven vertical chapters are:

.1 FUNDAMENTALS
.2 PROCEDURES AND NAMES
.3 OPERATIONS
.4 STRUCTURES
.5 MULTIPLE VALUES
.6 UNIONS
.7 DISTINCTIVE FEATURES

Thus the horizontal chapters are subdivided into seven sections ".1" through ".7". Likewise, the vertical chapters are subdivided into eight sections "1." through "8.".

You may read row-wise:

for *i* **to** *8*
do for *j* **to** *7* **do** *elaborate section* [*i,j*] **od od**

or you may read column-wise:

for *i* **to** *7*
do for *j* **to** *8* **do** *elaborate section* [*j,i*] **od od**

The latter (vertical) route is the more didactic one, for those who wish to learn the language. The horizontal one (along which this Introduction has been bound) is more appropriate for those who wish to survey the essential principles of the language as a whole. In particular, the first chapter on BASIC CONCEPTS is a survey of the main part of the basis on which the language was "orthogonally designed" [R 0.1.2]; i.e. the generalized concept of "mode", and all its consequences.

If you are now in some doubt as to which route is for you, then take our suggestion – read horizontally in Chapter 1 until you find the waters beginning to get a little deep: then return to 2.1 and read by the vertical route thereafter.

1. BASIC CONCEPTS

1.1.Fundamentals

You write or read a 'particular-program' which is embedded in an environment consisting of the 'standard-prelude' (and '-postlude') and a 'library-prelude'.

The standard-prelude is a comprehensive selection of features, generally accepted as a standard environment for a modern programming language. A library-prelude is a continuation of the standard-prelude. It may contain more specific features you would like to have at your disposal in certain classes of problem. The implementation is supposed to cater for some provision which enables you to subjoin one or, ideally, a selection of library-preludes.

The whole constitutes a 'program'.

1.1.1.Objects

A program may be parsed into a tree of "constructs", such as identifiers, denotations, formulas, procedures, declarations, clauses, etc. These are all classified as "external objects", since they comprise the written, external form of the program. A construct (or for that matter the whole program) may be "elaborated" by a "computer" (be it a human being or an automaton), whereupon a defined sequence of "actions" takes place and, upon the completion of these, a "value" is "yielded". With some constructs, the sequence of actions is the prime reason for the elaboration. With others, it is the value yielded which is important–indeed, in the case of identifiers and other such "indicators" there are no actions at all, and the value yielded is simply that which had previously been "ascribed" to that indicator in a declaration.

A "value" (or, more precisely, an "instance" of that value) must presumably be kept somewhere, either in the human's mind or in the memory of his automaton. A value is therefore classified as an "internal object". Each internal object (in the sequel often "object" for short) has three relevant attributes:

1) it is of some "mode",
2) it is a particular instance of a value of that mode,
3) it has some location.

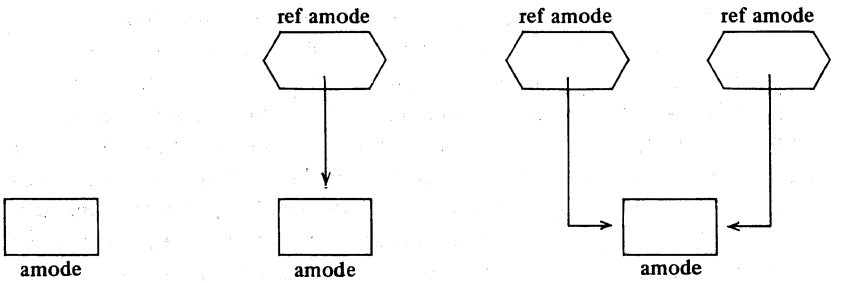
1) The mode specifies how the object is built up from basic material (bits, or

the little grey cells in your brain) and to what kind of entities (numbers, characters, records, names, etc.) it is related. Partly this is a matter of implementation (the building of a real number for example), partly this construction may be specified by the program in terms of modes already defined (for example the building of a complex number as an ordered pair of real numbers). In the program text a mode may be indicated by a bold faced word, which is then to be considered as one indivisible symbol (e.g. **amode**). Such a mode-indication may be regarded as the badge of some class of values.
2) Some modes define but a few values (e.g. a **bool** can only be **true** or **false**), some quite a lot (e.g. **int** and **real**), some in principle an infinity (e.g. **string**); but there may be any number of instances of any particular value within the automaton, and such an "instance of a value" of some mode is an internal object.
3) An object is to be found somewhere, and this somewhere is its location (its address in the memory). The physical address is none of your business, but in many cases you will certainly want to have control of that location (for example you may wish to supersede the object by another instance of a value, or to enter its location in some chain), that is you may wish to "refer to" that object. As far as the location is concerned, there are two possibilities:
3a) The internal object was, in some previous declaration, "ascribed" to an external object (specifically, an indicator). Now you can always obtain the internal object by elaborating that indicator and inspecting its yield, but you still have no control over its location (because it may well be concealed in the object code) and you have to take it as it arises in the elaboration of the program, in which it is a "constant".
3b) The internal object is "referred to" by a second internal object (specifically, a "name"). In that case, its location is at your disposal in the form of that name. This gives you the right to supersede it, and it is therefore not a constant but a "variable". A name is an object of another (!) related mode—a 'reference to' (**ref**) mode. A **ref amode** object (a name) refers to an **amode** object.

You may visualize the interrelation of the concepts mode, value and name (which are of fundamental importance in this language) by drawing boxes in a "paper computer".

Boxes of the same shape then represent internal objects of the same mode. Each box holds an instance of a value (not necessarily different from the instance in another box). Names may come into the picture by drawing boxes of another shape, holding those names.

The relation "to refer" between two internal objects is depicted by an arrow pointing from the name to the object referred to by that name:



We shall presently show you how actions may be depicted in our paper computer. They will always achieve the effects defined by the hypothetical computer in the Report (but sometimes by a different method).

1.1.2.Identifiers

In order to discuss internal objects, we need 'identifiers'. An identifier is a sequence of letters and digits with a leading letter, like *marllyn* and unlike *1marlyn.*

The meaning of an identifier is defined in an 'identifier-declaration', of which there are two sorts—the variable-declaration (1.1.2.1) and the identity-declaration (1.2.2). (Alternatively, a label-identifier is defined as such when it occurs as a label in the program text.)

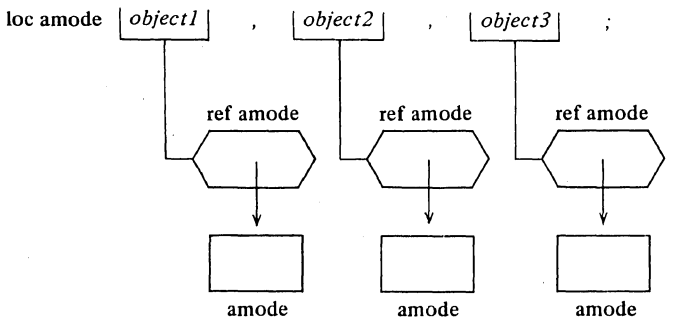
1.1.2.1.Variable declarations

By declaring:

(E1) **loc amode** *object1 , object2 , object3 ;*

three internal objects are generated in the memory, each of them being an **amode.** Three names (**ref amode** objects) referring to the **amode** objects generated are then ascribed to the three identifiers. Therefore they are known as **ref amode** identifiers and the generated **amode** objects are 'variables'. In a picture: three boxes come about, each of them holding the name (location) of another box of the mode **amode.**

Now, *object1 , object2* and *object3* are external objects, being constituents of the program. By the variable-declaration E1 an internal object is ascribed to each of them. The relationship between an identifier and the object ascribed to it cannot be changed; and the object ascribed (in this case a name) cannot be changed.



We preferably draw external objects at the top of the diagrams in our paper computer (to separate them from the boxes, which are internal) and, if the elaboration of one of the external objects yields one of the internal ones (at the instant of time under discussion), we depict this by a line joining the two. The diagram above shows that, since **ref amode** objects have been ascribed to the identifiers *object1, object2* and *object3*, the yields of these identifiers will henceforth be the three objects drawn immediately below them. These objects are constant (the top row of boxes in our diagrams will generally depict constants). The boxes in the bottom row, however, are variable (we can see that this is so, because we can see that there exist names referring to them).

Thus, *object1*, for example, yields a constant name (mode **ref amode**) which refers to a variable **amode**. In the sequel we shall abbreviate this by saying, simply, that *object1* refers to that variable.

This is nothing new. In many other programming languages the proper relation between an identifier and its variable value is exactly the same, although perhaps you were never aware of it.

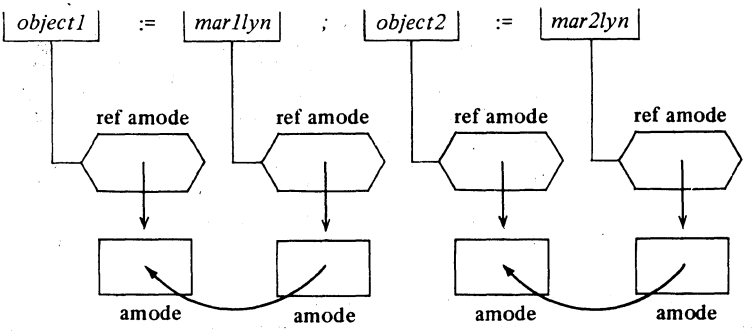
1.1.2.2. Assignation, collateral elaboration

If *mar1lyn* and *mar2lyn* are likewise declared to be **ref amode** identifiers (consequently yielding **ref amode** objects), then by assigning:

(E2) *object1* := *mar1lyn* ; *object2* := *mar2lyn*

the object referred to by *object1* (*object2*) is "superseded" by an instance of the value referred to by *mar1lyn* (*mar2lyn*). Nothing happens to the names ascribed to the identifiers. The value referred to by the LHS (Left Hand Side) becomes a copy of the value referred to by the RHS (Right Hand Side). The copy-action "to assign" is depicted below by a bowed arrow originating from

the object to be copied and pointing into the location of the copy:



Again this is nothing new. In most other programming languages the process of "assignment to a variable" takes place in exactly the same way.

In an assignation the LHS (the 'destination') and the RHS (the 'source') are elaborated "collaterally", i.e. there is no prescribed order for the actions of getting the name (the **ref amode**) in the LHS and getting the value (the **amode**) in the RHS. Consequently, if these two actions should happen to have any side effect upon each other (in the case of more involved assignations this could occur), then the result of the assignations is "undefined" (i.e. not defined by the Report alone).

In E1 also we met a collateral elaboration. In fact E1 involves three variable-declarations, the declaration of *object1*, of *object2* and of *object3*; and these three declarations are elaborated collaterally.

1.1.3.Phrases, serial and collateral elaboration

The piece of program text:

(E3)
```
amode object1, object2, object3 ;
      object1 := mar1lyn ;
      object2 := mar2lyn
```

is a simple case of a 'serial-clause'. The "constituents" separated by semicolons are 'phrases' which may be either 'declarations' or 'units'. The semicolons represent the 'go-on-symbol'. The action defined by the phrase following a go-on-symbol begins after the completion of the action defined by the phrase preceding it.

As we have already pointed out, the variable-declaration:

loc amode *object1* , *object2* , *object3*

is a collateral-declaration. In fact it is a "contraction" of the phrase:

loc amode *object1* , **loc amode** *object2* , **loc amode** *object3*

Tne commas represent the 'and-also-symbol' and achieve the collateral creation of the objects.

The symbol **loc** expresses the act of generation of the variable (1.2.2.3), but it is optional in this context and is frequently omitted.

Besides collateral-declarations we may have serial-declarations, for example:

amode *object1* ; **amode** *object2* ; **amode** *object3*

and the go-on-symbols achieve serial creation of the objects (one by one). Eventual side effects (which in the case of more involved declarations could occur) now act precisely as prescribed by the order of elaboration thus defined.

Enclosing a serial-clause between "*(*" and "*)*" or "**begin**" and "**end**", we obtain a 'closed-clause':

(E4) *(* **amode** *object1, object2, object3 ;*
object1 := *mar1lyn ; object2* := *mar2lyn ;*
XXXXX *)*

By "XXXXX" we denote here and in the sequel an arbitrary constituent valid in the context.

By enclosing a serial-clause, a 'range' is demarcated (see also 3.1.5). A range has much in common with what in some other programming languages is known as a block: in particular it defines the "scope" of the values (names) created by the declarations within it.

A unit always yields a value of some mode (which may, however, be **void** if no value is actually required). For example, the unit:

object1 := *mar1lyn*

yields the value yielded by its LHS (and not, as you might have expected, the value yielded by its RHS), that is the **ref amode** object yielded by *object1*. An assignation yields the name in its LHS.

Correspondingly, a serial-clause yields a value of some mode, namely the value yielded by the unit which completes its action.

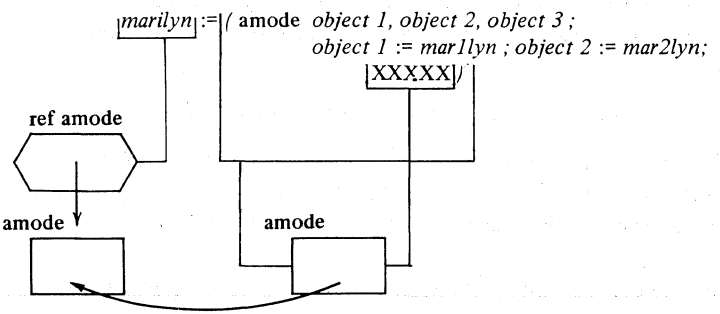
The same applies to closed-clauses.

For example, if XXXXX in E4 yields by elaboration an **amode** object,

(and *marilyn* is declared to be a **ref amode** identifier) then:

(E5) *marilyn* := (**amode** *object1, object2, object3 ;*
object1 := *mar1lyn ; object2* := *mar2lyn ;*
XXXXX)

is a perfectly sound assignation. It assigns the value yielded by E4, which is the value yielded by XXXXX, to *marilyn.*



1.1.4.Routines

An internal object of fundamental importance is the "routine", which is the internal equivalent to the sequence of symbols which comprises a 'routine-text'. A routine may or may not have formal-parameters, and may or may not return a value of some mode. A routine-text is rather close to what in some other programming languages is known as "a procedure-heading with procedure-body".

In this language a routine may be ascribed, not only to an identifier, but also to an 'operator'.

A routine may be "called":
a) in a 'formula' by means of a operator yielding the routine,
b) or else by means of an identifier yielding (or, which may also be the case, referring to) the routine, i.e. by a 'call'.

By declaring:

(E6) **op** ◊ = (**amode** *formal1, formal2*) **amode**: XXXXX ;

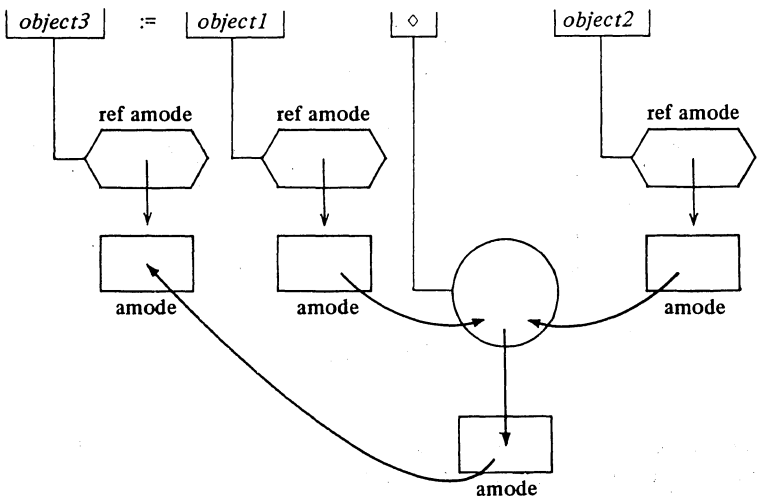
the symbol "◊" is declared to be an operator, and the routine yielded by the routine-text on the RHS is ascribed to it. In this routine-text, XXXXX is some unit defining the action, using the formal-parameters *formal1* and

formal2. The **amode**: preceding it expresses that the routine is to return an **amode** value.

By virtue of this declaration, the unit:

(E7) *object3* := *object1* ◊ *object2*

results in:



The routine ascribed to ◊ is depicted by a circle.

In E7 again, the LHS and the RHS are elaborated collaterally. The RHS is a formula in which both 'operands', *object1* and *object2*, are in their turn elaborated collaterally (corresponding to the fact that the formal-parameters in E6 are separated by a comma). Formulas are described more fully in 1.3 and 5.1.3.

By declaring:

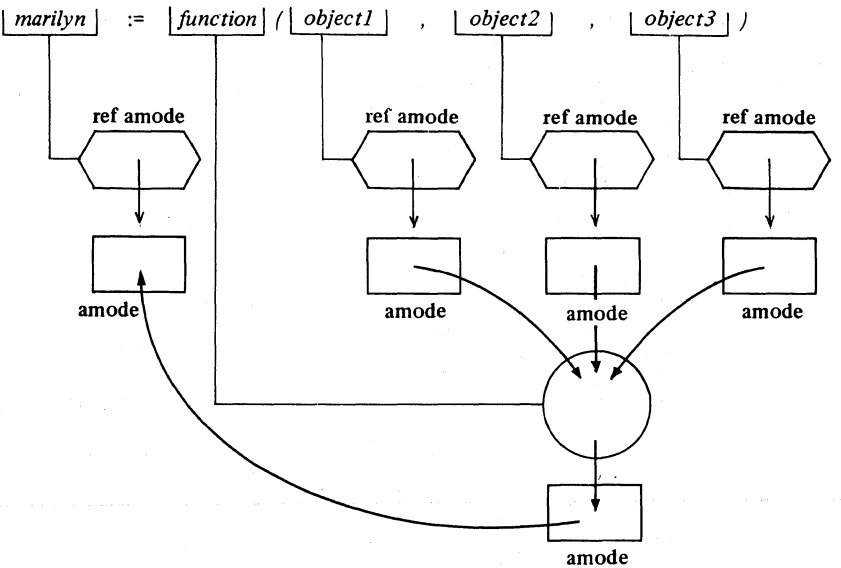
(E8) **proc** *function* = *(***amode** *formal1, formal2, formal3)* **amode**: XXXXX *;*

the identifier *function* is declared, and the routine yielded by the RHS is ascribed to it. In this routine-text, XXXXX is some unit defining the action, using the formal-parameters *formal1, formal2* and *formal3.*

By virtue of this declaration, the unit:

(E9) *marilyn* := *function (object1 , object2 , object3)*

results in:



Again, the LHS and the RHS are elaborated collaterally, as are the three actual-parameters *object1, object2* and *object3* on the RHS, by virtue of the commas between the formal-parameters in the routine.

1.1.5. Defining and applied occurrences

Consider the assignation:

(E5*) *marilyn* := (**amode** *object1, object2, object3 ;*
object1 := *mar1lyn ;*
object2 := *mar2lyn ;*
object3 := *object1* ◊ *object2 ;*
function (object1, object2, object3))

To *marilyn* is assigned the value of a closed-clause. The outmost "*(*" and "*)*" enclose a serial-clause consisting of a collateral-declaration, followed by three consecutive assignations, in the last of which the RHS is a formula, followed by the call of a procedure returning an **amode** value. This value returned by *function* is now the value yielded by the closed-clause and consequently the value assigned to *marilyn.*

It might be worth your while to try to visualize the elaboration of E5* in one picture, in the same way as we did for the separate constituent phrases. You will meet then several "occurrences" of the identifiers *object1, object2* and *object3*, the first of which are the "defining occurrences" in the declaration **amode** *object1, object2, object3.* All other occurrences of these identifiers are "applied occurrences". Here we have a relation between two external objects, the technical term for this relation is "to identify": the second occurrence of *object1* identifies its defining occurrence. You might depict this relation by an arrow pointing from the applied occurrence to its defining occurrence.

1.1.6.Coercion

Every external object has, independent from the particular syntactic position in which it stands, an "a priori" value of some a priori mode. In order to make it fit its particular context, the external object may be "coerced", that is "forced to yield a value of another mode": its "a posteriori" mode and a posteriori value.

For example, the a priori mode of *marllyn* in E2 is **ref amode** (by virtue of its declaration), and thus its a priori value is a name (of an **amode** object). Now, by the assignation process as described in 1.1.2.2 ("getting the value" on the RHS), the a posteriori mode of *marllyn* must here be **amode** (we want the **amode** value referred to, and not the name). In this particular context *marllyn* must be "dereferenced", which is one of the six basic coercions in this language.

Observe that *object1* in E2 (the LHS of this assignation) is not dereferenced, but in E7 (in the syntactic position of an **amode** operand) as in E9 (in the syntactic position of an **amode** actual-parameter) it is.

Another example of coercion is "widening", implicit change from mode **int** to mode **real**, mode **real** to mode **compl**, and some others. Once you know what the term is about, you will find quite a lot of coercions in other programming languages, although perhaps they are not always so well defined if at all [see R6].

In a language in which the basic concepts are extended as far as possible, one must inevitably be very clear and precise on the subject of the actions concealed in the language. It is dangerous to presume actions to be implicit without stating exactly why, where and how. Of course, one could have supplied a certain number of operators, expressing explicitly the desired transitions from a priori to a posteriori mode and value. But then you would have been coerced into writing *object1* := **DEREFERENCE** *marllyn* and

object3 := **DEREFERENCE** *object1* ◊ **DEREFERENCE** *object2* or some such, and very soon you would encounter much more miserable constructions (see 5.1.0.2).

In ALGOL 68 at least six rather offensive monadic operators of this kind are incorporated in the syntax, being implied by the syntactic position of the external object to which they otherwise ought to have been explicitly applied. This, indeed, complicates in no small measure the syntax. However, once you have mastered that part of it [the whole of R 6], you will appreciate that the burden is taken away from your shoulders. Apart from that, coercion has one great charm: it does exactly for you what you want, but could easily have forgotten. You will feel happy that you can write $x := i$ instead of x := **WIDENTOREAL DEREFERENCE** i. For a systematic discussion of all the coercions, see Chapter 5.

Vertical readers, please turn to 2.1.

1.2.Names and declarers

1.2.1.Ascription and assignation

We have already introduced variables to you. They are internal objects and new values can be "assigned" to them at any time and as often as you like. This is natural, for why should it be called a "variable" if you cannot vary it?

A constant is, of course, quite a different thing. Obviously it cannot be varied: it is an external object which is given a value once and for all, and the process of giving it its once only value is termed "ascription". This process can be brought about in identity-declarations (1.2.2) and in procedure calls (1.2.3.2.1). It also arises in variable-declarations (as we have seen in 1.1.2.1), but for a different reason. Behind each **amode** variable there hides a constant **ref amode** name (it has to be a constant name, for otherwise you might lose the variable). It is this constant name which is ascribed to the **ref amode** identifier in a variable-declaration.

Often, when creating a new object, you have the choice of declaring it as a constant or as a variable. Which should you do? (You might say that it does not matter, since it will work either way, but this is a dangerous belief.) Take our advice. If you do not intend to vary it again (at least within the lifetime of the relevant range–see 2.2.1), declare it as a constant and ascribe its only value to it. Variables are dangerous objects, and assignation is a dangerous tool.

1.2.2.Identity declarations

An 'identity-declaration' (defining the meaning of an identifier within its range) consists of an equals-symbol "=" with a 'formal parameter' on its left and an 'actual-parameter' on its right:

formal-MODE-parameter = actual-MODE-parameter

(The "MODE" stands for an arbitrary mode. In the syntax you will find a number of production rules starting with MODE, generating an infinity of different constructs. MODE is a so-called "metanotion"; the capitals express that there are separate metaproduction rules for it. You may forget this remark; everything will become clear in the sequel.)

A formal-MODE-parameter consists of a formal-MODE-declarer followed by a MODE-identifier. The formal-MODE-declarer determines the mode of the internal object which will be ascribed to that identifier:

formal-parameter declarer:			identifier:	this identifier is a:
		amode	*thing*	amode-identifier
	ref	**amode**	*name*	reference-to-amode-identifier
ref	**ref**	**amode**	*pointer*	reference-to-reference-to-amode-identifier

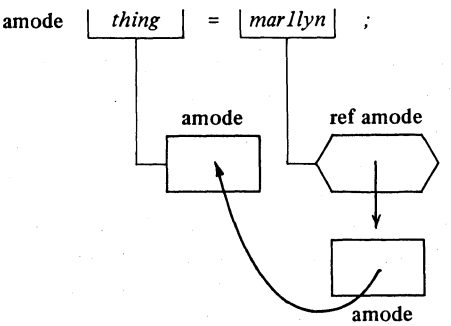
etc.

The actual-MODE-parameter yields an internal object to be ascribed to the MODE-identifier of the formal-parameter. We shall consider several possibilities for the actual-parameter:
1) a unit yielding the required mode (1.2.2.1 and 2)
2) a local-generator (1.2.2.3)
3) an initialized local-generator (1.2.2.3).

1.2.2.1.Constants

(E1) **amode** *thing* = *marllyn* ;

The actual-parameter *marllyn* is a simple case of a unit and it yields a **ref amode**; *thing,* however, is declared to be an amode-identifier (by the formal-declarer **amode**). Consequently, *marllyn* must be dereferenced to yield an **amode** value and what happens is:

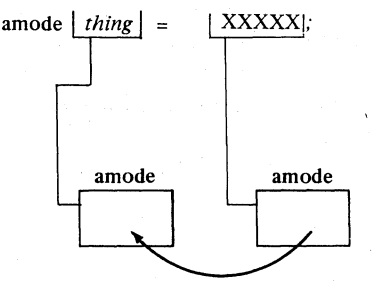


Now, by this declaration, a copy of the value currently referred to by *marllyn* is ascribed to *thing*. You can never assign to such a *thing*, because it is not a name. You may consider *thing* as a constant and indeed, whatever may happen to the **amode** value referred to by *marllyn*, *thing* always yields the instance of an **amode** value it got from *marllyn* at its declaration. "Constant" is to be understood as "constant until next elaboration of the declaration"; then it may get a different value from *marllyn*.

Instead of *marllyn* we may write any unit yielding, after the proper coercions, the required mode:

(E1*) **amode** *thing* = XXXXX ;

which may be depicted as follows:



1.2.2.2.Equivalences

(E2) **ref amode** *name* = *marllyn* ;

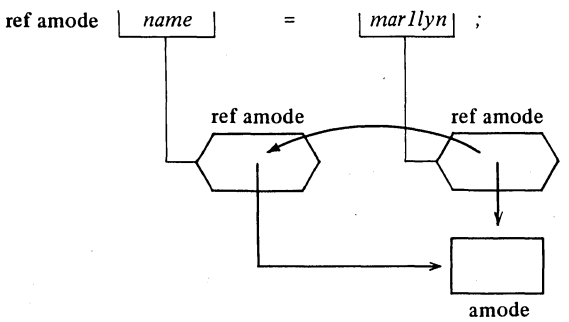
Although here again, as in E1, the actual-parameter is a strong unit, this is a story completely different from E1.

The formal-declarer in E2 is **ref amode**; and the formal-declarer determines

the mode of the object to be ascribed to the newly declared identifier, in this case a 'reference to amode' value is required from the actual-parameter.

The a priori mode of *marllyn* happens to be reference-to-amode. Consequently no dereferencing of the actual-parameter is needed. A copy of the name yielded by *marllyn* is ascribed to the identifier *name.*

What happens is:



The result of the elaboration of E2 is that we have got two different identifiers, *name* and *marllyn,* yielding different instances of the same name and consequently referring to the same internal **amode** object.

Assigning to *name* or to *marllyn* has the same result (supersedes the same **amode** value); different identifiers but the same variable value.

In some other programming languages this phenomenon is known as "equivalence". In this language "equivalence" is only a particular case of a general (and extremely fruitful) construction.

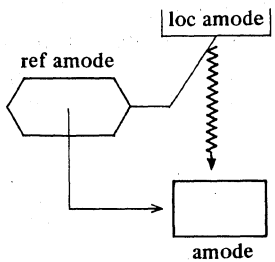
1.2.2.3. Local generators

We may want an identity-declaration to create the name of a new object (we want to define a new variable). Then we choose for the actual-parameter a 'generator'. A generator "generates" a new object; a local-generator creates a new object on the "stack"; an amode-local-generator creates a new **amode** object on the stack. On creating a new object, the generator yields its name.

The new object created by a local-generator ceases to exist when the range in which it occurs has been elaborated up to the hilt.

A MODE-local-generator consists of the local-symbol "**loc**", followed by an actual-MODE-declarer; the amode-local-generator is **loc amode. loc amode** generates an **amode** object on the stack and yields its name on that special occurrence. We shall depict the creation of the **amode** object by a special kind

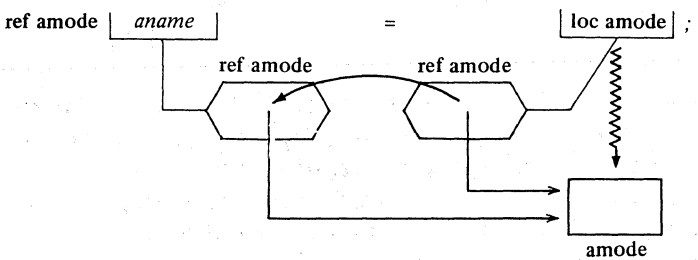
of arrow:



Now consider the identity-declaration:

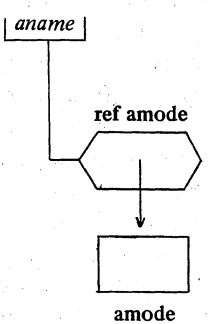
(E3) **ref amode** *aname* = **loc amode** ;

What happens is essentially the same as in E2:



The name yielded by **loc amode** is ascribed to the identifier *aname*, which consequently refers to the newly created **amode** object on the stack.

When the local-generator has done its work, the picture we are left with looks like this:



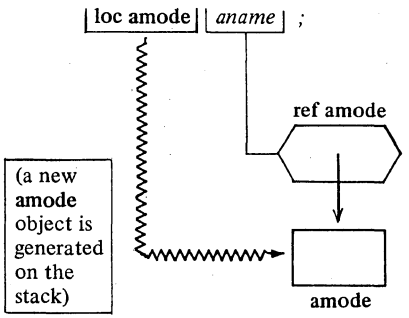
But we have seen many pictures like this before. They were brought about by variable-declarations such as:

(E3*) **amode** *aname ;*

It seems that E3* means exactly the same thing as E3. Every variable-declaration has an identity-declaration hiding behind it. Moreover, the **amode** in E3* is really a disguise for the generator **loc amode** in E3. Indeed, as was explained in 1.1.3, you may write the **loc** in E3* if you prefer:

(E3**) **loc amode** *aname ;*

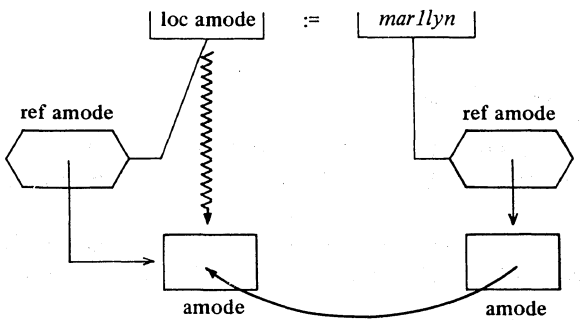
which goes to show why the **amode** in E3* must be regarded as an actual-declarer (remember that the **ref amode** in E3 was a formal one). We can show how the process of generation enters into our pictures of variable-declarations in the following way:



We often wish to assign an initial value to a newly generated variable. Now consider the assignation:

(E4) **loc amode** := *marllyn*

This is, of course, a perfectly correct assignation. What happens is:

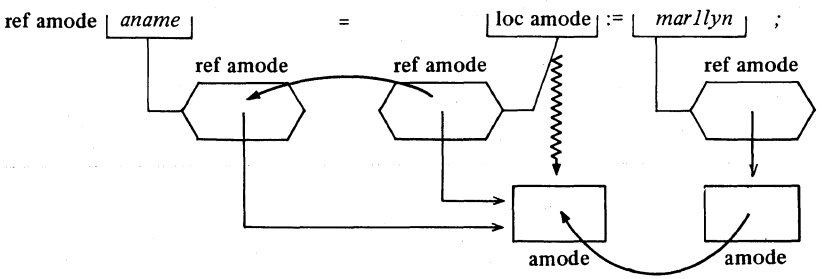


The local-generator yields the name of the newly created **amode** object, which is initialized to the value referred to by *marllyn*; *marllyn*, of course, is dereferenced (compare 1.1.E2). So far so good, but we cannot do much with it, because no external object in the whole program yields the name of our new **amode**.

However, an assignation yields the value yielded by its LHS (see 1.1.3) which is **ref amode**. We may now consider E4 as a special case of a reference-to-amode-unit (compare E2) and write:

(E4′) **ref amode** *aname* = **loc amode** := *marllyn ;*

And see what happens:



We should expect there to be a variable-declaration corresponding to this, and indeed there is:

(E4*) **loc amode** *aname* := *marllyn ;* Please, do not confuse this with **amode** *thing* = *marllyn ;* (see E1)

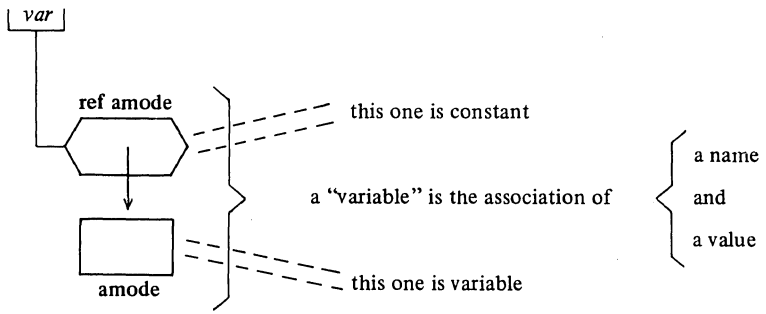
in which the **loc** is, as before, optional.

1.2.2.4. Variables and names

Syntactically, a 'variable' is a reference-to-MODE; a name.

Semantically, the object which is in fact "variable" is the object referred to.

Informally, we may choose an intermediate position and regard the pair of objects, consisting of an instance of a value and the name referring to it, as a variable:



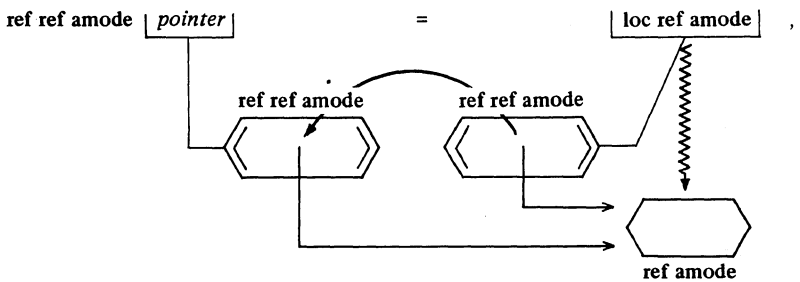
Going up one stair of references, we can generate variable names:

(E5) **ref ref amode** *pointer* = **loc ref amode** ;

or the corresponding variable-declaration:

(E5*) **ref amode** *pointer* ; Observe that again one **ref** is embezzled; it is a **ref ref amode** value that is ascribed to *pointer*.

What happens is:



This is essentially the same as E3. The generator **loc ref amode** generates a name on the stack. Such a **ref amode** on the stack may, by assignation to *pointer*, become an instance of a name referring to an **amode** object on the stack. In this way, "indirect addressing" is another fruit of the general concept of an identifier-declaration.

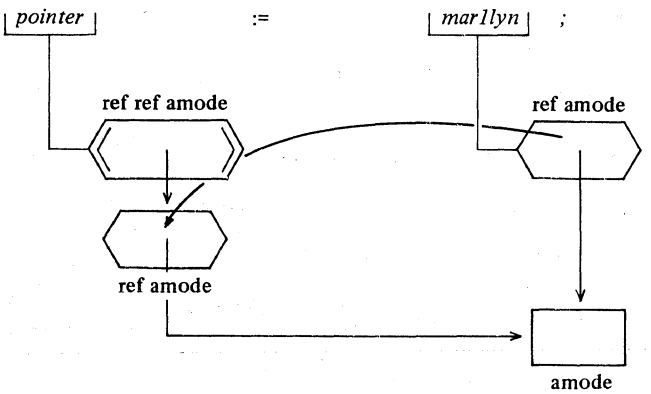
1.2.2.5. Casts

Consider the assignation:

(E6) *pointer* := *marllyn* ;

Here the value referred to by *pointer* (a **ref amode** object) is superseded by the value yielded by *marllyn* (its a priori value; of course there is no dereferencing in this syntactic position; the required mode is **ref amode**).

What happens is:



Now the value referred to by *pointer* refers to the value referred to by *marllyn* (describing indirect addressing in a natural language always leads to muddling sentences). Observe the resemblance with the situation in E2, we do the same thing at one reference level higher.

We could also have achieved this in the declaration of *pointer*, again by an initialized declaration.

(E6*) **ref amode** *pointer* := *marllyn* ;

To make things workable on the higher reference-to-something levels, we often need dereferencing in syntactic positions where coercion cannot do it for us; for example in the LHS of an assignation. In a reference-to-reference-to-MODE, we have at least two name levels, and we have to make clear which name is meant (how far down we want to assign). In E6* (*pointer* := *marllyn*) the value assigned is the **ref amode**.

Now suppose we want to assign the **amode** value of *object1* ◊ *object2* to the variable referred to by *pointer* (which is, after E6, the variable *marllyn*). We cannot do it without further preface. *pointer* is one **ref** above the level at which we want to assign.

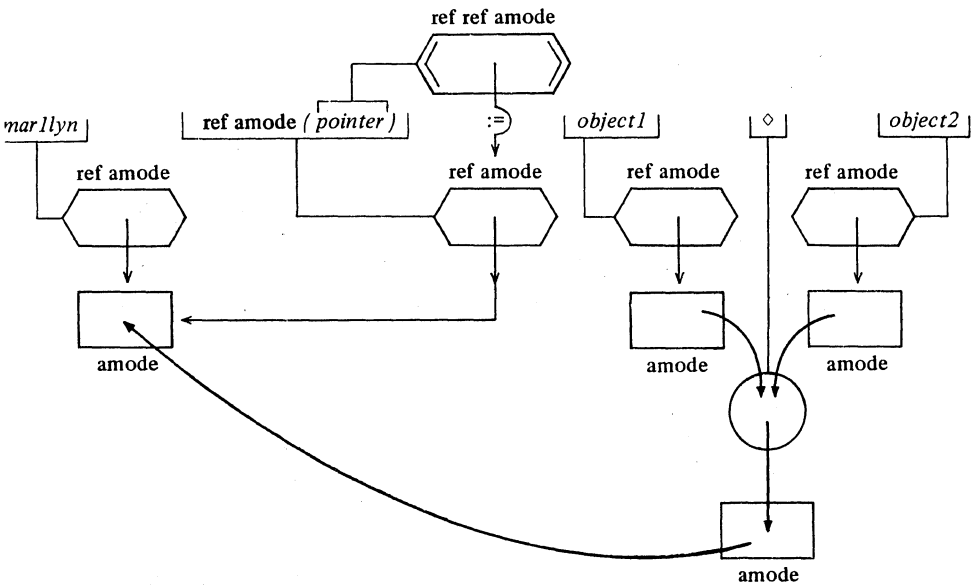
Now the "preface" is a remarkable little magic wand termed a 'cast', which provides in many situations where coercion fails.

If we now write:

(E7) **ref amode** *(pointer)* := *object1* ◇ *object 2*

the LHS is a cast obtaining a **ref amode** value from *pointer* (dereferencing it once), which is (after E6) the variable yielded by *marllyn.*

What happens is:



1.2.3. The metanotion MODE

In this language a mode is something you can define (declare) yourself in terms of other, already defined, modes (see 1.3.3.1). In the Report [R 1.2.1] you will find a set of metaproduction rules for "MODE", defining an infinite number of possible modes. They are all derived from the primitive modes:

bool , **char** , **int** , **real** , **bits** , **bytes** (see also Section 2.1.1)

There is also a special mode **void** (implying no value at all) which can turn up as the yield of a procedure (4.2.1) and also as the resident value of a **union** (5.6.1).

(Occasionally, we shall follow the syntatic style of the Report, as we already did on some occasions, writing for instance "reference-to-reference-

to-MODE-identifier". We do so just to point out that for these "notions" exist certain production rules in the syntax, by which they are defined ultimately as sequences of symbols. You can have some confidence that the intuitive meaning of these notions is in good accordance with their syntactic coherence and the meaning imposed upon them by the semantics of the language. Certain parts of notions are written in capital letters. For such "metanotions" exist separate metaproduction rules, defining them in terms of other notions. Some of these metanotions stand for an infinite number of other notions, which is the case with "MODE". Some others cover only a finite number.

There is no reason to worry about the syntax, but in the long run you might appreciate our attempts to break you softly to the syntactic saddle and the metanotional stirrups of the Report.

In this informal Introduction "**amode**" stands for "a mode" (you may conceive **amode** as a declarer for some, not specified, MODE). We shall also use indications like **bmode**, **umode**, **zmode**. For all these declarers you may substitute any MODE-declarer derived from the primitives, with the assistance of the symbols:

ref	(1.2.2)
proc	(1.2.3 and 4.2.1)
struct	(Sections 1.4.1 and 2.4.1)
"[" and "]"	(Sections 1.5.1 and 2.5.1)
union	(Sections 1.6.1 and 2.6.1)
long and **short**	(Sections 1.7.1 and 2.7.2)

We have already met *marilyn* and her sisters *mar1,2,3lyn* who all yield **ref amode** objects. We shall soon meet also their cousins *mar u lyn mar v lyn* and other *mar*-vellous ladies. However, we shall in the sequel substitute for **amode** other declarers (even **ref amode**), and all the girls will then follow the new fashions. We trust that you will recognise them in their other moods.

1.2.3.1. **proc** modes

In this section we consider the case in which we substitute for **amode** the declarer of a procedure with parameters delivering a value or not. We mainly do so to elucidate further the principle of identity, which is the main subject of 1.2. A more complete discussion of declarations in which procedures are involved will be found in 4.2.

All values of mode PROCEDURE are routines. A routine is the internal equivalent of the sequence of symbols which comprises some routine-text. In

a call this routine is activated. In a routine we can make use of formal-parameters; the actual-parameters are then supplied when the routine is called. It is not without reason that the LHS of an identity-declaration is denominated as the "formal-parameter", and the RHS as the "actual-parameter". The fact is that the identity-declaration states very precisely what happens when an actual-parameter is supplied, be it in a procedure call or in a formula.

A routine can be denoted by a routine-text, in much the same way as, for instance, "**true**" may denote the value of this sentence. In the denotation of a routine with formal-parameters we find the formal-PARAMETERS-pack, a sequence of formal-parameters separated by commas ",".

Declarers for procedures with parameters (see also 4.2.1) have the form:

not returning a useful value:	returning a **zmode** value:
proc *(* **umode** *)* **void**	**proc** *(* **umode** *)* **zmode**
proc *(* **umode** , **vmode** *)* **void**	**proc** *(* **umode** , **vmode** *)* **zmode**
proc *(* **umode** , **vmode** , **wmode** *)* **void**	**proc** *(* **umode** , **vmode** , **wmode** *)* **zmode**
etc.	etc.

We now reconsider:

(E1) **amode** *thing* = *marllyn ;*

We take for **amode** the declarer:

(E8.1) **proc** *(* **umode** , **vmode** *)* **zmode**

and for *marllyn* the routine-text:

(E8.2) *(* **umode** *u* , **vmode** *v)* **zmode**: XXXXX

in which we find the formal-PARAMETERS-pack *(* **umode** *u* , **vmode** *v)*, corresponding to the *(* **umode** , **vmode** *)* in E8.1.

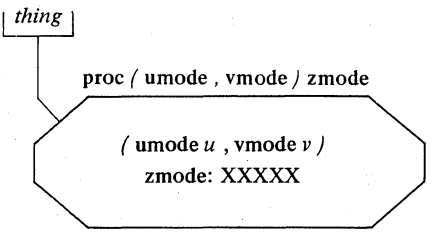
We thus obtain the identity-declaration:

(E8) **proc** *(* **umode** , **vmode** *)* **zmode** *thing* =
(**umode** *u* , **vmode** *v)* **zmode**: XXXXX *;*

This, however, seems to contain some redundancy, and it may therefore be replaced by the contracted form:

(E8*) **proc** *thing* = *(* **umode** *u* , **vmode** *v)* **zmode**: XXXXX *;*

The result of the elaboration of the 'routine-identity-declaration' E8* is that (a copy of) the routine E8.2 is ascribed to *thing*. Observe that the XXXXX is not elaborated at this stage. The result of the elaboration of E8* may be depicted as below:



1.2.3.2.1. The supply of the actual parameters (call by value)

Calling the *thing* of E8*, we have to supply an actual-umode-parameter and an actual-vmode-parameter. Let *mar u lyn* be a **umode** variable and *mar v lyn* a **vmode** variable. If we now "parametrize" *thing* by writing the actual-PARAMETERS-pack *(mar u lyn, mar v lyn)* right behind *thing*, we obtain the procedure call:

(E9) *thing (mar u lyn, mar v lyn)*

To elaborate this, we must first do some transformation of the routine (E8.2) yielded by *thing*:

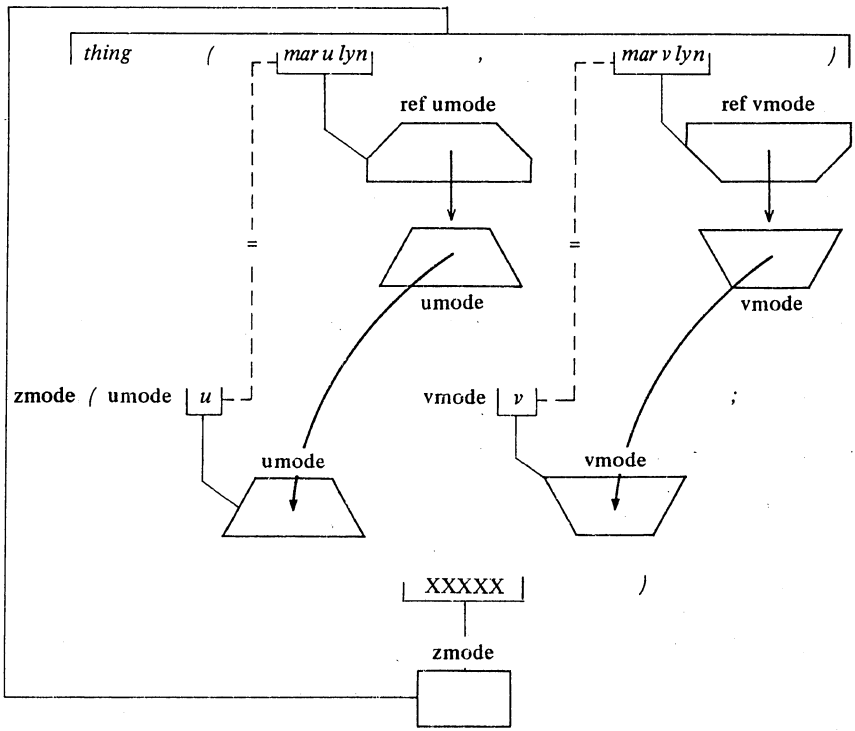
(E8.2*) **zmode** *(* **umode** *u* = ~, **vmode** v = ~; XXXXX *)*

The "~"s are only there as locum tenens for the actual-MODE-parameters; we will now get rid of them by replacement with the corresponding actual-parameters.

The result is the cast:

(E10) **zmode** *(* **umode** *u* = *mar u lyn* , **vmode** *v* = *mar v lyn* ; XXXXX *)*

This cast is then elaborated, yielding a **zmode** value which is then the value returned by the call:



If, for instance, *mar z lyn* is declared to be a **zmode** variable, then you may assign:

(E9*) *mar z lyn* := *thing (mar u lyn , mar v lyn)*

which, in fact, elaborates into:

(E10*) *mar z lyn* := **zmode** *(* **umode** *u* = *mar u lyn* , **vmode** *v* = *mar v lyn* ;
XXXXX *)*

1.2.3.2.2. The supply of the actual parameters (call by reference)

Observe that in E10 you cannot assign to the formal-parameters *u* and *v*; the identity-declarations in E10 are of type E1; *u* and *v* are constants, copies of the values referred to by *mar u lyn* and *mar v lyn* respectively. This situation has some similarity to "call by value" in some other programming languages.

If you want to assign a formal-parameter, you have to declare it to be a reference-to-MODE; the replacement action then leads to an identity-declaration of type E2 (equivalence, two names referring to the same instance of a value).

Consider, for example, the procedure declaration:

(E11) **proc** (**ref zmode** , **umode** , **vmode**) **void** *assign thing* =
(**ref zmode** *z* , **umode** *u* , **vmode** *v*) **void**: *z* := XXXXX

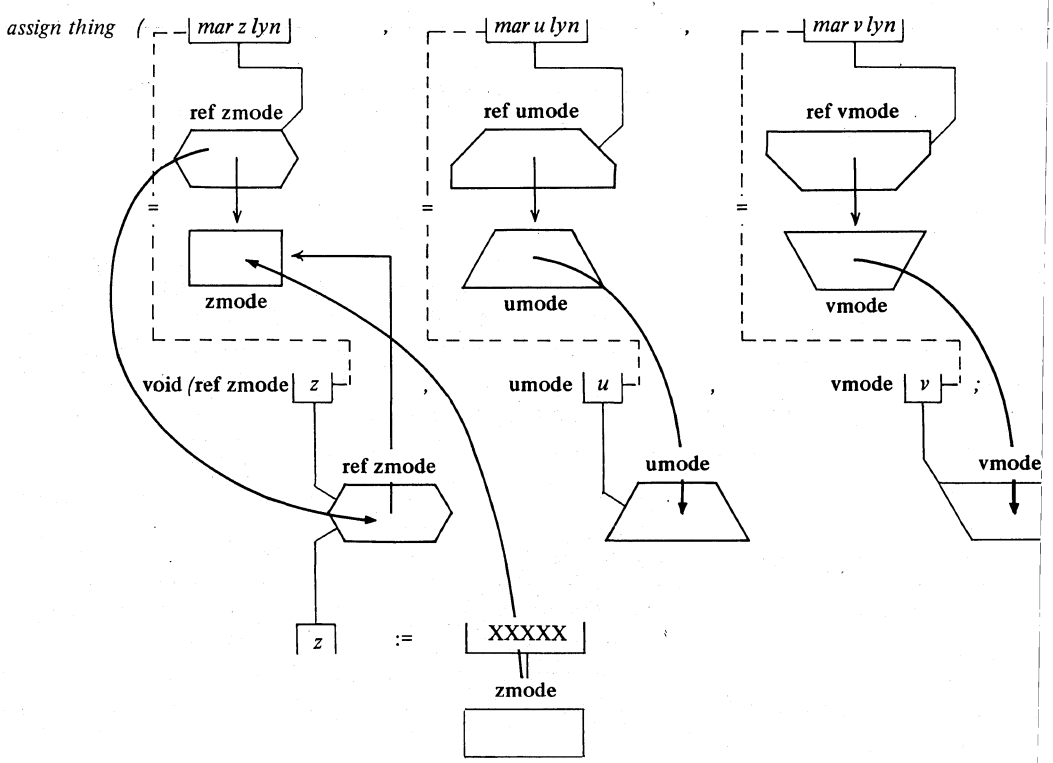
or, contracted:

(E11*) **proc** *assign thing* = (**ref zmode** *z* , **umode** *u* , **vmode** *v*) **void**:
z := XXXXX ;

The call:

(E12) *assign thing* (*mar z lyn* , *mar u lyn* , *mar v lyn*)

elaborates into:



The call E12 is equivalent to the cast:

(E12*) **void** *(* **ref zmode** *z = mar z lyn* , **umode** *u = mar u lyn* ,
vmode *v = mar v lyn ; z* := XXXXX *)*

It is important to observe that the first identity-declaration is of the type E2. Its effect is that, by its elaboration, a copy of the name yielded by the actual-parameter *mar z lyn* is ascribed to the formal-parameter *z*. The result is that *z* and *mar z lyn* both refer to the same **zmode** value. Consequently, assignation to *z* within the routine has the same result as assignation to *mar z lyn*. This is exactly what we wanted to achieve.

Conforming to the domesticated terminology of a "call by value", we might refer to the elaboration of a reference-to-MODE-parameter as a "call by reference".

The second and third declarations ascribe constants to *u* and *v*, copies of the values referred to by *mar u lyn* and *mar v lyn* respectively; these actual-parameters are "called by value".

1.2.3.2.3. The supply of the actual parameters (other possibilities)

Now we know exactly what happens to the actual-parameters in a procedure call, we shall find no difficulties in other applications of the principle of identity. For example:

Suppose you want to call *assign thing*, but you are only interested in the elaboration of the routine (for its side effects, for instance) but not in the implied assignation to the first parameter. Then you may call:

(E13) *assign thing (* **loc zmode** , *mar u lyn , mar v lyn)*

which elaborates into:

(E13*) **void** *(* **ref zmode** *z =* **loc zmode** , **umode** *u = mar u lyn* ,
vmode *v = mar v lyn ; z* := XXXXX *)*

and see what happens. The first identity-declaration is now of type E3; a variable of local scope is ascribed to *z*. The value in which you were not interested is assigned to this local variable and disappears when the elaboration of the routine is completed.

Suppose the value of, for instance, *mar v lyn* does not matter under some circumstances, and you have no **vmode** value at hand in the range where you want to call *assign thing*. Then you may call:

(E14) *assign thing (mar z lyn , mar u lyn ,* **skip** *)*

Now, when this call is elaborated, the textually third "~" in the transformed routine is replaced by **skip**. A **skip** happens to be a very docile little dud: it always delivers an undefined value of the required mode without any further action.

1.2.4. Summary

For their importance in this language, we review briefly the constructions E1, ---, E6. Ascribe them to identifiers in your own memory:

recommended form:		extended form :		
amode *thing* = *marllyn* ;		**amode** *thing* = *marllyn* ;		(E1)
ref amode *name* = *marllyn* ;		**ref amode** *name* = *marllyn* ;		(E2)
loc amode *aname* ;		**ref amode** *aname* = **loc amode** ;		(E3)
loc amode *aname* := *marllyn* ;		**ref amode** *aname* = **loc amode** := *marllyn* ;		(E4)
loc ref amode *pointer* ;	**ref**	**ref amode** *pointer* = **loc ref amode** ;		(E5)
loc ref amode *pointer* := *marllyn* ;	**ref**	**ref amode** *pointer* = **loc ref amode** := *marllyn* ;		(E6)

where the **loc**s in the left hand column may be omitted.

And remember:

(E1) *thing* does not yield a name and you cannot assign to it (provided, however, that **amode** does not happen to be a mode-indication for a **ref bmode**).

(E2) *name* yields the same name as *marllyn*; assignation to *name* has the same result as assignation to *marllyn* and vice versa.

(E3) *aname* yields a new name, different from all other names (that is what the local-generator achieves) and you can assign to it.

(E4) *aname* yields a new name (variable) and is initialized by assigning the value referred to by *marllyn* to it.

(E5) *pointer* yields a reference to a name (a variable name or name of a name); you can assign a name to it.

(E6) *pointer* yields a reference to a name and is initialized to refer to the name yielded by *marllyn*.

Vertical readers, please turn to 2.2.

1.3. Symbols, modes and operators

1.3.1. Representations

A 'program' is defined to be a sequence of 'symbols'. Consider for example:

```
begin real x, y, z;
      read (x); read (y); x := abs x; y := abs y;
      z := (x + y)/2 – sqrt(x × y);
      print (z)
end
```

This piece of program begins with the symbol "**begin**" followed by the sequence "**real**" "*x*" "," "*y*" "," "*z*" ";" "*read*" "(" "*x*" ")" ";" and so on. Typographical display features, such as blank space, change to new line, and change to new page have no significance in the language. Strictly speaking "**begin**", "**real**", "*x*", "," etc. are not themselves symbols; they rather represent them.

In the Report the representation(s) of symbols is strongly recommended, rather than explicitly prescribed. For the benefit of available charactersets, other representations may be chosen for a specific implementation of the language; one and the same implementation might even accept different representations from different input-devices. The given piece of program could for example look like:

```
'BEGIN' 'REAL' X, Y, Z;
      READ(X); READ(Y); X := 'ABS' X; Y := 'ABS' Y;
      Z := (X + Y)/2 – SQRT(X × Y);
      PRINT(Z)
'END'
```

or even:

```
.BEGIN .REAL X, Y, Z;
       READ(X); READ(Y); X := .ABS X;  Y := .ABS Y;
       Z := (X+Y)/2-SQRT(X*Y);
       PRINT(Z)
.END
```

On the other hand, if small letters and capitals are both available, one could for instance reserve the capitals for the construction of representations

for special symbols such as the begin-symbol, the real-symbol etc. If suitable tokens are available one could also choose other representations for the go-on-symbol and the becomes-symbol and the example might then look like:

```
BEGIN REAL x, y, z ⇒
      read(x) ⇒ read (y) ⇒ x ← ABSx ⇒ y ← ABSy ⇒
      z ← (x + y)/2 – sqrt (x × y) ⇒
      print(z)

END
```

In this Informal Introduction we shall always follow the recommendations of the Report [see R 9.4]. Where the Report suggest alternatives, we shall follow our own taste which may, however, depend upon the context. A complete list of the alternatives recommended by the Report is given in Appendix 1 and a particular recommended standard is given in Appendix 5.

1.3.2. Symbols, bold words and comments

In this language a rather extensive set of symbols is required, and moreover, we need some expedient for constructing an arbitrary number of new symbols.

A decisive point of course is the set of characters, types and marks producible by your input equipment; or, to state it more precisely, distinguishable by the input devices on your computer. If this happens to be you, then there is hardly any problem, thanks to the productive power of human handwriting and the perceptive qualities of the human eye . With an automaton there may, however, be some difficulty. Usually its senses are only able to distinguish a rather small set (some power of two) of different combinations of punched holes or magnetized spots in some material. In that binary form usually at least one font of letters (we represent them in lower case), ten digits, the punctuation marks ".", ",", ":", ";" and "'", one pair of brackets "(" and ")" and a more or less generally accepted set of marks, such as "+", "–", "×" (or "*"), "/" and "=", can be represented. In more favourable cases, the equipment may afford more luxury in the form of a second case of letters and/or some selection of types such as "<", ">", "[", "]", "∨", "∧" and perhaps even "¬", "↑", "$_{10}$" etc. In particular an underlining "_" and a vertical stroke "|" may be available and can be used to assist in the construction of other tokens like, for instance, "≠". "≤", "≥" etc.

Nevertheless, this language needs much more than all the marks

mentioned, and thus, even for the representation of the finite set of required symbols, an expedient to construct symbols from available marks appears to be essential. The Report recommends [R 9.4.2.2.b] that the extra symbols, or "bold words", be constructed similarly to identifiers, but distinguished from them by means of a "stropping" convention. The strop mechanism may be open- and close-apostrophes, or a period used as "boldface shift", or underlining, or bold type face, or the use of upper case letters (i.e. the capitals):

"*begin*" is an identifier or 'tag' (see 1.4)

but:

"'*begin*'", ".*begin*", "**begin**", "*begin*" *or* "*BEGIN*"

might be used as a stropped word to represent the begin-symbol.

Here we adopt stropping by bold type face. In this notational convention, a sequence of marks like "**notification**" is to be considered as one indivisible symbol and definitely not as a sequence of the symbols "**not**", "**if**", "**i**", "**c**", "**at**", "**i**" and "**on**", even though "**not**", "**if**", "**i**" and "**at**" happen to be proposed representations for required symbols, and "**c**" and "**on**" very well might be operators or mode-indications. Whether you want to consider such a construct as an ill chosen representation (in particular if such a splitting happens to make sense) or not is related to your inclination to meditate on problems concerning the amount of blank paper needed to separate spots of ink. Anyhow, you will be wasting your time, because the Report dictates [R 9.4.2.2.b] that, even though blank space, change to a new line and the like normally have no meaning in this language, you must use them to resolve this ambiguity by writing **not if i c at i on** if that is what you really want. Otherwise, the bold word **notification** is to be assumed.

We are thus able to construct as many symbols as we need. For example "**isnt**" for ":≠:" and "**at**" for "@" if we are unable or unwilling to produce or use ":≠:" and "@" on our input equipment. In particular we are now in the position to introduce as many bold words as we want, and we really need them for:

1) MODE-mode-indications (see 1.3.3.1)

for example we used **amode** as an amode-mode-indication

2) operators (see 1.3.3.2 and 3)

For the representation of an operator, it will be appropriate to use "+", "x", "∧" etc. if the action defined can be considered as an "addition",

"multiplication" or "conjunction" in some technical sense. Moreover, composite symbols such as +x, >=, +:=, ÷x:= are admissible [R 9.4.2.1]. For the rest we shall use bold words for operators as well.

A particular role is played by the comment-symbol represented by "**comment**", "**co**", "#" or "¢" and also, for special puposes, by "**pr**" or "**pragmat**". These symbols serve to step outside the language for a while.

A 'comment' consists of two matched comment-symbols enclosing an arbitrary sequence of characters, marks and types, not containing that comment-symbol. Thus **comment** *this is a comment* **comment** and **co** ¢ **co** are comments, but **co** *this is not a comment* ¢ is not. Comments can be inserted at any place in a program except inside an identifier, a bold word or a denotation (5.1.1.1 and 5.5.1.1).

A specific implementation may distinguish human comments, between two **comments** or ¢s, from pragmats between two **pragmat**s or **pr**s.

A (human) comment then serves to supply additional human information for the possibly human reader.

A pragmat may contain a message for a specific compiler (for instance to inform it to compile in some special mode or sub-language or to subjoin the program to some library or something), or for an operating system (for instance to inform it concerning certain required equipment or availability of hardware features, certain libraries etc.). A pragmat will usually be subject to the rules of a specific command language.

Consider the following program:

```
pr ALGOL68 pr
begin   comment this example is based on the revised
                report on the algorithmic language
                algol68 section 9.2; end of comment
        proc pr NONREC pr pr = void: pr;
        pr
        comment if NONREC means "nonrecursive compilation",
                whatever that may be, then we got into
                trouble comment

end pr RUN pr ¢ ??? ¢
```

In the standard-prelude (the standard declarations) of the Report a special comment-symbol "c" is used to express that the so called "pseudo-comment" should be replaced by a representation of a declarer or closed-clause suggested by that comment [R 10.1.2 Step 7]. In this Informal Introduction we shall follow this convention (e.g. in 3.7.2.E5).

1.3.3. Other declarations

Besides the identifier-declarations (1.1.2) we have:

1) mode-declarations (1.3.3.1)
2) operation-declarations (1.3.3.2)
3) priority-declarations (1.3.3.3).

1.3.3.1. Mode-declarations

A mode-declaration has the form:

mode MODE-mode-indication = actual-MODE-declarer

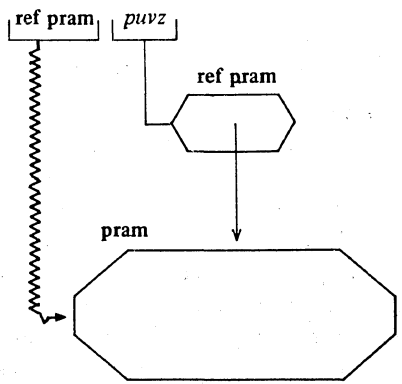
For example, in:

(E1) **mode pram = proc (umode, vmode) zmode ;**

the pram-mode-indication **pram** is declared to stand for the actual-declarer, which is **proc (umode , vmode) zmode**, and now, by declaring for example:

(E2) **loc pram** *puvz ;*

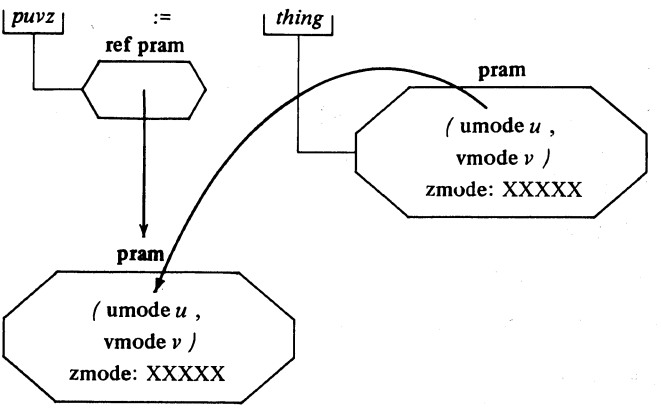
you ascribe to *puvz* a name referring to a value of the mode **pram,** which is a routine with a **umode** and a **vmode** as parameters, returning a **zmode** value:



We may now assign to the procedure variable *puvz*, for example:

(E3) *puvz* := *thing*

where *thing* is declared as in 1.2.3.1.E8 to yield a **proc (umode , vmode) zmode**. What happens in the elaboration of E3 is:



In fact we are now repeating the things we discussed in 1.2.

By virtue of the mode-declaration E1, the variable-declaration E2 "develops" into:

(E2*) **loc proc** (**umode** , **vmode**) **zmode** *puvz* ;

"to develop" is a technical term, which should be distinguished from "to elaborate".
In "elaboration" actions on internal objects are performed.
In "development", a mode-indication is replaced by its actual-declarer.

(One could say that "elaboration" is performed by the object code (at run time), while "development" is an action of the compiler.)

(In E3 the semantics of the lanugage state that the **pram** yielded by *thing* is copied into the **pram** referred to by *puvz*. As in other situations where copying is prescribed, one should remember an important remark in Section 2.1.4.1.a of the Report: "Any of these actions . . . may be replaced by any action . . . which causes the same effect". In particular where routines are manipulated, the implementor usually has other expedients at his disposal which "cause the same effect" as copying. The same applies to many other situations of this kind, particularly where copying might appear to be involved.)

You may declare a mode-indication as a convenient abbreviation for certain declarers (as, for instance, was the case in E1 and E2). You could do without them in these situations, at the price of time and ink.

There are, however, very interesting and important situations in which mode-declarations are indispensable for expressing certain essential

interrelations of objects in the memory. Some of these more involved mode-declarations will be used in other sections (see 1.4).

Circular mode-declarations like:

mode amode = amode ;
mode amode = bmode ; **mode bmode = cmode** ;
mode cmode = amode ;
mode amode = ref amode ;

might bring the compiler into difficulties and are apparently of no use. Consequently, they are regarded as not well-formed (2.4.3). However, there are constructions which might puzzle you at first sight, because they have an appearance of circularity (in fact are circular in some aspect), but nevertheless are very useful and can (easily) be implemented. Of course, such mode-declarations are well-formed and are not excluded.

1.3.3.2. Operation declarations

There are two kinds of operators:
monadic, declared as:

(E4) **op** (**umode**) **zmode m** = (**umode** *u*) **zmode**: XXXXX ;

or, by contraction:

(E4*) **op m** = (**umode** *u*) **zmode**: XXXXX ;

and dyadic, declared as:

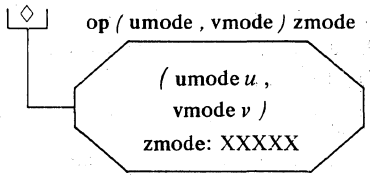
(E5) **op** (**umode** , **vmode**) **zmode** ◊ =
(**umode** *u* , **vmode** *v*) **zmode**: XXXXX ;

or, by the same contraction:

(E5*) **op** ◊ = (**umode** *u* , **vmode** *v*) **zmode**: XXXXX ;

Observe the resemblance to procedure declarations. Instead of declarers like **proc** (**umode**) **zmode** and **proc** (**umode** , **vmode**) **zmode** we have here **op** (**umode**) **zmode** and **op** (**umode** , **vmode**) **zmode**.

The result of the elaboration of an operation-declaration is that (a copy of) the routine is ascribed to the operator. For example, E5 (E5*) elaborates into:



In contrast to procedures, an operator can only be defined to yield a routine and not to refer to one. Consequently, the uncontracted forms (as in E5) are rarely of practical use, and we shall never write one again.

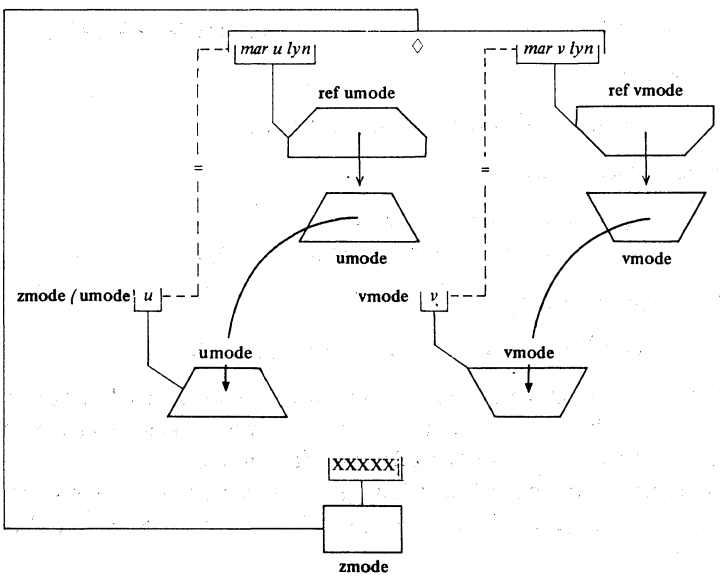
If *mar z lyn* is declared to be a **zmode** variable, then the assignation:

(E6) *mar z lyn* := *mar u lyn* ◊ *mar v lyn*

elaborates into:

(E6*) *mar z lyn* := **zmode** (**umode** *u* = *mar u lyn* , **vmode** *v* = *mar v lyn;*
XXXXX)

The cast in E6* is then elaborated, yielding a **zmode** value which is returned by the formula:



Observe that this picture is almost identical to the picture of Section 1.2.3.2.1.

There is, however, a fundamental contradistinction to procedures. For one and the same operator more than one declaration may occur within the same range. Which one then applies depends entirely on the mode(s) of the operand(s) in the particular formula in which the operator is applied.

For example:

(E7) **op** m = (**amode** *a*) **amode**: XXXXX ;
(E8) **op** m = (**bmode** *b*) **bmode**: WWWWW ;
(E9) **op** ◊ = (**amode** *a1, a2*) **amode**: ∀∀∀∀∀ ;
(E10) **op** ◊ = (**bmode** *b1, b2*) **bmode**: ∧∧∧∧∧ ;
amode *am, am1, am2* ; **bmode** *bm, bm1, bm2* ;

am1 := m *am2* ; ⇒ E7 applies
bm1 := m *bm2* ; ⇒ E8 applies
am := *am1* ◊ *am2*; ⇒ E9 applies
bm := *bm1* ◊ *bm2* ⇒ E10 applies

Observe that it is determined during the compilation of the formula which operator, i.e. which routine, applies.

1.3.3.3. Priority declarations

All monadic-operators have the same, the highest, priority.

For dyadic-operators nine priority levels can be declared by a priority-declaration of the form:

prio ◊ = DIGIT-token

in which "DIGIT-token" produces one of the nine digits "*1*" to "*9*".

In a formula with dyadic-operators of equal priority like:

(E11) *am* ◊ *am1* ◊ *am2* ◊ *am*

the implied bracketing is:

(E11*) *((am* ◊ *am1)* ◊ *am2)* ◊ *am*

Priority-declarations may impose another (implied) bracketing:

(E12) **prio** ○ = 6, ◊ = 7, x = 8 ;

the bracketing implied in the formula:

(E13) *a* x *b* ◊ *c* x *d* ○ *e* ◊ *f* ○ *g* x *h* ◊ *i* x *j* x *k* ○ *l*

is:

(E13*) *((((a x b) ◊ (c x d)) ○ (e ◊ f)) ○ ((g x h) ◊ ((i x j) x k))) ○ l*

Unless explicit bracketing requires otherwise, monadic-operators are elaborated first, i.e. they have the highest priority:

(E14) *am* := **m** *am1* ◊ **m** *am2*

is parsed like:

(E14*) *am* := *(* **m** *am1)* ◊ *(* **m** *am2)*

Of course it would have been possible (and in fact has been investigated) to declare different priority levels for different monadic-operators. However, it makes matters very awkward without much gain. The main root of this smallness of gain is that, if **m1**, **m2**, ---, **mn** were monadic-operators with different priorities, nevertheless only one parsing is conceivable for the formula **m1 m2** --- **mn** *operand,* Namely:

(**m1** *(* **m2** *(* --- *(* **mn** *operand)* --- *)))*

The gain can thus be found only in combination with dyadic-operators. There is only one situation in which you might feel sorry (see 5.1.3).

Vertical readers, please turn to 2.3.

1.4. **Stowed values, structures**

1.4.0. STOWED values

In this language values (one or more) can be STOWED (i.e. collected) to form a value of a new mode. The metanotion "STOWED" stands for:

1) 'structured with FIELDS mode'

or

2) 'ROWS of MODE'

corresponding to two entirely different systems of collecting:

1) into a "structured value" (this section)
2) into a "multiple value" (section 1.5).

In a multiple value you collect values of essentially the same mode, its "elements", each of which can be selected by a specific set of subscripts. The mode of a multiple value is 'ROWS of MODE' and covers the concept of "array" (or "vector", "matrix". "dimension", etc.) in other programming languages.

In a structure you collect values of (not necessarily) different modes, the "fields" of the structure, each of which can be selected by a specific field-selector. Structured values cover what in other programming languages are known under a variety of names like "records", "lists", "trees", "queues", "chains", etc.

The important feature of structures is that values of different modes may be collected into them. In particular, one or more of the fields may be references to other values, in which way lists and trees of all kinds may be constructed.

Another important feature, however, is that the selection of a field in a structure may very well take place at compile time, whereas the subscripts of an element in a multiple value are usually determined (computed) at run time. Therefore, even in situations in which multiple values are the only possibility in many other programming languages, in this language you will often use structures instead. A good example is the **compl** (see Section 2.4.4).

Finally, the general concept of MODE allows you to build multiple values the elements of which are structures, and vice versa to build structures the fields of which are multiple values.

1.4.1. Enumeration by tagging

By a mode-declaration like:

(E0) **mode triple = struct** *(* **umode** *first,* **vmode** *second,* **wmode** *third) ;*

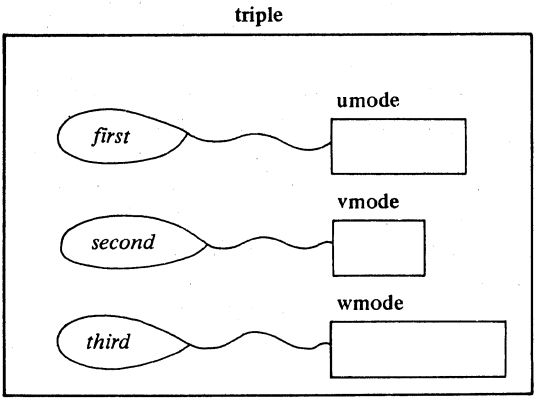
triple is declared to "specify" a new class (mode) of values, each of which is structured with three fields, a **umode** field *first*, a **vmode** field *second* and a **wmode** field *third* . *first, second* and *third* are the 'field-selectors'.

Syntactically a field-selector is a sequence of letters and digits with a leading letter. It may look like an identifier but it is not. It is important to recognize clearly its function:

A field-selector as such does not yield any internal object.
A field-selector is part of a declarer or of a selection.

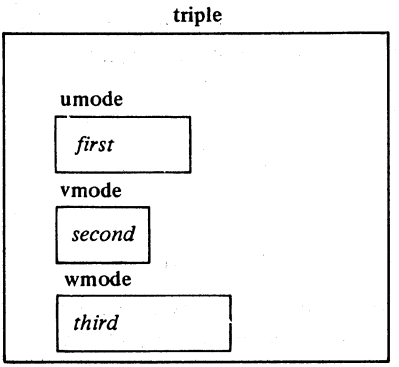
A **triple** object, as declared by E0, may be visualized as a box with a **umode**, a **vmode** and a **wmode** box within it, the fields of the **triple.** These

fields can be "pulled up" by their field-selectors *first*, *second* and *third*. We might imagine a piece of cord between the field and its tag:



It is important to bear in mind that *first*, *second* and *third* are not names referring to the fields (see 1.4.1.1)

Merely to simplify our drawings we shall often write the selectors inside the boxes:



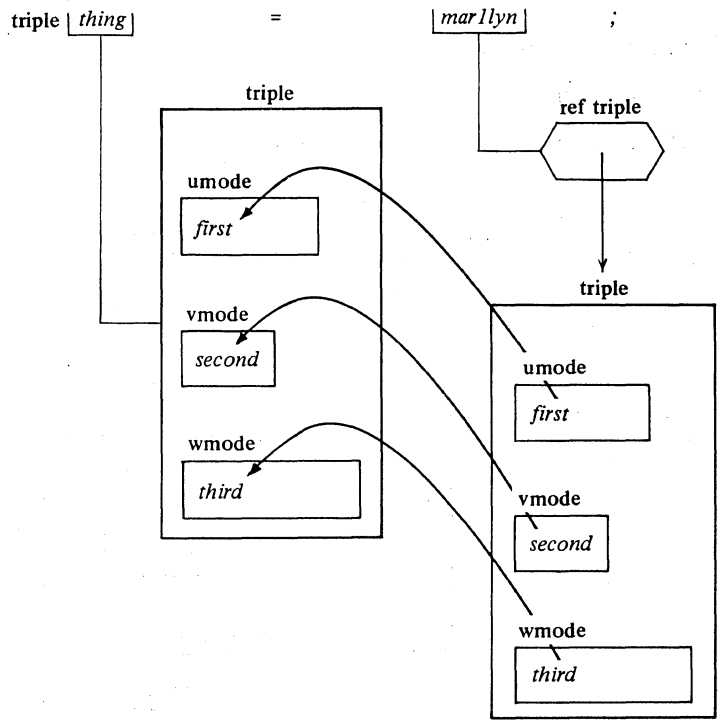
warning : our drawings serve to visualize internal objects and matters of elaboration;
please do not confuse the selector in the box with the instance of a value in it.

1.4.1.1. Structured constants

We now reconsider the three fundamental identity-declarations E1, E2 and E3 of Section 1.2, in which we substitute systematically **triple** (as declared in E0) for **amode**:

(E1) **triple** *thing* = *marllyn* ;

By E1, a copy of the **triple** value referred to by *marllyn* (which is itself of mode **ref triple**) is ascribed to *thing*:



After this declaration *thing* yields a **triple** object and its fields can be selected separately by pulling them up by their tags:

first	**of** *thing*	selects	the **umode** object ,
second	**of** *thing*	selects	the **vmode** object ,
third	**of** *thing*	selects	the **wmode** object .

Because *thing* does not refer to a **triple**, you cannot assign to it. Consequently you cannot assign to its fields *first* **of** *thing*, *second* **of** *thing* and *third* **of** *thing*, which in their turn yield the fields of *thing*.

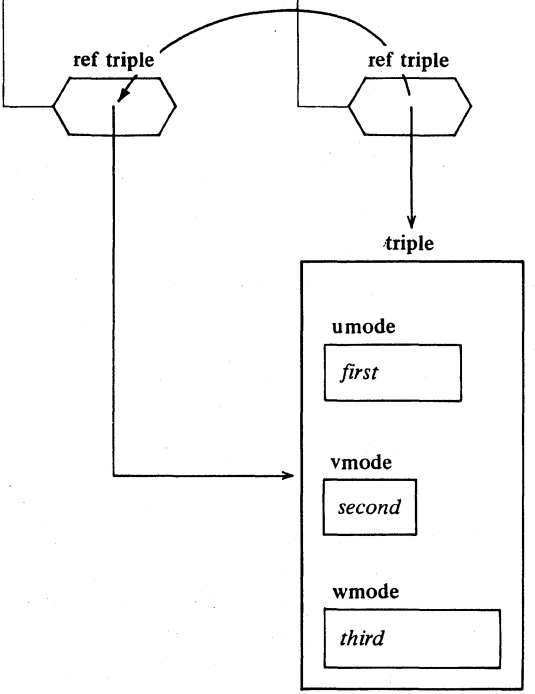
In the declaration E1, *marllyn* is dereferenced, because *thing* is required to be of **triple** mode, and its fields are copied into the **triple** *thing* thus defined; the supersession is a triple action (three fields are copied). But, in the reach of this declaration, whatever may happen to the fields of the **triple**

referred to by *mar1lyn,* nothing can happen to the fields of the **triple** yielded by *thing*. You may select them, you cannot change them; *first* **of** *thing* etc. are not names.

1.4.1.2. Names of structures

By the declaration:

(E2) **ref triple** *name* = *mar1lyn* ;



mar1lyn is not dereferenced (see 1.2.E2) as in E1, because the formal-parameter requires a **ref triple** value.

Clearly, *name* is a name, referring to the same **triple** object as *mar1lyn* (two identifiers referring to the same internal **triple**).

You may assign to *name*, for example:

(E2*) *name* := *mar2lyn*

by which assignation the **triple** object referred to by *mar2lyn* is copied into the **triple** object referred to by *name* (and by *mar1lyn*).

Moreover you can assign to the names:

first **of** *name* , *second* **of** *name* and *third* **of** *name* .

For example, the assignation E2* is equivalent to the collateral assignation:

(E2**) (*first* **of** *name* := *first* **of** *mar2lyn* ,
second **of** *name* := *second* **of** *mar2lyn* ,
third **of** *name* := *third* **of** *mar2lyn*)

Recapitulating:

first **of** *thing* , *second* **of** *thing* and *third* **of** *thing*
yield the fields of the **triple** yielded by *thing* ,

but:

first **of** *name* , *second* **of** *name* and *third* **of** *name*
yield names which refer to the fields of the **triple** referred to by *name* . These are known as the "subnames" of *name* .

1.4.1.3. Creation of new structures

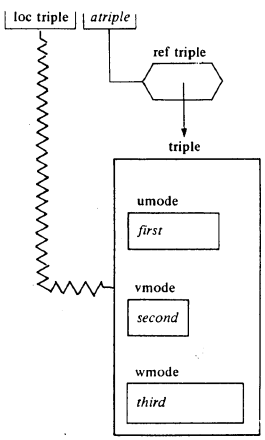
A new **triple** variable can be declared (see 1.2.E3) by means of a local-generator:

(E3) **ref triple** *atriple* = **loc triple** ;

or by the more usual variable-declaration:

(E3*) **triple** *atriple* ; or **loc triple** *atriple* ;

What happens is essentially the same as in 1.2.E3**:



Of course you can also initialize a thus declared new **triple**:

(E4) **ref triple** *atriple* = **loc triple** := *marllyn ;*

or more usually:

(E4*) **triple** *atriple* := *marllyn ;*

You are not required by the syntax to declare a new mode like **triple** (E0) before you give identifier-declarations; that is, you may very well declare:

(E3**) **struct** *(* **umode** *first* , **vmode** *second* , **wmode** *third) atriple ;*

In most cases, however, you will spare time and ink by a mode-declaration. Moreover, if you use structures to construct lists etc., the mode-declaration is indispensable (see 1.4.3).

1.4.2. Different objects in one box

The fields of a structure may be of different modes, but could also be the same. In the latter case it is often a matter of efficiency (or even convenience) whether you declare them in a structure or in a multiple value. You can stow as many values in a structure as you wish, but you have to enumerate them in the declaration by tagging the fields explicitly. The number of fields in a structure is determined statically (at compile time). The minimum number is one, the maximum depends only on your perseverance in writing them down.

Of course the field-selectors in one structure must all be different. However, if it suits you, you may very well use the same sequence of symbols as a field-selector in different structures or even elsewhere as an identifier.

Just to give you some impressions, consider:

(E5) **mode threeofakind = struct** *(* **amode** *one , two , three) ;*

which is a contraction of:

(E5*) **mode threeofakind = struct** *(* **amode** *one* , **amode** *two* , **amode** *three) ;*

And consider further:

(E6) **mode couple = struct** *(* **man** *one* , **wife** *two) ;*

(E7) **mode largebox = struct** *(* **amode** *one , two , three ,*
bmode *first , second , third , fourth ,*
fifth , sixth ,

```
cmode a , b , c , d , e , f , g , h , i , j , k ,
      l , m , n , o , p , q , r , s , t , u ,
      v , w , x , y , z );
```

Now consider the variable-declarations:

```
(E8)  threeofakind  one , two , three , four , five , six , seven , eight ;
      couple        romeo and juliet , tristan und isolde ,
                    daphnis et chloe ;
      largebox      a, b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r ,
                    s , t , u , v , w , x , y , z ;
```

and observe that there are no ambiguities in:

```
(E9)  one of two, two of three, one of one ,
      two of romeo and juliet, one of daphnis et chloe ,
      one of t, t of t, o of o, a of b ,
      etc.
```

There is no restriction on the modes of the fields in a structure; every mode is allowed (including structures, see 1.4.4). Consider, for example:

```
(E10) mode surprisepacket =
          struct ( umode umode , vmode vmode , zmode zmode ,
                   proc ( ref zmode , umode , vmode ) void proc ) ;
```

and the variable-declaration:

```
(E11) surprisepacket s , s1 , s2 , s3 ;
```

and the assignations:

```
(E12) umode of s1 := mar u lyn ;
      vmode of s2 := mar v lyn ;
      proc  of s  := assign thing ;
```

By virtue of E11, *proc* **of** *s* is a procedure variable, a reference to a **proc** *(* **ref zmode** , **umode** , **vmode** *)* **void** to which we assign in E12 the compatible routine yielded by *assign thing* (see 1.2.E11*). If we now parametrize *proc* **of** *s* (1.2.3.2.1), it will be dereferenced to yield a routine which can be called (this will be discussed in 4.2.2.2). For reasons to be discussed later (5.5.1.3.E30) we must put brackets around *proc* **of** *s* before parametrizing it.

Now you may fish out what happens by elaboration of the call:

```
(E13) ( proc of s ) ( zmode of s3, umode of s1, vmode of s2 )
```

1.4.3. Chaining

By declaring a field of a structure to refer to (to be the name of) another value, you can chain this structure to that other value. If, in particular, this other value is of the same mode as the structure, we are able to chain values of the same mode (i.e. to construct "queues", "lists", "trees" etc.).

Consider:

(E14) **mode box** = **struct** (**amode** *value*, **ref box** *next*) ;

This mode-declaration is one of those which have an appearance of circularity (see 1.3.3.1). You might think that it develops into:

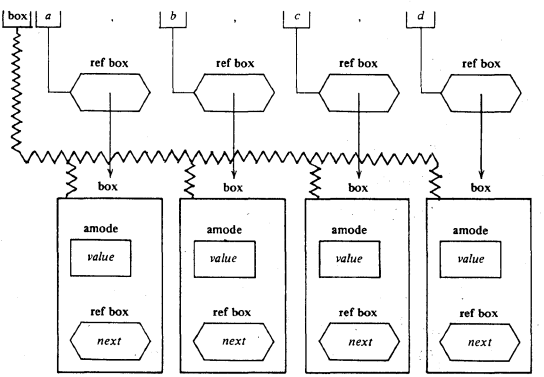
```
mode box = struct ( amode value ,
           ref   struct ( amode value ,
           ref   struct ( amode value ,
           ref   struct (                etc. ad infinitum
```

This, however, is not the case. The **box** textually contained in **ref box** *next* in E14, is "shielded" by the **struct** and the **ref** (see 2.4.3 for the details). Therefore the compiler does not develop **box** (in this syntactic position).

Consider the identity-declarations:

(E15) **box** *a* , *b* , *c* , *d* ;

The result of their elaboration will be:



Observe that the field tagged *next* is of the same mode as the name ascribed to *a* , *b* , etc. There is not a tittle of infinity about the size of these **boxes**.

Let us first assign:

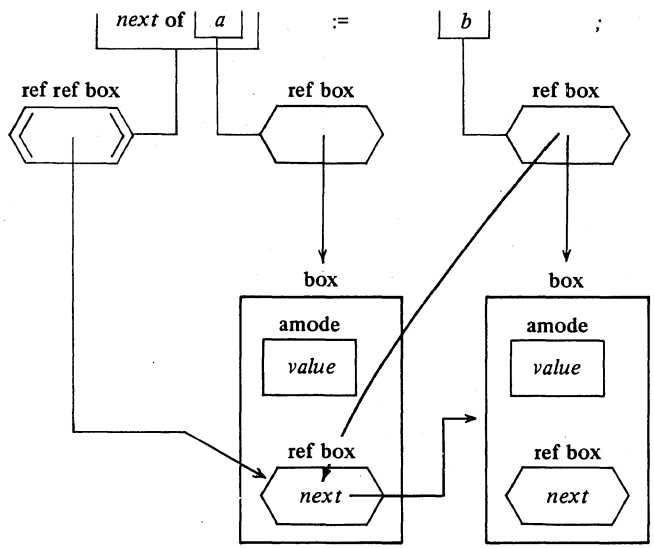
(E16)
```
value of a := mar1lyn ;
value of b := mar2lyn ;
value of c := mar3lyn ;
value of d := mar4lyn ;
```
mar3 and *4lyn* are **ref amodes** like their sisters (see 1.1.2.2).

Because *a* refers to a **box**, *value* **of** *a* is the subname referring to its **amode** field. Consequently, *value* **of** *a* yields a **ref amode** value and thus is an **amode** variable, which is why we can assign *mar1lyn* to it. The four *mar lyns* are dereferenced and their **amode** values are copied into the *value* **of** fields of the **box**es referred to by *a* , *b* , *c* and *d* respectively.

Much more interesting is to see what happens when we assign:

(E17.1) *next* **of** *a* := *b* ;

By the same reasoning as above, *next* **of** *a* is the subname referring to the **ref box** field of the **box** referred to by *a*. Consequently, *next* **of** *a* yields a **ref ref box** value and thus is a **ref box** variable (is a variable name) (see also 5.4.2). Therefore *b* in the RHS of the assignation is not dereferenced and the value yielded by *b* is copied into the field *next* of the **box** referred to by *a*:



The result is that we have chained the **box** referred to by *a* to the **box** referred to by *b* via the *next* **of** field of the **box** *a*.

Similarly:

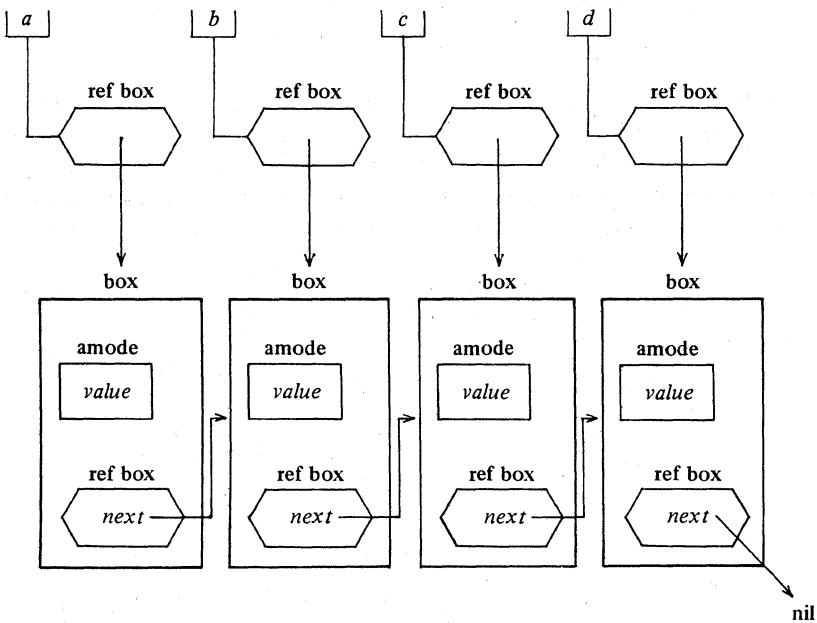
(E17.2) *next* **of** *b* := *c* ;

(E17.3) *next* **of** *c* := *d* ;

Finally, we want to express that *d* is the last **box** in the chain. Then we must give a special value to its *next* **of** field, recording this fact. Such a value is **nil**, which is "a name referring to no value":

(E17.4) *next* **of** *d* := **nil** ;

What we have achieved by the assignations E17 is:



Let us now consider the assignation:

(E18) *value* **of** *next* **of** *next* **of** *next* **of** *a* := *marilyn*

next **of** *a* refers to *b* ,
next **of** *b* refers to *c* ,
next **of** *c* refers to *d* ,

consequently, *value* **of** *next* **of** *next* **of** *next* **of** *a* refers to the **amode** field of the **box** referred to by *d*. It thus appears that E18 was a rather complicated

way of prescribing:

(E18*) *value* **of** *d* := *marilyn*

In a mode as declared in E14 we can build single threaded lists. Of course we can chain in much more complicated ways.

For example:

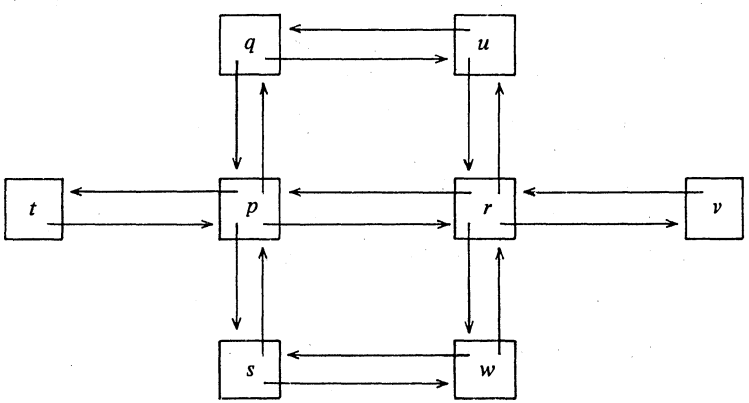
(E19) **mode node = struct (** **amode** *mainvalue* ,
proc (**amode** **)** **amode** *function* ,
ref node *north* , *east* , *south* , *west* **)** ;

(E20) **node** *p*, *q*, *r*, *s*, *t*, *u*, *v*, *w* ;

We may assign:

(E21) *mainvalue* **of** *p* := *marilyn* ;
etc.
function **of** *p* := **(** **amode** *a* **)** **amode**: XXXXX ;
etc.

(E22) *north* **of** *p* := *q* ; *east* **of** *p* := *r* ; *south* **of** *p* := *s* ; *west* **of** *p* := *t* ;
north **of** *r* := *u* ; *east* **of** *r* := *v* ; *south* **of** *r* := *w* ; *west* **of** *v* := *r* ;
north **of** *s* := *p* ; *east* **of** *t* := *p* ; *south* **of** *q* := *p* ; *west* **of** *u* := *q* ;
north **of** *w* := *r* ; *east* **of** *q* := *u* ; *south* **of** *u* := *r* ; *west* **of** *r* := *p* ;
east **of** *s* := *w* ; *west* **of** *w* := *s* ;



You might now like to meditate on expressions like:

```
(E23)  function of w := ( amode a ) amode:
                            if mainvalue of w = a
                          then mainvalue of north of w
                           else (function of v) (mainvalue of east of w)
                            fi
```

1.4.4. Pandora's boxes

A field of a structure may be another structure with a field which may be another structure and so on:

```
(E24)  mode pandora = struct ( amode a , pando p ) ;
(E25)  mode pando   = struct ( amode a , pan   p ) ;
(E26)  mode pan     = struct ( amode a , ref pandora next ) ;
```

By virtue of E25 and E26, E24 develops into:

```
(E24*) mode pandora = struct ( amode a ,
                               struct ( amode a ,
                                        struct ( amode a ,
                                                 ref pandora next
                                               ) p
                                      ) p
                             ) ;
```

Intentionally, we chose the selectors of the fields of **pandora** and its inner fields somewhat confusingly, just to point out that such is allowed (though it may not be wise).

```
(E27)  pandora p ;
```

Now observe that:

a **of** *p*	refers to an **amode** value ,
a **of** *p* **of** *p*	refers to an **amode** value ,
a **of** *p* **of** *p* **of** *p*	refers to an **amode** value ,

but:

a **of** *p* **of** *p* **of** *p* **of** *p*	is meaningless

as are:

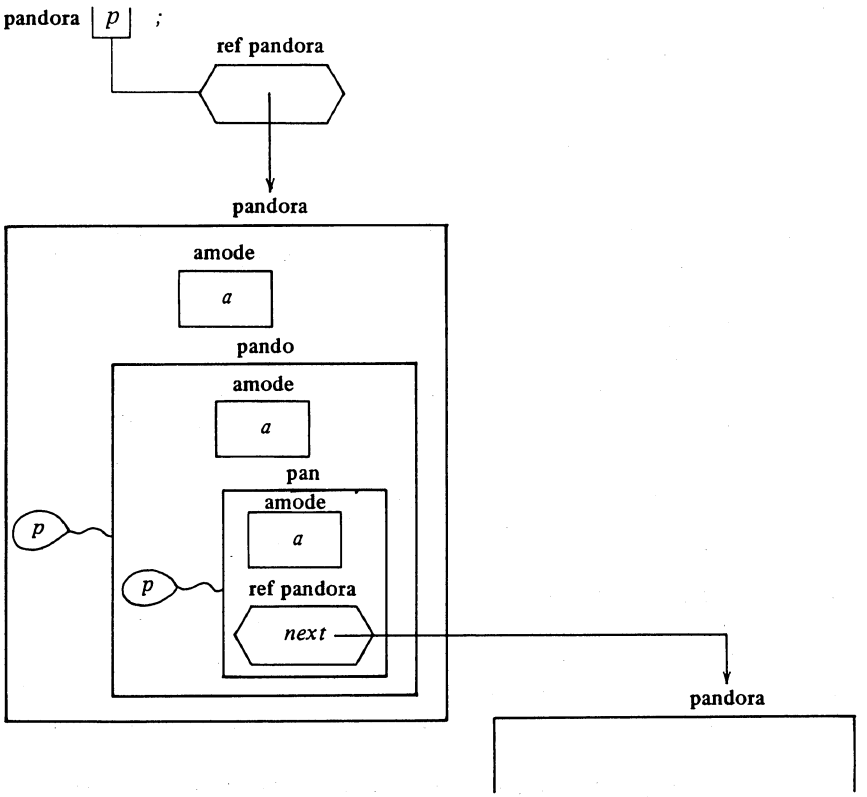
next **of** *p*	because a **pandora** has no *next* **of** field ,
next **of** *p* **of** *p*	because a **pando** has no *next* **of** field ,

but:

next **of** *p* **of** *p* **of** *p* refers to a **pandora**

as do also:

next **of** *p* **of** *p* **of** *next* **of** *p* **of** *p* **of** *p* ,
next **of** *p* **of** *p* **of** *next* **of** *p* **of** *p* **of** *next* **of** *p* **of** *p* **of** *p* , *etc.*



The well-formed rule (2.4.3) prohibits circular definitions like:

mode pandoravel = struct (**amode** *a* , **pandoravel** *p* **) ;**

Here **pandoravel** is not shielded by a **ref**. The compiler cannot do anything sensible with this **pandoravel**. Obeying such a mode-declaration would result in an endless loop of development.

Observe that, as soon as a **ref** stands in front of some declarer, the only thing the compiler has to do is (to be prepared) to reserve a location for holding a name; that is why a declarer following a **ref** in a **struct** can be shielded.

Vertical readers, please turn to 2.4.

1.5. Stowed values, multiples

1.5.1. Multiple values and descriptors

A multiple value (or "multiple" for short) consists of:

1) zero, one or more values, all of the same mode.
   These values are the "elements" of the multiple. Each element is selected by a set of one or more integers, its "subscripts". In this section, we use *h* , *i* , *j* , *k* , *m* , *n* , *m1* , *n1* , . . . as units yielding an integral value.
2) a "descriptor".
   A descriptor describes the subscripts that are required to select an element—how many of them are needed and what bounds are to be set on their values.

Examples of the modes of multiple values are:

[ ] amode [,] amode [,,] amode [,,,] amode

which should be pronounced as 'row of amode', 'row row of amode', 'row row row of amode' and so on, or, in general, as some 'ROWS of MODE' where "ROWS" stands for as many times 'row' as you may require. The number of 'row's in the mode is the number of subscripts needed to select an element, and "MODE" (**amode** in the examples) specifies the mode of each element.

Syntactically, **[ ] amode**, **[,] amode**, etc. are formal-ROWS-of-MODE-declarers (you can use them in formal-parameters). Actual-ROWS-of-MODE-declarers (for use in generators and therefore also in variable-declarations) are more complex, since they must contain the descriptor of the value to be generated. Here are some examples:

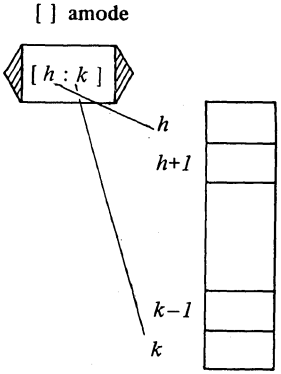
[*m* : *n***] amode**	¢ one 'row' in the mode ¢
[*k* : *k* , *m* : *n***] amode**	¢ two 'row's in the mode ¢
[*m1* : *n1* , *m2* : *n2* , *m3* : *n3***] amode**	¢ three 'row's in the mode ¢

and even

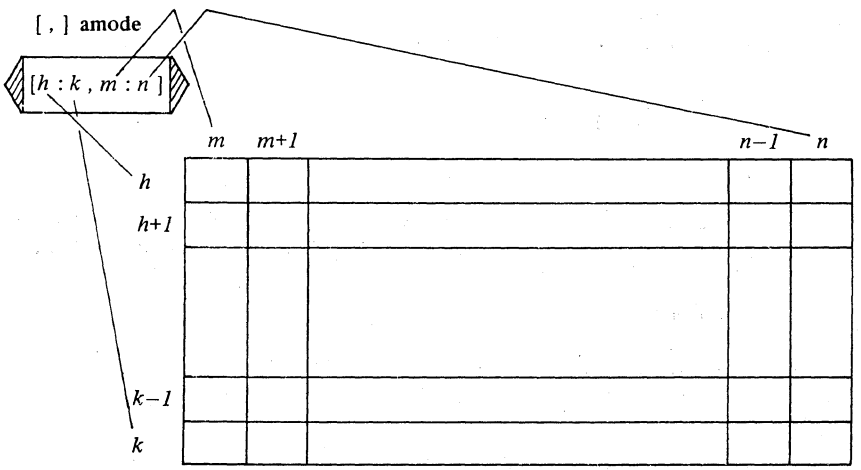
[*k* : *k***] [***m* : *n***] amode** ¢ one 'row', but the 'MODE' also has a 'row' in it, giving 'row of row of amode' ¢

m : *n*, *k* : *k*, etc. are "boundpairs"; [*k* : *k*, *m* : *n*] specifies the complete descriptor. Each boundpair consists of a lower-bound (to determine the lowest acceptable subscript), a colon, and an upper-bound (to determine the highest acceptable subscript). If the upper-bound of any boundpair is lower than the corresponding lower-bound, then the descriptor is "flat" and the multiple value consists of zero elements.

We shall bring multiple values into our pictures in the following way:



It is important to realize that the descriptor belongs to the multiple value. You even have a certain access to it (see 1.5.5).



1.5.2. Indexing

To select a "subvalue" of a given multiple (i.e. a value which is a subset of that given multiple value), we use 'indexers'. The smallest subvalue is one individual element of the multiple, which is obtained by "subscripting". All other subvalues can be obtained by "trimming"; the mode of a subvalue thus obtained is some 'ROWS of MODE'.

1.5.2.1. Indexers

An indexer consists of a sub-symbol "[", followed by one or more 'trimscripts' separated by comma-symbols followed by a bus-symbol "]" (see below). A trimscript is a trimmer-option (i.e. a 'trimmer' or EMPTY) or a 'subscript'. Examples:

[*i*] [*i* , *j*] [*i* , *j* , *k*]		*i* , *j* and *k* are subscripts; a subscript may be almost any unit yielding an integral value (see 5.5.1.3)
[*i* : *j*]	[*h* : *j* , *i* : *k*]	all *i* s and *h* s are lower-bounds ,
[*i* :]	[: *j* , *i* : *k*]	all *j* s and *k* s are upper-bounds ;
[: *j*]	[, *i* : *k*]	all such bounds must again yield
[ ]	[*i* : , : *k*]	integral values ;
	[,]	*i* : *j* , *h* : *j* , *i* : *k* etc. are trimmers

If a bound is omitted, then its value is that of the corresponding bound in the descriptor of the given multiple. If both bounds are omitted, then the colon may be omitted also.

Examples in which we find trimscripts of both kinds in an indexer are:

[*h* , *i* : *j*] [, *k*] [, , *k*]
[*h* , *i* : *j* , *k*] [: *h* , *i* , *k* :]

A special kind of a trimmer is a trimmer with a revised-lower-bound:

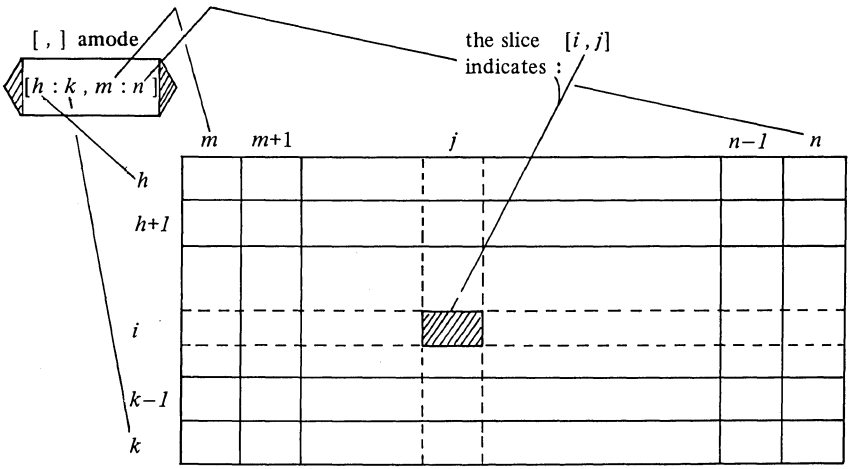
	[*i* : *j* **at** *h*]	the lower-bound of the multiple subvalue trimmed by *i* : *j* gets a new value which is the value of *h* ; a revised-lower-bound must again yield an integral value.
or	or (in another notation)	
	[*i* : *j* @ *h*]	

In the absence of a revised-lower-bound (but not when both bounds and the colon are omitted also), the multiple subvalue gets a revised lower-bound of 1.

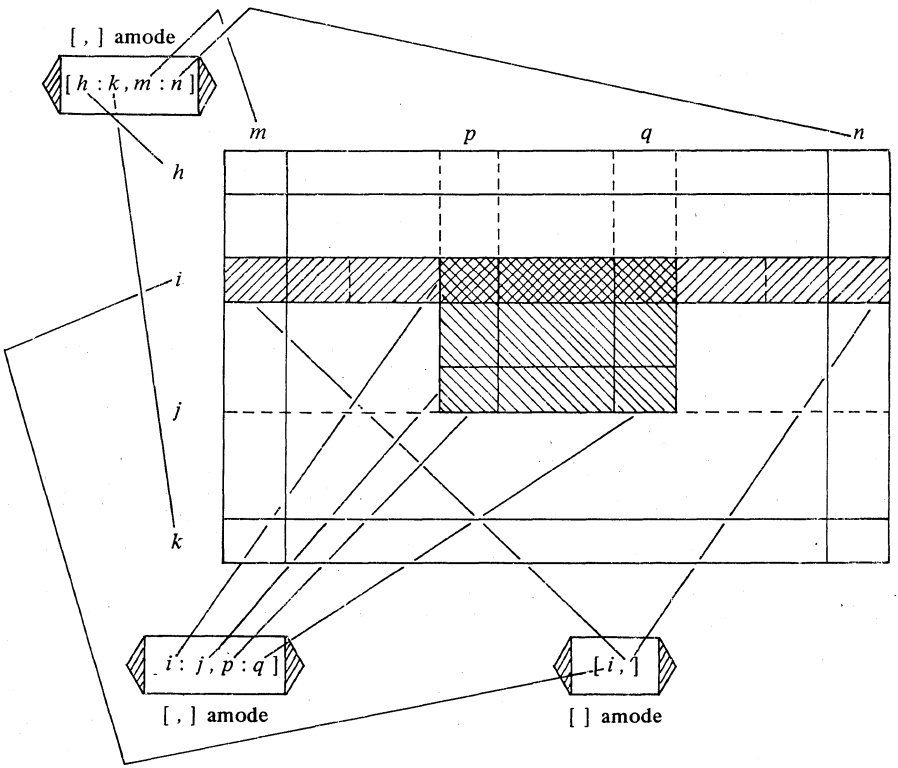
Semantically, an indexer is pretty close to a descriptor. In fact, unless all

its trimscripts are subscripts, it describes a multiple (sub)value in much the same way that a descriptor does. We give two examples:

1.5.2.2. Subscripting



1.5.2.3. Trimming



1.5.3. Identifier declarations for multiples

We could discuss, systematically again, the three fundamental identity-declarations E1, E2 and E3 as elaborated for the general **amode** in Section 1.2, substituting now for **amode** all kinds of 'ROWS of MODE'. An exhaustive discussion would, however, be rather boring without giving substantially new information. We shall, therefore, confine ourselves to a brief survey of many possibilities and a few remarks on mixed matters.

(E3.1) **ref** [ ] **amode** *arow* = **loc** [*m* : *n*] **amode** ;

↑ formal-declarer

↑ actual-declarer

or, as usual:

(E3.1*) [*m* : *n*] **amode** *arow* ;

The identifier *arow* will now yield the name referring to a row of **amodes**, the descriptor of which is [*m* : *n*].

(E3.2) **ref** [,] **amode** *arowrow* = **loc** [*h* : *k* , *m* : *n*] **amode** ;

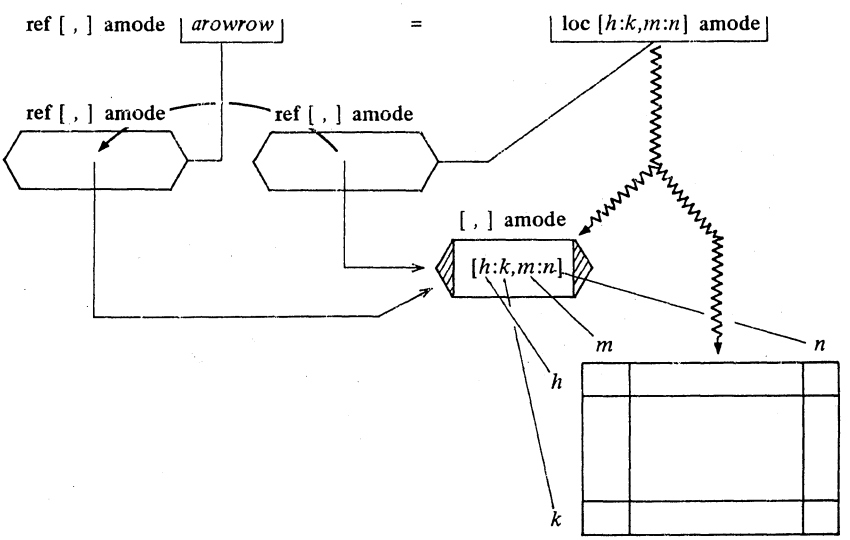
or:

(E3.2*) [*h* : *k* , *m* : *n*] **amode** *arowrow* ;

The identifier *arowrow* will now yield the name referring to a row row of **amodes**, the descriptor of which is [*h* : *k* , *m* : *n*].

In both declarations E3.1 and E3.2 a new multiple value will be generated onto the stack; together with these multiple values, their descriptors will be made.

E3.2 elaborates into:



An example of an E1-type identity-declaration for multiples is:

(E1) [,] **amode** *multipleconstant* = *arowrow* ;

in which the multiple value (and its descriptor) referred to by *arowrow* is ascribed to the identifier *multipleconstant*.

An example of an E2-type identity-declaration for multiples is:

(E2) **ref** [ ] **amode** *namultiple* = *arow* ;

in which the name yielded by *arow* is ascribed to the identifier *namultiple* (which is thus made to refer to the multiple value referred to by *arow*).

By virtue of E1 you cannot assign to *multipleconstant* (being no name) and by virtue of E2 an assignation to *namultiple* results in assigning to *arow*.

namultiple ¢ or *arow* ¢ := *anotherow*

Here, *anotherow* must of course yield a [ ] **amode** value, but it must do more than that—the bounds must match also. The bounds of the location referred to by *arow* (and so by *namultiple*) are [*m* : *n*]. Therefore the bounds of *anotherow* must be [*m* : *n*] also—otherwise it will not fit.

However, we can declare a "flexible" name to which the restriction does not apply:

(E3.3) **ref flex** [] **amode** *arowflex* = **loc flex** [*m* : *k*] **amode** ;

or:

(E3.3*) **flex** [*m* : *k*] **amode** *arowflex;*

arowflex is a flexible name, and its mode is **ref flex** [ ] **amode** (different from the mode of *arow*). Now, when we assign

arowflex := *anotherow*

it does not matter that *anotherow* has *n–m+1* elements and the location referred to by *arowflex* has room for *k–m+1*. It is a flexible location and will be expanded or contracted to suit.

This flexible feature will, however, be an expensive luxury in some implementations. It presupposes a storage allocation regime in which multiples are allowed to "breathe".

1.5.4. Slices

The external object which yields or refers to a subvalue of a multiple is the 'slice'; it consists of an identifier (yielding or referring to a multiple value) followed by an indexer:

arow [*i*]	is an **amode** variable, *arow* is subscripted
arow [*i*:*j*]	is a [ ] **amode** variable, *arow* is trimmed
arowrow [*i*,*j*]	is an **amode** variable, *arowrow* is subscripted

arowrow [*i*,] is a [ ] **amode** variable, *arowrow* is subscripted and trimmed

arowrow [*i*:*j*,*p*:*q*] is a [,] **amode** variable, *arowrow* is trimmed

All these slices yield (sub)names (by virtue of E3.1 and E3.2) and you may assign to them (provided the bounds fit, of course).

arow := *arowrow* [*i*,] ;
arowrow [*i*:*j*,*p*:*q*] := *anotherrowrow* [*h*:*k*,*r*:*s*] ;
arowrow [*i*,] := *arow* ;
arow := *arowrow* [*j*,]

In an assignation like:

arowflex := *arow* [*i* : *j*]

nothing can go wrong, because *arowflex* is flexible. See 5.5.4 for full details of such assignations.

Slices may also turn up in identity-declarations:

(E1.1) [ ] **amode** *rowcopy* = *arowrow* [*i*,]

A copy of the *i*th row of the multiple value referred to by *arowrow* is ascribed to *rowcopy* (this copy of a subvalue has got its own descriptor, which is [*m* : *n*]).

(E1.2) **ref** [ ] **amode** *arowname* = *arowrow* [*i*,]

The name of the *i*th row of the multiple value referred to by *arowrow* (for which subname a new descriptor has been made) is ascribed to *arowname*. The element:

arowrow [*i*,*j*]

may now also be accessed by:

arowname [*j*]

Such identity-declarations are of the utmost importance in situations where you want to have an efficient access to a multiple subvalue. For applications see 8.5.3.

1.5.5. Interrogations

We already mentioned that its descriptor belongs to a multiple (sub)value and that you have a certain access to it. Bounds of a flexible location, for

example, can be changed by assignation (see 5.5.4.1).

In a formal-row-of-MODE-parameter the lower- and upper-bounds do not have to be specified. In that case, you cannot in general know the bounds that will appear in the actual-row-of-MODE-parameter. For example, if you are in a routine, then you cannot know the actual bounds from the formal-parameter(s). For this purpose some standard operators, **lwb** and **upb**, are provided:

1 **lwb** *arowrow*	or	**lwb** *arowrow*	yields the first lower bound
2 **upb** *arowrow*			yields the second upper bound

For further details see 5.5.3 and 6.5.

Vertical readers, please turn to 2.5.

1.6. Unions

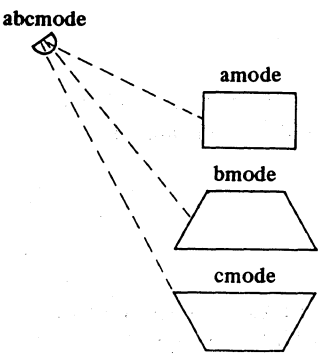
1.6.1. United modes

A MODE may be MOOD or UNITED. Thus far we have considered MOODs. Every MOOD defines a certain class of values. A UNITED mode does not define a new class of values.

If we declare:

(E0) **mode abcmode = union** *(* **amode , bmode , cmode** *);*

then an **abcmode** is either an **amode** value or a **bmode** value or a **cmode** value. There is no such thing as an **abcmode** value. Nevertheless we shall bring an **abcmode** into our pictures in the following way:



In any given situation, one of the dotted lines will be thick (i.e. one of the possible modes in a union will be in force).

Unions "commute" and "associate":

(E0.1) **mode bacmode = union (** **bmode , amode , cmode** **);**

specifies the same united mode as **abcmode**.

(E0.2) **mode abcmodedaemodeuv = union (** **amode , bmode , cmode ,**
union (**dmode , amode , emode ,**
union (**umode ,**
vmode)));

specifies the same united mode as:

(E0.2*) **mode abcdeuvmode = union (** **amode , bmode , cmode ,**
dmode , emode ,
umode , vmode);

Let there be declared:

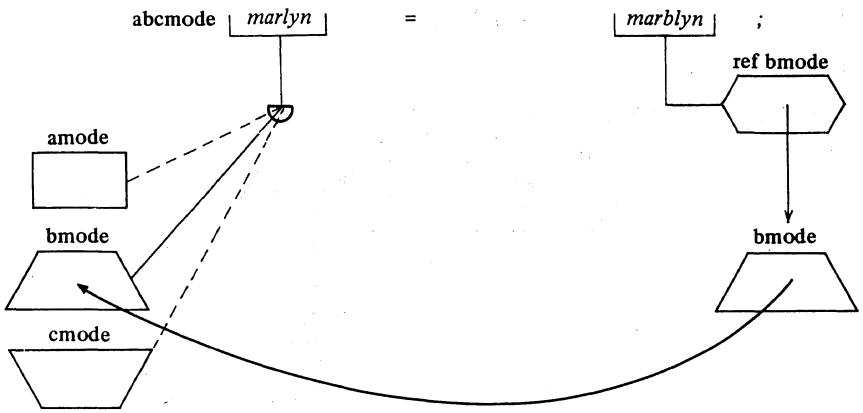
amode *maralyn* , **bmode** *marblyn* , **cmode** *marclyn* ;

1.6.1.1. United constants

Consider the identity-declaration:

(E1) **abcmode** *marlyn* = *marblyn* ;

it elaborates into:



A copy of the **bmode** value referred to by *marblyn* is ascribed to the identifier *marlyn*. Observe that *marlyn* is still an **abcmode** identifier, but she now yields a **bmode** value (we say that **bmode** values are "acceptable" to the mode **abcmode**).

When we declare:

(E1.2) **abcmode** *marlyn* = *maralyn ;*

then she is made to yield an **amode** value, and in:

(E1.3) **abcmode** *marlyn* = *marclyn ;*

she is made to yield a **cmode** value.

In all these cases, *marlyn* is a constant. Strictly speaking *marlyn* is either an **amode** constant, or a **bmode** constant, or a **cmode** constant. So, recalling the fact that *marlyn* is declared to be of united mode, we might term *marlyn* a "united constant". United constants will be of little (if any) use when declared in this way, but these identity-declarations can easily arise when matching an actual-parameter to its formal counterpart in a routine (as in 2.6.2.E8, for example).

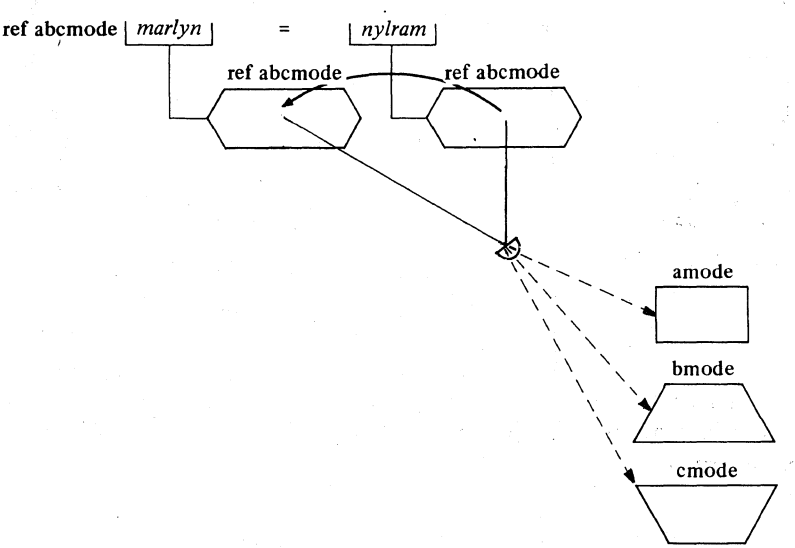
1.6.1.2. Equivalence of unions

Let *nylram* be a **ref abcmode**, i.e. *nylram* yields the name of a **union** *(* **amode** , **bmode** , **cmode** *)*. Although there is no such thing as an **abcmode** value, a **ref abcmode** is a well shaped internal object as are all names; it is simply a name referring to a union.

Consider the identity-declaration:

(E2) **ref abcmode** *marlyn* = *nylram ;*

it elaborates into:



A copy of the name yielded by *nylram*, which refers to an **abcmode**, is ascribed to the identifier *marlyn.* Now *marlyn* and *nylram* both yield names referring to the same union.

But what about the identity-declaration:

(???) **ref abcmode** *marlyn* = *maralyn* (???)

The actual-parameter refers to an **amode** value; the formal-parameter, however, requires a reference to a union. Although in this union there occurs an **amode**, this water is too wide. You can assign an **amode** value to a variable which is united from **amode**; you can never make a reference to such a union refer to a value which occurs in that union. Try drawing the picture, it cannot be done!

1.6.1.3. Local united generation

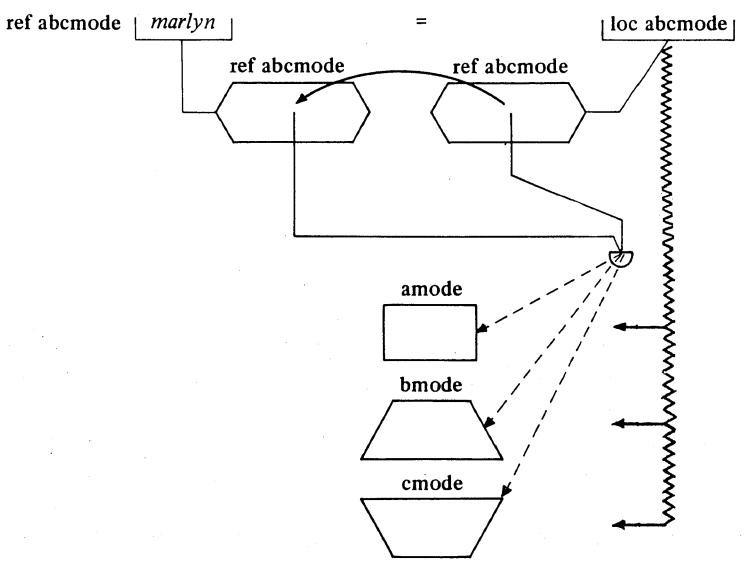
The happening:

(E3) **ref abcmode** *marlyn* = **loc abcmode** *;*

or:

(E3*) **abcmode** *marlyn ;*

can be depicted as follows:



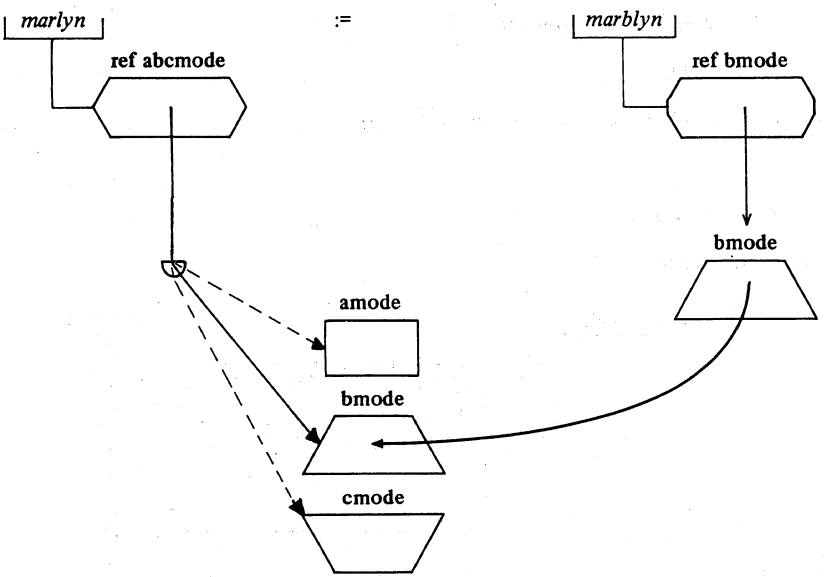
You might ask what value is now generated on the stack, because there is no such a thing as an **abcmode** value. You have to ask your implementor. His answer will be something pretty close to "I reserve sufficient locations for an **amode**, or a **bmode**, or a **cmode** and, for use in conformity-clauses (see 1.6.2), I also reserve space to record which of these is actually in residence".

1.6.2. Assignations and conformity-clauses

To an **abcmode** variable you may assign either an **amode** or a **bmode** or a **cmode** value. For example:

(E4) *marlyn* := *marblyn*

elaborates into:



After this assignation the name yielded by *marlyn* now refers to a **bmode** value. In order to enable you find out which is the mode in force in a union, we have the 'conformity-clause'.

An important application of unions will be found in routines. Suppose you want to switch in a routine depending upon the mode of an actual-parameter when you declared the formal-parameter to be the union of several modes; then you most likely will want to find out (inside the routine) the mode of

the parameter actually supplied, which you may achieve as follows:

(E5)
```
case marlyn in
    (amode) : c do this c ,
    (bmode) : c do that c ,
    (cmode) : c do the other c
esac
```

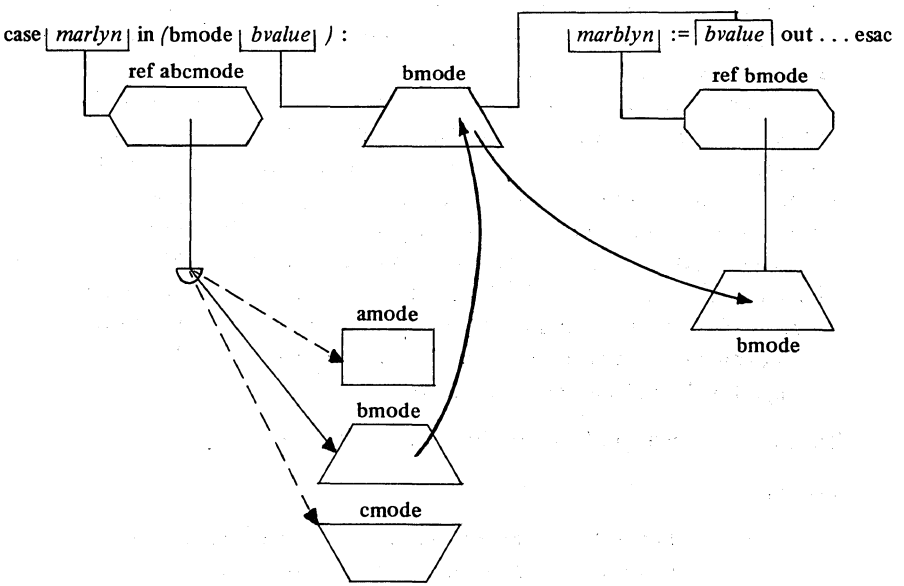
In many cases it will not suffice to find out the mode actually supplied, you may also want to know its value. Now you cannot assign:

marblyn := *marlyn*

not even when the modes conform. The proper tool in such cases is the conformity-clause again:

(E6)
```
case marlyn in
    (bmode bvalue) : marblyn := bvalue
out c marlyn's mode was not bmode; some alternative action can be
      taken here c
esac
```

Here, we have declared a **bmode** identifier *bvalue* to which, provided *marlyn* "conforms" to **bmode**, her **bmode** value can be ascribed. The assignation *marblyn* := *bvalue* is then quite straightforward:



Contrariwise, if *marlyn* does not conform to **bmode**, no assignation takes place and, since no specific alternative modes have been mentioned (as they were in E5), the clause in the **out** part will be taken.

Vertical readers, please turn to 2.6.

1.7. Distinctive features

As we pointed out in 1.2.3, all modes in this language are derived from the primitive modes with the assistance of the symbols **ref** , **proc** , **struct** , "[" and "]", **union**, **long** and **short**. Until now we have not discussed **long** and **short**, and we have not discussed the identity-relation for **ref** modes (names); these are the subject matter of this section.

1.7.1. The **long** and **short** modes

Going down to the level of a concrete computer, the values of all primitive modes (1.2.3) will be mapped into bit-patterns. A **bool** will most likely be a single bit, a **char** may be stored in at least six bits (more likely seven or eight, i.e. a "byte"), an **int** may occupy an entire machineword (a **bits**, see 2.7.1), a **real** one or two machinewords.

On most modern computers you will find provision for (if not in the hardware, then in the standard software) multilength arithmetic. That is to say, apart from an **int** occupying a single machineword, we may also be enabled to add, subtract, multiply and divide integers occupying, say, two machinewords, and maybe even larger ones (occupying three or more machinewords). The same may apply to **reals** and also to the primitive modes **bits** and **bytes** (discussed in 2.7.1), and there may in addition be further versions of all these occupying half, or even a quarter, of a machineword.

To distinguish between the various sizes of such values we have the long-symbol "**long**" and the short-symbol "**short**".

Thus we may distinguish an infinity of different modes

For integers:

int , **long int** , **long long int** , **long long long int** , ---
short int , **short short int** , ---

for real numbers:

real , **long real** , **long long real** , **long long long real** , ---
short real , **short short real** , ---

and similarly for **bits** and **bytes**.

In a specific implementation, only a few of these will in fact be distinguishable as values of different length. The effective number of **longs** or **shorts** is not necessarily the same for **int**, **real**, **bits** and **bytes**. It may be acquired from corresponding environment enquiries (see 6.7.1).

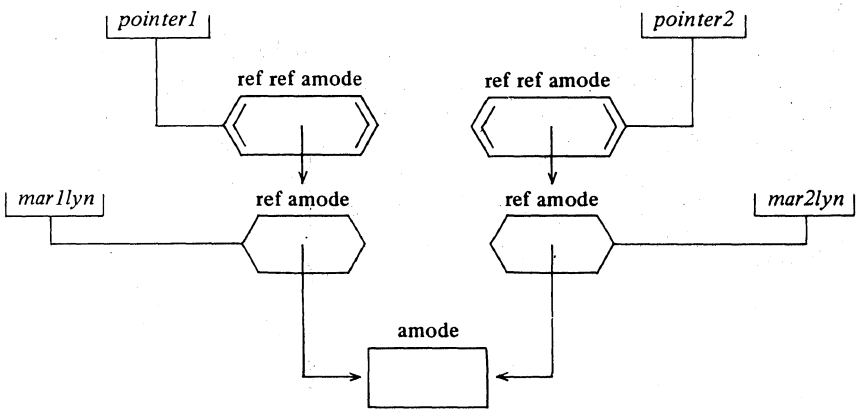
Modes of different **long**th and **short**h derived from the same primitive mode are different modes and it is therefore quite proper to unite from different **long**ths and **short**hs derived from the same (or different) primitives:

```
mode integral = union ( short int , int , long int );
mode number = union ( real , long real , long long real );
```

1.7.2. Identity relations

As was pointed out in 1.2, a 'reference to MODE' value is also an internal object in the computer, i.e. a **ref** will also be mapped into a certain bit-pattern (the address of the value referred to). Consequently, names may also be operated upon and, in particular, compared. To compare names of the same mode we have the identity-relators :=: (or **is**) and :≠: (or **isnt**).

Consider the following picture:



Apparently, the assignations:

pointer1 := *mar1lyn*

and

pointer2 := *mar2lyn*

have been made.

Now:

(E1) *pointer1* :≠: *pointer2* because they refer to different internal objects

but:

(E2) *mar1lyn* :=: *mar2lyn* because they refer to the same internal object.

We may also write:

(E3) **ref amode** *(pointer1)* **is ref amode** *(pointer2)*

which means the same thing.

We might write:

(E4) *pointer1* = *pointer2*

or:

(E5) *mar1lyn* = *mar2lyn*

or even:

(E6) *pointer1* = *mar2lyn*

but, in all of these, the *pointer*s and the *marlyn*s would be dereferenced to yield the **amode**s ultimately referred to (assuming the operator "=" to have been declared for a pair of **amode**s), and it is these that would be compared. It would not then be possible to declare "=" between **ref amode**s or **ref ref amode**s (for otherwise E4-6 would become ambiguous). This is why ":=:" and ":≠:" had to be specially included in the language.

Vertical readers, please turn to 2.7.

2. DECLARATIONS

2.1. Primitive declarations

2.1.1. Primitives

In the previous chapter, we considered values of a hypothetical mode **amode**. In ALGOL 68 there is in principle an infinite number of possible modes which could be substituted for **amode**, and in the course of the present chapter we shall show you how to construct them all. They are, however, all derived from a small number of "primitive" modes (1.2.3). The primitives are as follows:

int The values of this mode are the integers within some finite range dependent upon the implementation (e.g. from -2^{31} to $2^{31}-1$ for a 32 bit binary machine). See 6.2.1 for how to find the size of the range. Arithmetic performed upon **int** values will in general yield exact results, the same in every implementation.

real The values of this mode will in general be held as floating point numbers by the implementation. Thus the range of numbers that can be held is much greater than for **int**, but one pays for this by a restricted precision (again see 6.2.1 for details).

bool There are only two values of this mode, **true** and **false**.

char The values of mode **char** are characters – i.e. internal representations of certain graphic marks on external media. These graphic marks will include at least the letters *a* to *z*, the digits *0* to *9*, ", +, –, *(*, *)*, point, comma and space. Most implementations will add others to this list, and we shall assume, in our examples, that this has been done (and in particular we shall use *A* to *Z* quite freely).

bits, **bytes** The values of these modes are computer words, regarded as a collection of bits or of characters, see 2.7.1.

2.1.2. Variable declarations

Whenever we wish to have, at our disposal, a variable value of some mode, we must declare it, and provide an identifier to yield its name (1.1.1):

(E1) **real** *x;* **int** *i;* **bool** *p;* **char** *c;*

A variable-declaration [R 4.4.1.e] consists of:

a) an 'actual-declarer' (**real**, **int**, etc.) which specifies the mode of the variable value created thereby. For every mode which can be constructed in the language, an actual-declarer can be written. For example, **real**, **ref real**, **ref ref real**, **int**, **ref char**, **bool**, etc. are all perfectly good actual-declarers, and the mode that each specifies is obvious.

b) an identifier (*x*, *i*, etc.) to which the name referring to the newly created variable is to be ascribed.

If we have to declare several variables of the same mode, we have three methods:

(E2) **real** *x;* **real** *y;* **real** *z;* **int** *i;* **int** *j;*
(E3) **real** *x,* **real** *y,* **real** *z,* **int** *i,* **int** *j;*
(E4) **real** *x, y, z,* **int** *i, j;*

E3 and E4 are 'collateral-declarations'. In fact they mean exactly the same thing, but the "contraction" E4 is more convenient to write. The difference between E2 (with ";"s) and E3 (with ","s) is that the declarations take place serially in the first case and collaterally in the second. The difference between them is quite academic in the case of these simple primitives, but could be of crucial importance in more complex situations, such as we shall meet in 2.5. Declarations are always separated from each other, and from other clauses, by go-on-symbols (i.e. ";"s), and for this purpose a collateral-declaration such as E4 counts as one declaration:

(E5) **bool** *p,* **real** *x, y, z,* **int** *i, j;* **char** *c;*

Now that we have declared these variables, we are free to use them:

(E6)
```
x := 3.142;
y := x;
i := 3
```

2.1.3. Sample declarations

In the chapters that follow, we shall give many examples using identifiers such as *x, y, i, j,* etc. To save confusing you, we shall not declare them each time we use them, and so whenever you see such an example, please assume the declarations listed in Appendix 2 to have been already made. You have

already met most of them in Chapter 0 (where they were conspicuously marked with a "D"), and the Report itself also uses most of them in the same way [R 1.1.2].

Vertical readers, please turn to 3.1.

2.2. Identity declarations

Identifiers are declared in 'identifier-declarations' of which there are two kinds – the variable-declaration, which has just been described (2.1.2), and the identity-declaration, which follows.

2.2.1. Identity declarations

An identity-declaration serves to introduce a new identifier, to specify the mode of the internal object (value) that is to be ascribed to it, and to fix that value. Thereafter, until the end of the range (3.2.1) in which that identity-declaration occurs all other occurrences of that identifier are deemed to yield that same value (see 3.2.3 for the precise mechanism of this).

An identity-declaration [R 4.4.1.a] has two sides – its left hand side, or 'formal-parameter', and its right hand side, or 'actual-parameter'. Consider:

(E1) **real** *e* = *2.718281828;*

An identity-declaration is constructed as follows:

Its LHS (the formal-parameter) (**real** *e* in the example) consists of:

a) a 'formal-declarer' (**real**) which simply specifies the mode of the internal object. For every mode which can be constructed in the language, a formal-declarer can be written. For example, **real, ref real, ref ref real, int, ref char, bool,** etc. are all perfectly good formal-declarers, and the mode that each specifies is obvious.

b) an identifier (*e*) to which the object is to be ascribed.

Its RHS (the actual-parameter) (*2.718281828* in the example) consists of:

a unit whose context is strong, and which yields a value whose mode (after coercion if necessary) is the same as that specified by the formal-parameter. This value is now ascribed to the identifier, after which the identifier will always yield that value.

The whole of Chapter 5 is devoted to describing what can and cannot stand as a unit, and to explaining all about the strength of contexts and

coercion, so that it would be inappropriate here to do more than give a few examples that are particularly important. If you have already read 5.1.4.1, you will have noticed the similarity between the rule just given for an actual-parameter and that appropriate to the RHS of an assignation.

An identity-declaration is therefore a very simple concept, with a simple syntax and simple rules. Do not therefore be afraid when you come across a particular example which seems to go on for page after page. It is simply because a long and complicated formal-declarer has been used to specify a long and complicated mode, or because the unit on the RHS happens to be rather a long one. The effect is just the same. We associate together a particular mode and a particular identifier, and ascribe to that identifier a particular value of that mode.

Note, however, that once a value has been ascribed to the identifier, this value cannot be changed (it is a constant (see 1.2.2.1)), and the compiler should be able to take advantage of this. After E1:

(E2) $x := e$

should compile into exactly the same code as:

(E3) $x := 2.718281828$

Things are slightly more complicated with examples like:

(E4) **real** $xy = x \times y$

Here, the value to be ascribed to xy is to be calculated ($x \times y$) at the time when this declaration is encountered during the elaboration of the program, and if it is encountered several times the values will presumably be different on each occasion. Nevertheless, although the compiler will now, presumably, have to reserve a word of store to hold this value, it should still be able to gain some benefit from knowing that it cannot change until the next time.

2.2.2. Another look at variable declarations

(E5) **real** $x;$ or even **loc real** $x;$

We have already seen that **real** is an actual-declarer which creates a **real** variable, and that the name referring to this variable is ascribed to x (making x a **ref real** identifier). So it seems that we have two methods of creating a **ref real** identifier and ascribing a value (more specifically a name) to it. Let us therefore try to construct an identity-declaration to do precisely the same job as the variable-declaration E5. Presumably it will look something like this:

ref real *x* = "something"

In this example, "something" must obviously be a unit that yields a constant value (a name) of mode **ref real**, but it must also have the property that it reserves a space in the store where a **real** value may be put, and it is the name referring to this space that it must yield. If you search through Chapter 5 looking for such a unit, you will find that it is known as a 'generator' and is not described until 5.7.2. This is because the use of generators is rather specialised. There are two kinds of generator, and we recommend the **loc** one for the present purpose (as we have already explained in 1.2.2.3):

(E5*) **ref real** *x* = **loc real**;

loc real is the generator, and the mode it yields is **ref real** in spite of its appearance to the contrary. The **loc** signifies that the **real** value thus created is local to the current range, as will be explained in 3.2.2. **real** is an actual-declarer.

If you feel like trying a **heap** generator, then you should read 2.7.3 first.

Of course, we have already met this before in 1.2.2.3 – E5 means exactly the same as E5*. But please remember the distinction between formal-declarers (such as **ref real** in E5*), which merely specify modes, and actual-declarers (such as **real** in E5 and on the RHS of E5*), which, in addition, generate values of the mode specified. This distinction may not seem important just now, but you will forget it at you peril when you come to 2.5.2 and even in 2.2.3 it will be relevant.

2.2.3. Initialized variable declarations

We explained in 1.2.2.3 how you could write:

(E6) **real** *ee* := *2.718281828*;

which has created a **real** variable and assigned an initial value to it all in one go. But beware! E6 (at least in this representation) looks deceptively like E1 – the difference is just one ":". We can subsequently assign a different value to *ee* (E6), but never to *e* (E1).

Now, if we have the collateral-declaration:

(E7) **real** *x* := *1.0*, **real** *y* := *2.0*;

we may apply our usual contraction to obtain:

(E8) **real** *x* := *1.0*, *y* := *2.0*;

Likewise, if we had had:

(E9) **real** *x* = *1.0*, **real** *y* = *2.0;*

we could have obtained:

(E10) **real** *x* = *1.0*, *y* = *2.0;*

which simply goes to show that formal-declarers (the **reals** in E9) may be gathered together in just the same way as actual ones (the **reals** in E7). But:

(E11) **real** *x* = *1.0*, **real** *y* := *2.0;*

which is perfectly good collateral-declaration, creating a **real** object *x* and a **ref real** object *y*, cannot be contracted to:

real *x* = *1.0*, *y* := *2.0;*

for here we would be gathering together one formal-declarer and one actual one, and moreover it would be too confusing to have the one declarer **real** being used to create objects of two different modes.

Vertical readers, please turn to 3.2.

2.3. Mode declarations

We introduced mode-declarations to you in 1.3.3.1; Consider:

(E1) **mode myproc = proc(real, int, ref char) bool;**

This is a 'mode-declaration'. On the LHS we have introduced the bold word (1.3.2) **myproc** as a 'mode-indication'. On the RHS we have an actual-declarer specifying the required mode. Henceforth, (or at least within this range) **myproc** and **proc (real, int, ref char) bool** may be used interchangeably. You may have declarations such as:

(E2) **myproc** *proc;*
ref myproc *refproc;*

and you may now embark upon the construction of even more elaborate modes such as:

(E3) **proc(myproc) void**

See 2.5.2.2 for mode-declarations of 'row of' modes where bounds must be specified.

Naturally, a mode-declaration may be combined with other declarations (whether they be other mode-declarations, or even identifier- (1.1.2), priority- (4.3.1) or operation- (4.3.2) -declarations) into a collateral-declaration:

(E4) **mode rl = real, mode it = int , mode bo = bool , real** *x;*

to which we may now apply our usual contraction, collecting together all the modes:

(E5) **mode rl = real, it = int , bo = bool , real** *x;*

Vertical readers, please turn to 3.3.

2.4. Declarations of structures

2.4.1. **struct** declarers

The concept of a "structure" was introduced in 1.4.0. Each structured value is of some mode, and for each such mode we can write a declarer (and up to this point, formal- and actual-declarers are still looking the same):

(E1) **struct(real** *first,* **int** *second,* **ref char** *third)*

This is the way you would write a **struct** declarer [R 4.6.1.d]; but should you wish to declaim it in public you would take a deep breath and say*:

"structured-with-(a-)real-field-first-(and-an-)integral-field-second-(and-a-)reference-to-character-field-third-mode"

This is the way in which the Report would specify this mode [R 1.2.1], but in this Introduction we shall stick to the corresponding declarers – they are much cleaner.

first, second, and *third* in E1 are 'field-selectors', not identifiers, and they are a part of the declarer, which identifiers could never be. Thus:

(E2) **struct** *(***real** *fourth,* **int** *fifth,* **ref char** *sixth)*

specifies a different mode from that specified by E1. A value of one could not be assigned to a name referring to a value of the other.

The fields inside a structure can be of any mode whatsoever, including of course other **structs**:

(E3) **struct** *(***proc** *(* **real, int, ref char***)* **bool** *pr,*
struct *(***real** *first,* **int** *second,* **ref char** *third) group)*

Where two adjacent fields are of the same mode, the usual contraction is possible, as in:

(E4) **struct** *(***real** *re, im)*

The only limitation is that a **struct** cannot contain itself (2.4.3), although it can contain a reference to itself. To achieve this, however, we must use a mode-declaration (2.3):

(E5) **mode sequence = struct** *(***int** *object,* **ref sequence** *next);*

Modes such as this are particularly useful in conjunction with **heap** generators

* If some pedant should notice that even this verbosity is not the full story, let him please keep the secret to himself.

(5.7.2.2), and for a substantial example of their use you are referred to 8.7.1.

2.4.2. **struct** declarations

Now we can use **struct** declarers in the formal-parameters of identity-declarations, or as the actual-declarers in variable-declarations:

(E6) **struct** *(***real** *x,* **int** *i) st = (3.14, 123);*

in which a constant *st* is created, and

(E7) **struct** *(***real** *x,* **int** *i) ss;*

in which a variable *ss* is created. *ss* can now be assigned to other variables of the same mode, or its individual fields may be accessed using their selectors:

(E8) *x* := *x* **of** *ss*

This subject will be treated more fully in 5.4.2.

2.4.3. Well-formed modes

The mode-declaration 2.4.1.E5 was circular. However, not all such circular mode-declarations are valid. There are two dangers to avoid:

a) we must avoid modes whose values would occupy an infinite amount of storage space (consider the problems of representing a **large** as specified by **mode large = struct** *(***int** *large,* **large** *larger);*);
b) we must avoid modes which could be strongly coerced into themselves (how many times should you dereference the RHS of **ref itself** *who* = **loc itself**, given that **mode itself = ref itself**?).

Here is how to distinguish the sheep from the goats [R 7.4.1].

Start from the mode-indication on the LHS of the suspect declaration.

Now look through the RHS, marking, or "shielding", each **ref** or **proc** with the word "yin", and each **struct** and each set of parameters of a **proc** with the word "yang".

At each mode-indication you encounter, find the mode-declaration that it identifies (3.3.1), and continue there.

Eventually (because it is circular), you will get back to the mode-declaration which you started from.

So we have:

(E9) **mode sequence = struct** *(***int** *object,* **ref sequence** *next);*
yang yin

and, in a more complicated case:

(E10) **mode a = struct (ref b** *f1*, **union (int, a)** *f2*),
yang yin

mode b = proc (int, int) a;
yin yang

Now consider all routes from your starting mode-indication returning to the same point. Is each route properly shielded by passing through at least one "yin" and one "yang"? If not, the mode is not well formed and the mode-declaration is invalid. E9 passes the test. In E10, there are two routes from **a** back to **a** again. One passes by way of "yang-yin-yin-yang" but the other can only manage "yang". It can be shown that a missing "yin" will land you in danger (a) above, and a missing "yang" will put you in danger (b). Keep your yin and yang in the correct balance and you will attain harmony.

2.4.4. The mode **compl**

The mode **compl** (for complex) is not a primitive in the language, although you would not come to much harm if you were to regard it as such, since it is provided with a complete set of operators and other useful facilities (5.4.0 and 5.4.3). It is, in fact, a **struct**, being declared in the standard-prelude (1.1) [R 10.2.2.f] by:

(E11) **mode compl = struct (real** *re, im);*

Vertical readers, please turn to 3.4.

2.5. Declarations of multiples

2.5.1. Row declarers

The concept of a "multiple value" was introduced in 1.5.1. Each multiple value is of some mode, and for each such mode we can write a declarer. Now, however, we are at the point where formal- and actual-declarers begin to look different. Here is a formal-declarer:

(E1) [, ,] **ref real**

which is pronounced:

'row row row of reference to real'

This specifies the mode of a multiple value which needs three subscripts (because there are two ","s between the "[" add the "]"), and whose elements are names of mode **ref real**.

Here is an example of a formal-declarer whose interest lies in its complexity, rather than in any use it might have:

(E2) [ ] **struct** *(***proc** *(***int**, **ref** [ ] real*)* [ ] **real** *p*,
[,] [ ] **ref compl** *q)*

The pronunciation of this one is left to the proverbial student as his proverbial exercise. You have enough information to do it, but have you the stamina? However, this example does show that we may have rows of **struct**s and of other rows, **struct**s containing rows, and **proc**s that use and yield rowed modes. Observe the difference between [,] [ ] **real** ('row row of row of real') and [,,] **real** ('row row row of real'). The first is a doubly subscripted multiple each element of which is a singly subscripted one. The second is a straightforward triply subscripted multiple value.

2.5.2. Row declarations

In a variable-declaration involving a 'row of' mode we encounter a problem that did not arise before. A multiple value consists (1.5.1) of a descriptor and a set of elements, and whenever we create such a value, not only must it be of the required mode, but its descriptor must fit our requirements as well. It is the responsibility of the actual-declarer to ensure that both these requirements are met.

2.5.2.1. Fixed and flexible names

A multiple value is simply a row of elements together with a descriptor, and its mode is something like [,] **real**. Observe that the mode tells you the number of subscripts, but not the number of elements. So, you may ask, if I have a multiple value with 100 elements, may I supersede it with another one of 200 elements (but of the same mode, of course)?

The answer is that "it all depends". Superseding takes place during assignation (1.1.2.2), in which the value superseded is that referred to by the name on the LHS. It is the name which controls the location where the elements are kept, and so it is the name which determines whether the location is flexible enough to accommodate the greater number of elements. If the mode of the name is **ref flex** [,] **real**, well and good, but if its mode is only **ref** [,] **real** the assignation will not be allowed.

Thus we have the situation (which only arises with multiple values) that a value may be referred to by either a "flexible name" (with a **flex** after the **ref** in its mode) or a "fixed name" (without a **flex**). Nevertheless, the mode of the value itself is the same in either case. There is no such mode as **flex** [,] **real**.

Suppose that *rowvar* yields a flexible name of mode **ref flex** [ ] **real**. You will see presently that *rowvar* could have been declared as follows:

flex [*1* : *0*] **real** *rowvar* := **skip**;

where the **skip** is to signify that we do not at the moment have the slightest idea what size it is, or what are the values of its elements.

Here now is an identity-declaration:

(E3) [ ] **real** *xl* = *rowvar*;

Since *xl* is not a name, no question of flexibility arises. *xl* now simply yields the multiple value obtained from *rowvar*, with whatever number of elements that had.

Here is another identity-declaration:

(E4) **ref flex** [ ] **real** *xlm* = *rowvar*;

Now *xlm* yields the same flexible name as *rowvar*, and refers to the same location in store. The **flex** was necessary in order to match the mode of *rowvar*. If we have a fixed name *rowfix* of mode **ref** [ ] **real** declared by:

[*1* : *10*] **real** *rowfix*;

then the corresponding declarations are:

(E5) [ ] **real** *xf* = *rowfix*;

(E6) **ref** [ ] **real** *xfg = rowfix;*

2.5.2.2. Actual 'row of' declarers

A 'ROWS of' variable-declaration contains, of course, an actual 'ROWS of' declarer. This has to reserve a substantial region of store, and so it must know how much store to reserve, and whether it is likely to be changed later (through being flexible). This information is provided by bounds in the actual-declarer:

(E7) [*1 : 99*] **real** *xfgh;*

(E8) **flex** [*1 : 99*] **real** *xlmn;*

Space for *99* **real**s is reserved, with the option of altering it later in the second case, and arrangements are made to release the space again when the current range is left.

Since an actual-declarer is actually going to reserve some actual store, of some actual size, it follows that the bounds must be actually present. If the upper-bound is less than the lower-bound, then the descriptor is "flat" and the number of elements in the multiple is taken as zero (the bounds [*1 : 0*] are frequently used when an initially empty multiple is to be created, as in 2.5.E11 below. If the lower-bound is 1, it may be omitted as in

(E7*) [*99*] **real** *xfgh;*

A bound can be any meek **int** unit, and you will see in Chapter 5 that this covers a large number of possibilities. All the bounds in the actual-declarer, together with the RHS if the declaration is initialized (2.2.3), are elaborated collaterally each time that the variable-declaration is encountered [R 4.4.2.b].

Also, since a variable-declaration creates a name whose mode has a **ref** in it (2.2.2) (even though this **ref** does not appear in the actual-declarer), it follows that an actual-declarer may start with a **flex** (as in E8) if the name created is to be a flexible name. This may seem more natural if E8 is written in its alternative form:

(E8*) **loc flex** [*1 : 99*] **real** *xlmn;*

Next you might like to reflect upon the fact that the RHS of a mode-declaration (2.3) is an actual-declarer, so that not only may a mode-indication be made to specify a mode, but it then also specifies bounds and **flex**, where relevant, as well. A splendid example of this is the mode **string** declared in

2.5.3. If such a mode-indication now appears as or in a formal-declarer, no harm is done, the actual-bounds and any **flex** associated with it simply being ignored.

However, if the bounds of a mode-declaration require elaboration, as *n* in:

(E9) **mode a** = [1 : *n*] **real**;

then they are not elaborated at the time this mode-declaration is encountered. Instead, it is the value of *n* in force at the time **a** is applied that matters:

(E10)
```
n := 1;
a a;      ¢ i.e. [1 : 1] real a; ¢
n := 2;
a b;      ¢ i.e. [1 : 2] real b; ¢
```

2.5.2.3. Summary

Let us now summarize the differences between actual- and formal-declarers: A 'row of' actual-declarer must have bounds and may start with **flex**:

[*1* : *99*] **real** **flex** [*1* : *99*] **real**

moreover, if an actual-declarer specifies a **struct** mode, then its fields are also actual-declarers:

struct (*[1 : 99]* **real** *a,* **struct** (**int** *c,* **flex** [*1* : *99*] **real** *d) b)*

A formal-declarer never has any bounds, and never starts with a **flex**:

[ ] **real**
struct (*[ ]* **real** *a,* **struct** (**int** *c,* [ ] **real** *d) b)*

However, if it starts with a **ref**, or has a **ref** anywhere within it, then **flex** may occur in parts of the declarer controlled by that **ref**:

ref [ ] **real** **ref flex** [ ] **real**
ref struct (*[ ]* **real** *a,* **struct** (**int** *c,* **flex** [ ] **real** *d) b)*
ref [ ] **flex** [ ] **real**
ref ref flex [ ] **real**

Moreover, certain declarers are always constructed like formal ones, even if they occur as or inside actual ones. These are declarers beginning with **ref**, **proc** or **union**, so that the following are all correctly formed actual-declarers:

ref [ ] **real** [*1* : *99*] **ref flex** [ ] **real**

struct *(*[*1* : *99*] **real** *a*, **ref struct** *(***int** *c*, **flex** [ ] **real** *d) b)*
proc *(*[ ] **real***)* [ ] **real**
union *(*[ ] **real**, [ ] **int***)*

2.5.3. The mode **string**

The mode **string** is not a primitive in the language, although you would not come to much harm if you were to regard it as such, since it is provided with a complete set of operators and other useful facilities (5.5.1.1, 5.7.0.2, 6.1). It is, in fact, a [ ] **char**, being defined in the standard-prelude (1.1) [R 10.2.2.i] by:

(E11) **mode string** = **flex** [*1* : *0*] **char**;

An interesting consequence of this is that if we declare:

(E12) **string** *t*;

we have not created an object with an undefined value as we would have done in:

real *x*;

Instead, *t* has been made to refer to an empty **string**, which is a very definite (and useful) entity, and the only thing undefined about it is the value of the elements which it hasn't got.

Because **flex** is only meaningful after a **ref**, the mode **string** is the same as the mode [ ] **char**, but the mode **ref string** is equivalent to **ref flex** [ ] **char**.

Vertical readers, please turn to 3.5.

2.6. Union declarations

2.6.1. **union** declarers

We introduced you to **unions** in 1.6.1. Although we cannot create values of united modes, we can talk about such modes, and to do this we need declarers:

(E1) **union** *(*[ ] **real**, [ ] **int***)*

Note how the inside of a **union** is always formal (2.5.2.3), so that there is no difference in appearance between formal and actual **union** declarers [R 4.6.1.s].

You will remember that (1.6.1) the order in which the modes are specified inside a **union** is quite immaterial [R 4.6.1.s, R 7.3.1.k], so that:

(E2) **union** *(*[ ] **int**, [ ] **real***)*

specifies exactly the same mode as that specified by E1. Moreover:

(E3) **union** *(***int**, **string**, **union** *(***real**, **union** *(*[ ] **char**, **int***)))*

could equally well (and with less ink) have been:

(E4) **union** *(***int**, **real**, [ ] **char***)*

However:

union *(***int**, **ref int***)*

is not allowed because, if this mode were required (a posteriori) in a firm context (e.g. as the operand in a formula), and a **ref int** were available (a priori), we should not know whether to dereference it and then unite it (5.6.0), or whether to unite it straight away. It is therefore forbidden for a component mode of a **union** to be firmly coercible to one of the other component modes or to the union of those others [R 4.7.1.f]. Thus:

union *(***ref union** *(***int**, **real***)*, **int**, **real***)*

would not be correct either, because **ref union** *(***int**, **real***)* can be firmly coerced to **union** *(***int**, **real***)*.

2.6.2. **union** declarations

Now we can use **union** declarers in the formal-parameters of identity-declarations, or as the actual-declarers in variable-declarations:

(E5) **union** *(***real, int***) ir = (p | 2 | 3.14);*

in which a constant *ir* is created (either **int** or **real** depending on the yield of *p*), and

(E6) **union** *(***real, int***) ri;*

in which a variable *ri* is created, to which either a **real** or an **int** may subsequently be assigned (5.6.0). At the moment, it is not defined whether *ri* refers to a **real** or an **int**, but it will certainly be one of them [R 4.6.2.a].

Since there is no such thing as a value of a united mode, there are some declarations which, whilst being legal, are not at all useful:

(E7) **union** *(***bool, real***) br = 3.142;*

br will now, in fact, always yield a **real** value, but wherever it is used allowance for both possibilities will nevertheless be made (and, for example, *x* := *br* will not be allowed). This declaration could, however, very reasonably arise when matching the actual-parameter of a call to the formal-parameter of a routine-text (1.2.3.2.1):

(E8) **proc** *pbr = (***union** *(***bool, real***) br)*: XXXXX;
pbr (3.142)

Here, *pbr* yields a routine which is prepared to accept either a **bool** or a **real** as its actual-parameter (and occurrences of *br* within XXXXX will be treated accordingly).

Vertical readers, please turn to 3.6.

2.7 bits, bytes, longs and shorts

2.7.1. bits and bytes

bits and **bytes** [R 10.2.2.g,h] are two primitive modes which are intended to give you access to the actual words in your computer, so that you may achieve greater efficiency. **bits** is similar to [ ] **bool** and **bytes** to [ ] **char** (or **string**), except that the number of **bools** or **chars** respectively is limited to exactly that number which can be fitted into one computer word. Thus individual **bits** or **bytes** values can be passed around inside your program with great efficiency, at the expense of some additional effort (by widening (5.7.0.2) or the procedures *bitspack* and *bytespack* (6.2.2) and the operator **elem** (6.1.2)) whenever you want to get at the individual **bools** or **chars** within them. Environment enquiries are provided (6.2.1) to tell you how much you can get into a single **bits** or **bytes** in your implementation.

bits and **bytes** are, of course, easily declared:

(E1) **bits** *bits;* **bytes** *bytes;*

Note that [ ] **bits** and [ ] **bytes** may be declared and **bits** and **bytes** may appear inside **struct**, **union** and **proc** modes.

2.7.2. long and short modes

Double, triple, etc. length working is used in computers in order to obtain greater accuracy, or to distinguish between a greater number of possible values of some mode, or to pack more information into one value. The ALGOL 68 modes where this facility would be useful are:

int, **real**, **compl**, **bits** and **bytes**

Indeed, it is possible to prefix all of these modes by "**long**" in order to obtain new modes of approximately double the precision, by "**long long**" for triple precision, and so on. A given implementation does not have to carry this on indefinitely, however. After some number of **longs** (perhaps only one) it will treat values of **longer** modes as being of the same precision. Various environment enquiries are provided to tell you how many **longs** are effective, and how precise they are (6.7.1), and a full set of operators (6.7.3) and procedures (6.7.2) is provided for them.

long modes are, of course, easily declared:

(E2) **long real** *reaeal;* **long long int** *iiiiiiint;*
proc *(***long int, int***)* **long long int** *power;*

In the same manner, these modes may be prefixed by "**short**" in order to take advantage of any facilities in the hardware for manipulating half words or individual bytes. Again, after some number of **shorts** (perhaps none at all) further **shorts** will make no further difference to the precision. Appropriate environment enquiries, operators and procedures are provided as before.

(E3) **short real** *rel;* **short short int** *it;*
proc *(***short int, int***)* **short short int** *root;*

2.7.3. **heap** declarations

Just as:

(E4) **real** *x;*

which may also be written as:

(E4*) **loc real** *x;*

means the same thing (2.2.2) as:

(E5) **ref real** *x* = **loc real**;

so:

(E6) **heap real** *x;*

means the same as:

(E7) **ref real** *x* = **heap real**;

in which the **heap real** is a **heap** generator. The effect of E6 is to reserve a space in the store for the variable *x* which will not disappear when the current range is left. (It will not, of course, then be accessible via the identifier *x*, but its name may in the meantime have been assigned to a **ref real** variable with a larger scope).

This and other uses of **heap** generators will be described more fully in 5.7.2.2.

Vertical readers, please turn to 3.7.

3. CLAUSES

3.1 Serial clauses

A particular-program (1.1) [R 10.1.1.g] consists of an ENCLOSED-clause (3.2.4), which is usually a 'serial-clause' enclosed by embedding between **begin** and **end** (or, if you prefer, between "(" and ")", which can be used as alternatives wherever **begin** and **end** may occur).

The bricks out of which a serial-clause is constructed are called 'declarations', 'statements' and 'expressions'. Declarations we have already met (1.1.3 and 2). 'Statements' and 'expressions' [R 3.0.1.b, c] are alternative names for 'void-units' and 'MODE-units' respectively, and units in general will be discussed in Chapter 5. In the meantime, it will suffice to say that:

x := *a*+*b*

is a statement (usually) and:

a+*b*

is an expression (likewise).

We shall also need 'go-on-symbols' (better known as semicolons) which constitute the mortar which bind the bricks together, and 'labels' which enable us to find our way around.

3.1.1. The declarations

The building rules are really quite simple [R 3.2.1]. The foundations, which come first, consist of declarations (as many as you like) with mortar in between:

(E1) **begin**
¢ the **begin** is not part of the serial clause proper; it is the earth in which the foundations are embedded ¢
real *a;*
int *i;*
char *c,d,e;*

The last one is a collateral-declaration (see 1.1.3) meaning the same as:

(E2) **char** *c*, **char** *d*, **char** *e;*

however, it all counts as one brick for our present purpose.

Statements are also allowed within the foundations (but not labels). This is particularly useful when you are declaring multiple values (as described in 2.5.2) in which the bounds are first to be calculated:

(E3) **begin**
int *i;*
read (i); ¢ (7.1.2) ¢
[*1* : *i*] **real** *x1;* ¢ declares a multiple with bounds *1* to *i* ¢

Note that it is perfectly possible for there to be no foundations at all, the building starting straight away with the walls. This would be rather unusual for the serial-clause which constituted the body of a particular-program, but there are plenty of other places where such serial-clauses could occur.

3.1.2. The statements

The walls come next, and these too may be entirely absent. They consist of statements and semicolons, with labels attached where required (a label always comes before a statement (or before an expression, or even before the whole particular-program) and consists of an identifier (1.1.2) followed by a colon).

(E4) **begin**
int *i;* **real** *x, y, z;*
comment those were the foundations: now for the walls **comment**
z := *1* − *3* × *sqrt (small real);* ¢ for *small real* see 6.2.1 ¢
labl: *read (i);*
x := *i;*
lobl:
lubl: *y* := *i*/*x*↑*2;*
x := *(2* × *x* + *y)*/*3;*
if *y*/*x* < *z* **then go to** *lobl* **fi;**
print (x); ¢ (7.1.1) ¢
go to *labl;*

This will compute and print the cube roots of the (nonzero) integers read in. The conditional statement (**if** ..... **fi**) does what you would expect it to do (see 3.2.4.2 for details).

We now observe (as you have doubtless guessed already) that when a statement is followed by a ";" the completion of the elaboration of that statement is followed by the initiation of the elaboration of the following statement. Only when we come to a **go to** statement (consisting of **go to**

followed by a label-identifier, or alternatively of just the label-identifier) is this sequence broken. Note that several labels can precede one statement (as *lobl*: and *lubl*: in E4).

3.1.3. The yield

Finally, we come to the roof. This consists of just one statement (void-unit) or one expression (MODE-unit) (and there may be some labels before it).

```
(E5) begin
         int i; read (i); real x, y, z;
             ¢ those were the foundations ¢
         z := 1 – 3 × sqrt (small real); x := i;
         lobl: y := i/x↑2; x := (2 × x + y)/3;
             if y/x < z then go to lobl fi;
             ¢ those were the walls ¢
         print (x)
     end ¢ the end is part of the embedding, too ¢
```

In this case the roof (*print (x)*) was a statement – it was, in fact, just the last statement of the clause, and if there had been any more following it would have been quite content ro be part of the walls.

Note that, in accordance with the best building practice, there is no mortar underneath the foundations, and there is none on top of the roof. Also, there is exactly one ";" between each brick. Contrast this with ALGOL 60 where extra ";"s mostly did no harm.

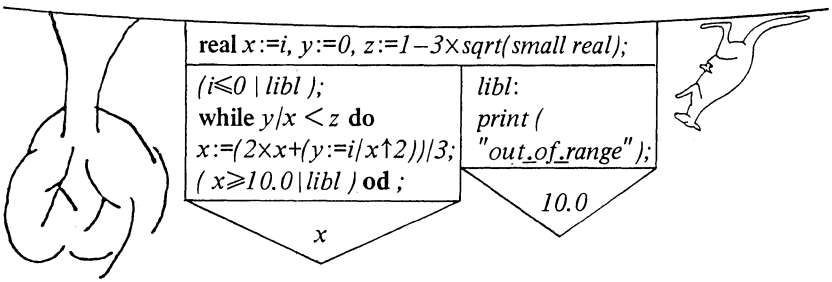
Now, if the roof is an expression, then it must yield a value. What happens to this value?

```
(E6)     a := b + (real x, y, z; z := 1 – 3 × sqrt (small real); x := i;
                   lobl: y := i/x↑2; x := (2 × x + y)/3;
                       if y/x < z then go to lobl fi;
                   x)
```

Here the piece between the "(" and the ")" is indeed a serial-clause and it occurs in a place where it is expected to yield a value (it is in fact a real-serial-clause). *x* is its roof and is an expression which (after a little dereferencing – see 1.1.6) yields a **real** value. This value now becomes the value of the serial-clause as a whole, and in due course it gets added to *b*, and the result is put into *a*.

3.1.4. Completers

We will now consider buildings with several roofs. Suppose, in the example E6, we only wanted the cube root of *i* if *i* was positive and the cube root was less than *10.0* (actually, it was rather a poor way of finding cube roots for largish numbers anyway). In the other cases, we wanted to print a message and to yield the result *10.0* regardless. Then we could build a house like this:



We have to make it an Australian house, so that you can read the program from the top downwards. We also included one or two short cuts in the program, which you might be able to follow.

Observe that the foundations are common to both roofs. The complete statement containing the two-roofed serial-clause (and without the short cuts) will now look like this:

(E7)
```
a := b + (real x, y, z; z := 1 – 3 × sqrt(small real); x := i;
          if x ⩽ 0 then go to libl fi;
   lobl:  y := i/x↑2; x := (2 × x + y)/3;
          if y/x < z then go to lobl fi;
          if x ⩾ 10.0 then go to libl fi;
          x exit
   libl:  print ("out of range");
          10.0)
```

exit means that we have come to the first roof, and if this point is reached during the elaboration, then *x* is the value of the serial-clause. Otherwise (there being no more **exits** in this particular clause) *10.0* is the roof and provides the value. Inevitably, the **exit** must by followed by a label (for how else could the following statement be reached), and so the **exit** with its label attached constitute what is known as a 'completer' and the process of leaving a serial-clause through the roof (i.e. via either the *x* or the *10.0* in the E7

example) is known as "completing". Contrariwise, if you jump out of the middle of a serial-clause by means of a **go to** (out of the window perhaps) then that is to "terminate" it, and in this case no value is yielded. The mode of the value yielded on completing may be coerced and balanced (5.2.0.1) according to the context in which the serial-clause as a whole appears.

You can try imagining how E7 could have been written without the **exit** facility. It would have been necessary to re-arrange it so that the x to be yielded came right at the end. After *libl,* it would have been necessary to assign *10.0* to x, and then to **go to** another label just before the final x.

Note that, syntactically, a completer is a special type of mortar, so one does not expect to see any ";"s either before or after it. It might be tempting to regard it as a statement meaning "and now **go to** the end of the clause" but this would be dangerous since you must be quite sure first that you know which clause it will go to the end of.

3.1.5. Delimiters

In all the examples given above, the serial-clause itself was the part between the **begin** and the **end** (or "*(*" and "*)*"). However, serial-clauses can occur in a variety of contexts, and the complete list of delimiter pairs applicable is as follows:

between	**begin**	and	**end**	(in particular-programs or closed-clauses)
	(	and	*)*	,,
	***if**	and	**then**	(in conditional-clauses)
	then	and	**fi** / **else** / **elif**	,,
	***elif**	and	**then**	,,
	else	and	**fi**	,,
	***case**	and	**in**	(in case- and conformity-clauses)
	***ouse**	and	**in**	,,
	out	and	**esac**	,,
	***while**	and	**do**	(in loop-clauses)
	do	and	**od**	,,

* Strictly, the clause between these delimiters is an enquiry-clause rather than a serial-clause – see 3.2.4.2.

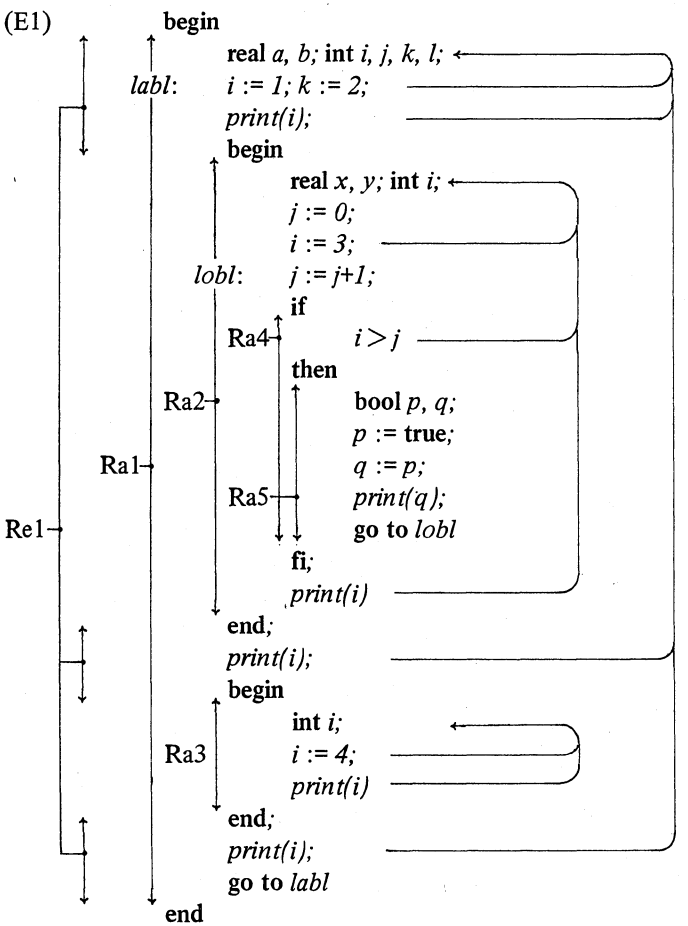
Closed- conditional- and case-clauses will be discussed in 3.2 and conformity-clauses in 3.6.

Vertical readers, please turn to 4.1.

3.2 Closed clauses

3.2.1. Ranges and reaches

A 'range' [R 3.0.1.f] is a piece of source text which constitutes a serial-clause (or one which constitutes a routine-text (see 4.2.2.1) or certain portions of choice-clauses and loop-clauses (3.2.4, 3.5.2)). A range can "contain" further ranges within itself, and so on recursively. Here is a (not very sensible) particular-program, with all its ranges marked and numbered:



A "reach" [R 3.0.2] is a range, with the exclusion of all the ranges

contained within it. Thus Re1 in the above example is a reach.

Note how the ranges in E1 are mostly serial-clauses contained between the delimiters listed in 3.1.5.

3.2.2. Scopes of names

Consider the declaration **real** *x* at the head of Ra2, which could also have been written as:

(E2) **loc real** *x*

When this is reached during elaboration of the program (i.e. just after the beginning of Ra2), a name is created (a location in the store is reserved) and is ascribed to the identifier *x* [R 4.4.2.b]. The **loc** in E2 means that the "scope" of this name is local, i.e. restricted to the lifetime of the range Ra2 (in whose declarations it occurs). Therefore, as soon as we cease elaborating statements within Ra2 (when we reach the delimiter **end** which terminates it in this instance), the name (to all intents and purposes) ceases to exist (the location in store is relinquished).

Thus, anything that you declare at the head of a range is only available to you inside that range (this is of course exactly the same as in ALGOL 60, except that there they are termed "blocks" instead of "ranges").

In an assignation (5.1.4.1) the scope of the RHS must be at least as old as that of the LHS, for otherwise the value referred to by the LHS would be undefined in some reach:

```
(E3)    begin ref real xx; real y;
            begin real x;
            read (x); ¢ reads a real value (2.0, perhaps) ¢
            y := x; ¢ this is all right because the real value is being as-
                      signed, and its scope is not limited ¢
            xx := x ¢ this one is going to cause trouble ¢
            end;
        print(y); ¢ no complaints ¢
        y := xx;
        print(y) ¢ now what? ¢
        end
```

In this example, the name *x* is newer in scope than the names *xx* and *y* (newer in the sense that it was created later than them and will disappear sooner). Thus the assignation *xx* := *x*, in which the name *x* was supposed to be assigned, was illegal.

3.2.3. Identification

Identifiers may occur in declarations – these are called "defining-identifiers". In all other places (in assignations to take the most obvious example) they are "applied-identifiers". Consider the range Ra5 in E1. The identifier *p* is defined in:

(E4) **bool** *p*

and is applied in:

(E5) *p* := **true** and in *q* := *p*

Now it is up to the compiler to correlate each applied-identifier with a defining one, and when this has been done the former is said to "identify" or to be "within the reach of" the latter. The two occurrences of *p* in E5 thus identify the *p* in E4 (or E5 is within the reach of E4), still in the context of the range Ra5, of course, and this means that all these *p*s yield the same name (they get hold of the same location in the store) and they are all of the same mode (**ref bool** in this case).

Now, what about the identifier *i*, which has defining occurrences at the head of Ra1, Ra2 and Ra3 in E1, and applied occurrences (notably as *print(i)*) all over the place? The rule is quite straightforward [R 7.2]:

Start at the applied occurrence in question (call it "A")
Look for a defining occurrence in the same reach as A (call it "B")
If none is found,
→then look for a defining occurrence in the reach which is immediately outside the range which contains the reach which you have just been looking at (call it "B")

If none is found, ←

A then identifies B.

Thus the "reach" of a particular defining-identifier is the range in which it is declared with the exclusion of all inner ranges within which it is re-declared. Note that, in some other languages, the term "scope" is used with this static meaning. In ALGOL 68, however, "scope" has another, dynamic meaning (3.2.2).

The arrows on the right hand side of example E1 show how all the applied occurrences of *i* are identified, and if you follow through the elaboration of this particular-program, you will find that what it prints out is:

```
1 T T 3 1 4 1 1 T T 3 1 4 1 1 T ... .
```

where T is printed for the value **true** (see 7.1.1).

Note that, each time a new defining occurrence of *i* is encountered, then, until further notice, a new name is ascribed to *i*. This certainly does not mean that the old name yielded by *i* ceases to exist. It simply goes underground, and cannot be accessed (at least not via *i*) until after the end of the range in which *i* was redefined. It can, of course, be accessed if the programmer has made provision for some other object to yield or to refer to it.

Labels are identified just like any other identifiers, being defined as in:

labl:

and applied as in:

go to *labl*

This means that the **go to** *lobl* at the end of Ra5 can be used to jump out of Ra5 (which is then terminated – 3.1.4) into Ra2, which contains it; but it would be quite impossible to jump into Ra2 at *lobl*: from anywhere in the reach Re1. The identification just would not work. Thus, a range can only be entered via its declarations (which is just as well, if you think about it).

Finally, as you might expect, each applied-identifier must identify one, and only one, defining-identifier [R 7.2.1]. Thus all variables which you use must be declared (as in ALGOL 60, but not as in FORTRAN) and any given identifier may only be declared once within a reach. However, an applied-identifier need not necessarily come after its defining-identifier:

(E6)
```
begin
    proc a = real: b := c;
    real b := 1, c := 2;
    x := a
end
```

is perfectly legitimate (see 4.2.2.1 for further details of routine-texts, of which **real**: *b* := *c* is one). On the other hand:

(E7)
```
begin
    real b;
    b := c;
    real c;
    c := b
end
```

is syntactically correct, and the identification of *c* works, but the assignation *b* := *c* will not work because no name has been ascribed to *c* at this point of

the elaboration (indeed, no such name has even been generated).

For the identification of modes see 3.3.1, and for operators see 4.3.3.

3.2.4. ENCLOSED clauses

An ENCLOSED-clause is either:

a closed-clause (3.2.4.1)
a collateral-clause (3.7.1)
a parallel-clause (3.7.2)
a structure-display (3.4)
a row-display (3.5.1)
a conditional-clause (3.2.4.2)
a case-clause (3.2.4.3)
a conformity-clause (3.6)
} these 3 being known collectively as choice-clauses
or a loop-clause (3.5.2)

ENCLOSED-clauses occur primarily in primaries (5.1.0.1). This means, inter alia, that they can stand as statements or as expressions (yielding respectively either **void** or some mode).

3.2.4.1. Closed clauses

A closed-clause is a serial-clause enclosed between **begin** and **end**, or between "*(*" and "*)*" [R 3.1]. There are two chief reasons for using them. The first is to create some variables (strictly names) which are to be local to some range:

(E8)
```
begin
real pie;
      begin
      real w := 0, int i := 1; real z = sqrt (small real/2);
loop: w := w + 2/(i x (i + 2));
      i := i + 4;
      if 1/i > z then go to loop fi;
      pie := 4 x w
      end;
print(pie)
end
```

Here, the closed-clause was a statement, and it was created because *w*, *i*, *z* and

loop were not needed outside it. Here is a similar example in which the closed-clause is an expression:

(E9)
```
begin
print (4 × ( real w := 0, int i := 1; real z = sqrt (small real/2);
             loop: w := w + 2/(i × (i + 2)); i := i + 4;
             if 1/i>z then loop fi;
             w ) )
end
```

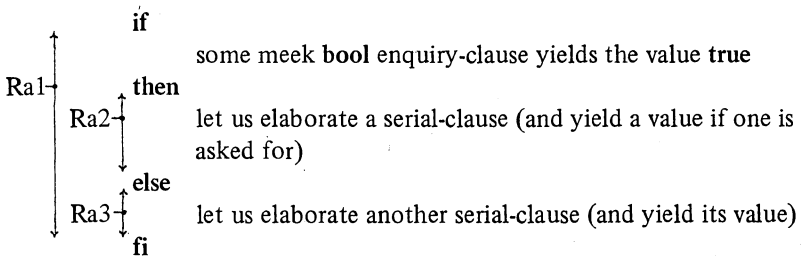
Indeed, 99% of an entire particular-program could be contained within one such expression. Note here the alternative form of the **go to** statement in which the "**go to**" is omitted.

The second reason why closed-clauses are used is to alter the priority of operators in formulas (5.1.3):

(E10) $y := x \times (a+b)$

In these cases, the serial-clause inside the closed-clause often contains just one unit (the building is nothing but a roof, using the metaphor of 3.1).

3.2.4.2. Conditional clauses



An enquiry-clause is built like a serial-clause (3.1), except that it may contain no labels (apart from ones nested inside some ENCLOSED-clause within the enquiry) and hence no completers. The range that commences with its declarations extends right to the end of the conditional-clause (Ra1 above) and the two serial-clauses are also ranges (Ra2 and Ra3). Each of them can contain declarations, statements, other ENCLOSED-clauses and all the rest of the paraphernalia, or it can be as simple as a single unit:

(E11) **if** *p* **then** *r* **else** *s* **fi**

where *p* would have to be a **bool** (or a **proc bool**), and *r* and *s* might be labels,

or they might be **procs**. Either way, *r* and *s* would be statements and therefore the whole conditional-clause would be a statement, and would yield no value.

In order to save ink, there are alternative representations that may be used for **if**, **then**, **else** and **fi**:

(p | r | s)

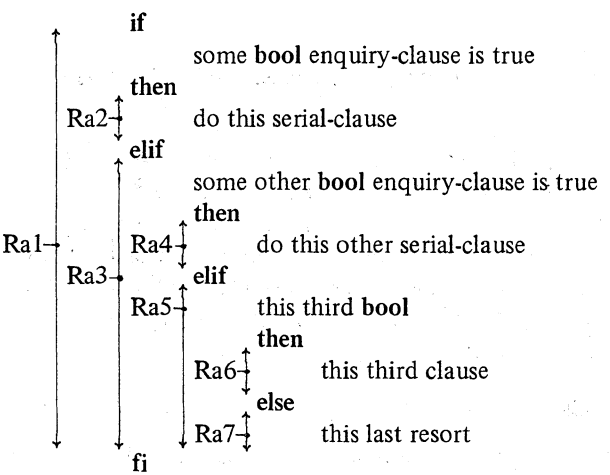
It is quite in order to omit the **else** and its associated clause:

(p | r)

If *p* is **true** the statement *r* is elaborated. Otherwise no statement is elaborated at all. However, this is not a sensible thing to do if the conditional-clause is expected to yield a value, for then the value yielded if *p* were **false** would be undefined.

x := *(i<j | a+b | a−b)*

That was a slightly more ambitious example. *i<j* is a formula yielding a **bool** value (5.1.3 and 6.1.2). The conditional-clause as a whole is required to yield **real** (in order that it may be assigned to *x*). Both *a+b* and *a−b* are formulas yielding **real**, and so all is well. As a matter of fact, it would have been sufficient for them to have been coercible to **real**, and a phenomenon known as "balancing" could have been invoked to aid the process. However, we shall leave discussion of this (and indeed of the coercion of all ENCLOSED-clauses) to 5.2.0.1.



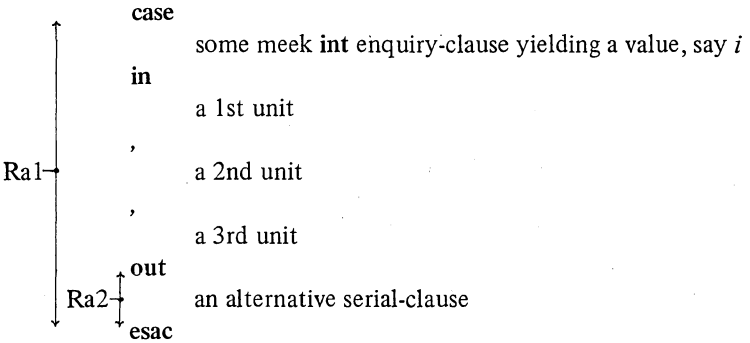
This is a piece of syntactic sugar to save you from writing (or forgetting to write) too many **fi**s. It has an alternative representation:

(E12) *(p | r |: q | s | t)*

meaning:

(E13) **if** *p* **then** *r* **else if** *q* **then** *s* **else** *t* **fi fi**

3.2.4.3. Case clauses



Some number, say *n*, of units are separated by ","s. If the value of *i* is such that

$$i \leqslant 0 \text{ or } i > n$$

then the **out** clause is elaborated. Otherwise the *i*th unit is elaborated. If the case-clause as a whole is required to yield a value, then each unit must be capable of yielding a value of the required mode (but all legitimate coercions and balancings may be applied to this end (see 5.2.0.1)).

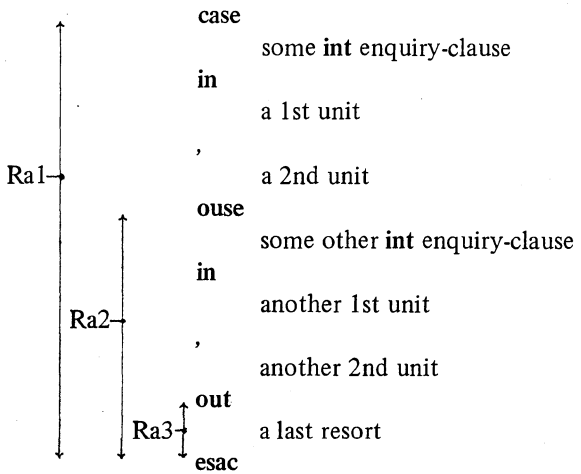
It is quite in order to omit the **out** and its associated serial-clause. If the clause as a whole is a statement, this means that no action is taken. If it is an expression, however, the value yielded is undefined:

(E14)
```
begin int days, month, year;
days := case month in
        31, (year mod 4 = 0 ∧ year mod 100 ≠ 0 ∨ year mod
            400 = 0 | 29 | 28),
        31, 30, 31, 30, 31, 31, 30, 31, 30, 31 esac
end
```

As usual, there are alternative representations for **case**, **in**, **out** and **esac**:

(E15) *print ((i | "SUNDAY", "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "SATURDAY" | "NODAY"))*

Corresponding to **elif** in conditional-clauses, there is **ouse** in case-clauses:



If you think it confusing that "*(*", "*)*". "|" and "|:" should be able to represent so many things, please accept our assurance that no syntactic ambiguity arises. They are quick and easy to write, although it might be kinder to use the longer versions in algorithms intended for publication.

Vertical readers, please turn to 4.2.

3.3. Bold words

Certain bold words have fixed meanings in this language (e.g. **real, begin, if** – see Appendix 1 for the full list). All other bold words may be used for mode-indications and for operators. They are declared to yield modes or routines by means of mode-declarations (2.3) and operation-declarations (4.3.2). These declarations are valid for some range, and so the question of identification arises.

3.3.1. Identification of mode-indications

The identification of identifiers was described in 3.2.3. The purpose and method of identification of mode-indications are exactly the same [R 7.2]. Consider the identification of **r** in:

(E1)
```
begin
  mode r = real;
    begin
      mode r = int;
      r i := j;
    skip
    end;
  r x := y;
  skip
  end
```

An explanation of the identification of operators will be postponed until 4.3.3.

Vertical readers, please turn to 4.3.

3.4. Structure displays

Structure denotations as such do not exist in the language. However, the required effect can be obtained by means of a particular form of ENCLOSED-clause known as a 'structure-display'. These can stand in any strong position where a primary yielding a structure would be allowed (because the position is strong, it follows that the exact mode of the structure-display is always known). Thus, given the declarations involving the mode **vec** in Appendix 2, and the declaration:

(E1) **vec** *v1, v2, v3,*

we can write:

(E2) *v1 := (1, 1, 1)*

but not:

(???) *vl := vl * (1, 1, 1)*

where the context would be firm (see 5.1.0 and 5.1.3).

A structure-display, then, is enclosed between "*(*" and "*)*" (or between **begin** and **end**) and contains one strong unit (5.1) for each field (of which there must be at least two) [R 3.3.1.e, h]. Because each field position is strong, widening is permitted (as indeed happened in the case of the *(1, 1, 1)* in the E2 example above). Because a field position is also a unit, structure-displays are considerably more than a substitute for structure denotations, e.g.:

(E3) *vl := (x + 2, 3.4, i–3)*

Note that the various fields are elaborated collaterally (1.1.2.2).

Vertical readers, please turn to 5.4.

3.5. Row displays and loops

3.5.1. Row displays

Multiple denotations as such do not exist in the language (except for the special case of **string** (5.5.1.1)). However, as in the case of structures (3.4), the required effect can be obtained by means of a particular form of ENCLOSED-clause known as a 'row-display'. These can stand in any strong position where a primary yielding a multiple would be allowed (because the position is strong, it follows that the exact mode of the structure-display is always known). Thus:

(E1) *x1 := (1.2, 2.3, 3.4);*
(E2) *y1 := (x, y, axb+1)*

but not:

(???) *xl := y1 + (1, 2, 3, 4, 5)*

A row-display, then, is enclosed between "*(*" and "*)*" (or between **begin** and **end**) and contains one strong unit (5.1) for each of its elements (of which there must be at least two) [R 3.3.1.d, i].

Because each element is strong, widening is permitted:

(E3) *x1 := (1, 2, 3, 4, 5)*

Note that the various elements are elaborated collaterally (1.1.2.2).

A row-display yields, of course, a multiple value, whose elements are yielded by its units [R 3.3.2.b]. The lower-bound of this multiple is always *1*, and the upper-bound is the number of units in the row-display. Thus

(1, 2, 3, 4, 5)

has bounds [*1*:*5*], and:

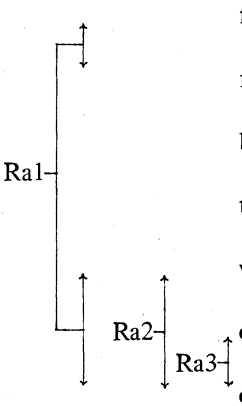
((1, 2, 3), (4, 5, 6))

has bounds [*1*:*2*, *1*:*3*]. Note how the row-display here contains as many row-displays as there are rows, each of which contains as many elements as there are columns.

A special kind of row-display, known as a 'vacuum', has no units at all and yields a multiple of appropriate mode with bounds [*1*:*0*], [*1*:*0*, *1*:*0*], etc.:

(E4) *a1 := ( )*

3.5.2. Loop clauses



for
some **int** identifier, which is hereby declared (I)
from
some meek **int** unit (J)
by
some other meek **int** unit (K)
to
a third meek **int** unit (L)
while
a meek **bool** enquiry-clause (P)
do
a **void** serial-clause (S)
od

For example:

(E5) **for** *i* **from** *k–2* **by** *1* **to** *m* **while real** *lim* = *max real/a; x* < *lim*
do *x* := *x* + *a*↑*i*/*i* **od**

This sums a certain series from *k–2* to *m*, or until *x* is getting too large, whichever happens first.

Now a loop-clause consists of various parts labelled as I through S above. S is the serial-clause which is to be repeated. In E5 it was just one unit, but it will usually contain a substantial amount of program within itself:

(E6) **for** *i* **from** *j* **by** *k* **to** *l* **while** *p*
do
x := *x* + *x1* [*i*];
p := *x* < *max real/2*
od

From inside S you may access I and you may do things which will alter the yield of P, but you may not alter I because its mode is **int** and not **ref int** (this means that the compiler is at liberty to treat I specially, perhaps keeping it in some fast access register). You can do what you like to J, K and L, but it will make no difference to the number of times the loop is obeyed, which is determined once and for all (effects of P apart) at the beginning. The loop is obeyed until I>L (or < if K is negative). I.e. the number of times obeyed is

– **entier** – ((L–J)/K + *1*)

or zero if this is negative. The loop will be obeyed zero times if P yields **false** upon entry.

More precisely, the interpretation of a loop-clause is illustrated by the following piece of program, which is entirely equivalent to E6:

```
(E7)    begin
        int from := j, int by = k, to = l;
           ¢ j, k and l are elaborated (collaterally) and their values are
           remembered. The counting is going to be done in from ¢
    m: if   by > 0 ∧ from ≤ to ∨ by < 0 ∧ from ≥ to ∨ by = 0
           ¢ i.e. if the count is not yet exhausted ¢
       then int i = from; ¢ the user's i is declared here, and is a copy
                            of the current value of from ¢
            if   p     ¢ the user's p, however complex it may be, is
                         elaborated here, each time the loop is about
                         to be obeyed ¢
            then ¢ now comes the user's serial-clause ¢
                    x := x + x1 [i]; ¢ the user may access his i ¢
                    p := x < max real/2; ¢ the user may change his p ¢
                 from := from + by; ¢ the count is in(de)cremented ¢
                 go to m
            fi
       fi ¢ we are now outside the reach of i, so the question of its
           value upon exit does not arise ¢
       end
```

You may omit those parts of a loop-clause that you do not need:

if **for** I	is omitted, there is no I to be accessed	
if **from** J	is omitted, **from** *1*	is assumed
if **by** K	is omitted, **by** *1*	is assumed
if **to** L	is omitted, **to** ∞	is assumed
if **while** P	is omitted, **while true**	is assumed

In fact, the only part which has to be there at all times is the **do** S **od**. If this does stand on its own, then the loop is executed indefinitely, unless you jump out of it.

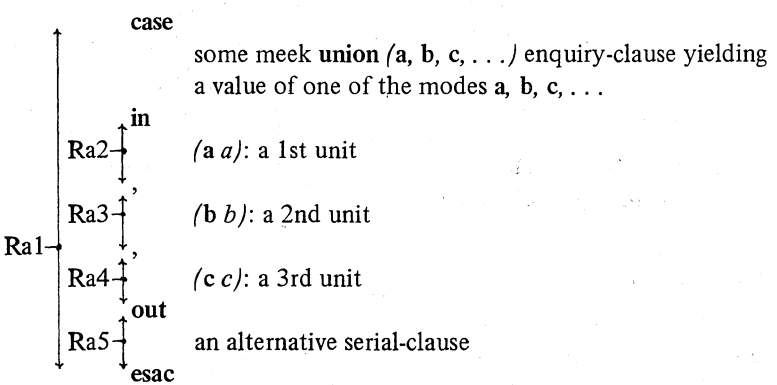
(E8)

ref int *h* = *i*; ¢ so that *h* and *i* are interchangeable identifiers ¢

```
i :=–4;  for i  from 3  by 2  to 3  while i<0   do h:= h+1 od; ¢ obeyed 0 times ¢
i :=–4;         from 3  by 2  to 3  while i<0   do h:= h+1 od; ¢ obeyed 1 time  ¢
i :=–4;                 by 2  to 3  while i<0   do h:= h+1 od; ¢ obeyed 2 times ¢
i :=–4;                       to 3  while i<0   do h:= h+1 od; ¢ obeyed 3 times ¢
i :=–4;                             while i<0   do h:= h+1 od; ¢ obeyed 4 times ¢
i :=–4;                                         do h:= h+1 od; ¢ ad lib ¢
```

Vertical readers, please turn to 4.5.

3.6. Conformity clauses



As with case-clauses (3.2.4.3), there is a number of units separated by ","s, but now each one of them is preceded by a "specification", such as *(***a** *a)*:. The enquiry-clause yields a value whose mode is of one of the modes in its **union**. If the formal-declarer in one of the specifications matches, or "conforms to", this particular mode, then that specification is chosen and the value is ascribed to its identifier (which is a defining-identifier), if it has one (these identifiers being optional). The unit following the chosen specification is then elaborated, and of course applied occurrences of that same identifier can make the value available inside it. On the other hand, if none of the specifications conforms to the mode of the value, then the **out** clause is elaborated.

Here is an example of a conformity-clause yielding a **bool**:

(E1)
```
union (char, bool, int, real) cbira;
if    case  cbira
      in    (bool b): b,
            (int i): i > 0,
            (real r): r > 0
      out   false
      esac
then  ¢ we get here if cbira was not a char and was otherwise
      true or > 0, as the case may be ¢
fi
```

As in ordinary case-clauses, you may omit the **out** clause, you may introduce an **ouse** followed by a new **union** to be tested and a new set of specifications and units, and you may substitute the usual alternatives for **case** etc.:

(E2)
```
union (char, bool) cba, union (int, real) ira;
if    (cba | (char): false, (bool b): b
      |: ira | (int i): i > 0, (real r): r > 0)
then skip
fi
```

(Here we must observe that every value referred to by a **ref union** must include within itself an indication to show of which of its permitted modes it currently is. Therefore a **union** (**int**, **real**), whilst being a very convenient example with which to illustrate the point, is likely to occupy two words of storage – one for the **int** or the **real**, and one to say which it is. Thus there is no practical benefit in using this mode if the intention is to save storage space.
A real saving would occur in the following case:

```
union ([ ] int, [ ] real) irl;
```

because the elements in these multiples are all **int** or all **real**, and only one additional word is needed to indicate the mode of the whole lot.)

Vertical readers, please turn to 5.6.

3.7. Collaterality

3.7.1. Collateral clauses

Collateral-clauses [R 3.3] include such things as structure-displays (3.4) and row-displays (3.5.1), but the ones we are particularly interested in at the moment are void-collateral-clauses. These consist of a list of two or more **void** units separated by commas, and enclosed between **begin** and **end**, or between "*(*" and "*)*":

(E1) *(x:=1, y:=2, z:=3)*

These three statements are elaborated "collaterally". There is not likely to be much gain in using collateral-clauses in this way unless your hardware contains three central processors (so that they can do a statement each), or unless you have reason to believe that your compiler is sufficiently clever to discover that they can be done more efficiently in an order other than that in which they were written down. Alternatively, it might be the case that one of the statements was likely to get held up awaiting some event in real time (transput perhaps), in which case the others would be carrying on. This situation is more likely to arise when parallel-clauses are used (see next section). In the meantime we must consider exactly what "collateral" means.

Collateral elaboration occurs, inter alia, in the following situations:

collateral-declarations
collateral- and parallel-clauses
structure- and row-displays
between the two sides of an assignation or an identity-relation
between the two operands of a dyadic-operator
amongst the primary and the actual-parameters of a procedure call.

Suppose two phrases A and B (it could be more) are to be elaborated collaterally. Then the elaboration of A may be merged in time with that of B in a manner left quite undefined by the Report [R 2.1.4.2.e]. So long as the elaboration of A has no side effect upon that of B, and vice versa, then the manner of this merging has no effect on the result – otherwise, anything might happen. Normally, the two elaborations would proceed until both were completed (3.1.4), but if one were terminated by a **go to**, then the other would be stopped abruptly at whatever stage (if any) it had reached.

In practical compilers, it is probable that A would be elaborated first and then B, or vice versa, but one is not entitled to make any assumptions based on this. Consider the following:

(E2)
```
begin int i;
proc side = int: (i := 1; i := 2; i);
proc add = (int ii, int jj)int: ii+jj;
print (add(side, side))
end
```

The two calls on *side* are elaborated collaterally. If one were elaborated entirely before the other (in either order), each would yield the value *2*, and 4 would be printed. In the corresponding ALGOL 60 program, this would be the guaranteed result. However, this ALGOL 68 program is perfectly entitled to print 3 as its answer, because *i* is global to *side* and the collateral elaboration of the two calls is quite entitled to be merged [R 2.1.4.2.e] in the following manner:

```
i := 1;              ¢ on behalf of the first side ¢
i := 2;              ¢ likewise ¢
        i := 1;      ¢ this is the second side starting up ¢
i                    ¢ on behalf of the first side, which therefore
                       yields the value 1, as set by the second side ¢
        i := 2;      ¢ the second side ¢
        i            ¢ the second side yields 2 ¢
```

Here is another example:

(E3)
```
i := 0;
x1[(i +:= 1; j – 2)] := x2[(i +:= 1; j – 2)];
print (i)
```

The two subscripts are to be elaborated collaterally. Suppose their elaborations were merged as follows:

```
Take i (=0)
Add 1 (=1)
                        Take i (=0)
                        Add 1 (=1)
Store in i (1)                      (because the operator +:= in-
                        Store in i (1)  cludes assignation)
Take j
                        Take j
subtract 2
                        subtract 2
use to index x1
                        use to index x2
```

Thus 1 is printed, even though *i +:=1* has been elaborated twice. In general it may be said that if two identical clauses are to be elaborated collaterally, then the implementation is quite entitled (but not of course bound) to perform the elaboration of only one of them and to assume that the other yields the same result. In other words, a compiler may detect common sub-expressions (such as the *(i +:= 1; j – 2)* in E3), and optimise its code accordingly, and if a user has put side effects into these sub-expressions, he has no right to complain if they do not behave as he expected.

So the moral of this story, if you were thinking of use side effects and collaterality is involved, is "Don't".

3.7.2. Parallel clauses

These are like collateral-clauses, except that you are provided with some control over the synchronisation of the constituent statements. They consist [R 3.3.1.c] of **par** followed by a void-collateral-clause:

(E4) **par***(x:=1, y:=2, z:=3)*

However, this example, although legitimate, does not take advantage of the synchronisation facilities provided. To take a more realistic example, suppose we have a procedure which generates lines of output at random intervals:

```
(E5) proc item generate = ( c computes the next item of output,
                                taking a random length of time to do so c);
     mode item = struct ( c a collection of values intended to be printed c );
```

We wish to print these **item**s of output on a lineprinter which operates in real time (i.e. not disguised by an operating system), and at some fixed number of lines per minute. In order to smooth out the irregular periods between the generation of **item**s (so as, hopefully, to keep both the printer and the central processor busy at all times), we shall put the **item**s into a buffer as they are generated, and take them out at a rate suited to the printer. First let us declare our buffer:

```
(E6) int nmb buffers = c the number of items the buffer can hold c;
     [1 : nmb buffers] item buffer;
     int index := 0, exdex := 0; ¢ pointers to items within the buffer ¢
     bool work to be done := true, printing to be done := true;
                ¢ we shall need these in order to know when to stop ¢
```

Now we must set up some semaphores so that the generating department and the printing department can communicate with each other. There is a special mode provided for this purpose:

(E7) **sema** *free slots, full slots;*

A **sema** has a reference to an **int** hidden inside it, but you are only allowed to get at it by means of the special monadic operators **level, up** and **down** [R 10.2.4]:

operator	priority	mode of *a*	mode of result	meaning
level	10	**int**	**sema**	yields a **sema** referring to a copy of the **int** *a*
level	10	**sema**	**int**	yields the **int** referred to by the **sema**
down	10	**sema**	**void**	if the **int** referred to is zero, then the elaboration of this part of the parallel clause is "halted", otherwise the **int** is reduced by *1*
up	10	**sema**	**void**	the **int** referred to is increased by *1* and all elaborations previously halted by the operation of **down** on this particular **sema** are "resumed" by repeating the tests for zero in their **down**s

We shall now initialise our semaphores:

(E8) *free slots* := **level** *nmb buffers;* ¢ *because we have not generated any* **items** *yet* ¢
full slots := **level** *0;* ¢ *because there are no* **items** *waiting for printing* ¢

Now we come to our parallel-clause:

(E9) **par begin**
¢ *the generating department* ¢
while *work to be done*
do

```
            down free slots; ¢ halts this department if all the slots
                                are full. Initially, there are plenty ¢
            index modab nmb buffers +:= 1; ¢ increment index
                                              modulo nmb buffers ¢
            buffer [index] := generate;
            if c there are no more items to generate c
               then work to be done := false fi;
            up full slots ¢ to enable the other department to get
                           going ¢
            od
          , ¢ comma to separate the two statements. ¢
            ¢ the printing department ¢
            while printing to be done
            do
            down full slots; ¢ halts this department if there is
                              nothing to print (as initially) ¢
            exdex modab nmb buffers +:= 1;
            print ( buffer [exdex] );
            printing to be done := work to be done ∨ index≠exdex;
            up free slots ¢ if the other department was halted,
                           it may now be resumed ¢
            od
    end
```

For a more ambitious example of parallel-clauses see R 11.12. For a general discussion of the use of these semaphores see:

E.W. Dijkstra, Cooperating Sequential Processes, contained in "Programming Languages", Ed. F. Genuys, Academic Press.
and E.W. Dijkstra, Comm ACM **11** 5 May 1968 p 341.

Vertical readers, please turn to 5.7.

4. ROUTINES

4.1. Procedures and operators

In ALGOL 68, procedures arise out of the structure of the language in a very natural way. Thus routines are values, which therefore have modes. They become ascribed to identifiers or operators by the elaboration of declarations, and they are called in the course of a variety of units.

Therefore, there is hardly a topic in this area which could not have been fitted elsewhere in our orthogonal plan (which is, indeed, why the Report itself contains no chapter on the subject). However, the chapter which you are about to read is not entirely redundant, since we thought it proper in view of their central importance to gather all the information about routines, procedures and operators into one place.

The necessary concepts were introduced in 1.1.4, which indicated how to declare a procedure:

(E1) **proc** *reciprocal* = (**real** *a*) **real**: *1/a;*

and how to call it:

(E2) **real** *x;*
x := *reciprocal (3.14)*

Also how to declare an operator:

(E3) **op oneover** = (**real** *a*) **real**: *1/a;*

and how to use it in a formula:

(E4) **real** *x;*
x := **oneover** *3.14*

All these matters will be discussed at greater length in 4.2 in the case of procedures, and 4.3 in the case of operators. In particular, note how the right hand side of E1 is the same as that of E3. This is the part which defines precisely what the routine, which is being created in either case, is to do, and is known as a 'routine-text' (4.2.2.1).

4.1.1. Standard prelude routines

However, a large proportion of the operators and procedures which you will call in the course of your programs will not have been declared by you in

this way, because they will be already built in to your program by means of the 'standard-prelude' (1.1). Amongst these you will find procedures for all the usual mathematical functions (sine, cosine, square root, etc. – the full list is given in 6.2.2), and operators for all the usual mathematical operations (+, –, ×, ÷, etc., and a lot of not-so-obvious ones – all listed in 6.1., 6.3, 6.5 and 6.7).

Although the procedures declared in the standard-prelude will be just that – when you call them, certain built-in routines will be entered – this may not be so for the operators. For example, the operator "+", used to add two **ints** together, is defined in the Report [R 10.2.3.3.i] by the following operation-declaration:

(E5) **op** + = *(***int** *a, b)* **int**: *a* – – *b;*

If you had used this in your own program, it would have compiled a routine to do *(a* – – *b)*, and called it in whenever such a "+" was encountered in a formula. This is not the intention for the standard-prelude operators. When this "+" appears in a formula, your compiler should produce, on the spot, the most efficient possible code to do the same job.

Therefore you need hardly be aware, when using these operators, of the strange way in which they were introduced into the language, and any such lack of awareness should be no bar to a full understanding of 5.1.3 where the use of operators in formulas is fully discussed.

Vertical readers, please turn to 5.1.

4.2. Procedure declarations

4.2.1. **proc** declarers

The concept of a "routine" was introduced in 1.1.4. Each routine is of some mode, and for each such mode we can write declarers. There are four classes of routine, categorised according to whether they require parameters or not, and whether they return a value or **void**.

(E1) **proc void** ¢ no parameters, returns **void** ¢
(E2) **proc int** ¢ no parameters, returns **int** ¢
(E3) **proc (real, int, ref char) void** ¢ 3 parameters, returns **void** ¢
(E4) **proc (real, int, ref char) bool** ¢ 3 parameters, returns **bool** ¢

This last one would be pronounced in public as:

> "procedure-with-(a-)real-parameter-(and-an-)integral-parameter-(and-a-)reference-to-character-parameter-yielding-(a-)boolean"

To say that a routine returns "**void**" is to say that it returns no useful value.

Formal **proc** declarers and actual **proc** declarers look exactly the same [R 4.6.1.o]. (Note also that the declarers specifying the modes of the parameters and of the value returned are always formal.)

The parameters and returned value of a routine can be of any mode whatsoever, including another **proc** mode:

(E5) **proc (real, proc (real, int) void) proc (int) real**

A 'procedure' is an external object which upon elaboration yields a routine:

(E6) **proc void** *proca;*

is a variable-declaration declaring a **ref proc** *proca*. The value referred to by *proca* is at present undefined, since we have not yet provided a routine for it. However, if it had referred to some routine, then:

(E7) **proc void** *procb* := *proca;*

would have created *procb*, making it refer to the same routine (initially). Which all goes to show that routines may be assigned and otherwise handled just like values of any other mode.

4.2.2. Routines

What, then, is a routine? It is a piece of code somewhere within your program, compiled there as a result of some statements written by you (or maybe it was put there by the standard-prelude (6.2.2)). When a routine is assigned, as in E7 above, there is of course no question of moving pieces of code around inside the computer – it is a pointer to the piece of code that is handled during these operations, but the effect is just the same. Note that there are no operations operating on routines defined within the standard-prelude (with a little ingenuity, you might construct some of your own, but they would probably not be particularly useful).

There are, therefore, two questions that we have to answer:

1) How do we introduce routines into a program, and cause them to be yielded by procedures?
2) How do we "call" them – i.e. cause them to be obeyed?

Major discussion of the second question will be deferred until 5.2.1.

4.2.2.1. Routines texts

A 'routine-text' [R 5.4.1] is used to create a routine. It may stand, inter alia, as the actual-parameter of an identity-declaration, or of a call (see E18 below), or as the RHS of an assignation. Here is a routine-text yielding a routine of mode **proc** *(***real, real** *)* **real**:

(E8) *(* **real** *a,* **real** *b)* **real**: *a+b*

Please note that *(***real** *a,* **real** *b)* **real** is not a declarer such as **proc** *(***real, real***)***real** is (declarers do not contain identifiers). The **real** *a* and the **real** *b* occurring in E8 are formal-parameters, such as you would expect to find on the LHS of an identity-declaration (2.2.1), and the two **reals** in these formal-parameters and also the **real** specifying the mode to be returned are therefore formal-declarers. The *a+b* in E8 is a strong unit (5.1.0.2) and forms the body of the routine.

Now we may use E8 in an identity-declaration, to declare a **proc** *(* **real, real** *)* **real**:

(E9) **proc** *(* **real, real** *)* **real** *sum* = *(* **real** *a,* **real** *b)* **real**: *a+b;*

Rather a cumbersome way of adding two **reals** together, and rather a cumbersome way of declaring it, too. There are two contractions we can use to

shorten it. Firstly, the **reals** in the two formal-parameters of the routine-text may be gathered together in the familiar (2.1.2) fashion (i.e. *(***real** *a, b)*). Then, there still being considerable redundancy, all of the formal-declarer after the **proc** may be omitted [R 4.4.1.b], leaving:

(E10) **proc** *sum* = *(***real** *a, b)* **real**: *a+b;*

which really is about as short as you could expect. In this form it is known as a 'routine-identity-declaration'. Likewise:

proc *refsum* ¢ of mode **ref proc** etc ¢ := *(***real** *a, b)* **real**: *a+b;*

which was a 'routine-variable-declaration'.

When there are no formal-parameters, the routine-text becomes very simple, as in this **proc real**:

(E11) **real**: *x + 3.14*

which may appear in an identity-declaration with the usual routine contraction:

(E12) **proc** *xplus* = **real**: *x + 3.14;*

Finally, there is one more contraction to be used where several procedures are to be declared (but please use it only for short snappy ones):

(E13) **proc** *iplus* = **int**: *i + 3*, *zplus* = **compl**: *z + 1* **i** *1;*

The part after the ":" in a routine-text can be any strong unit (Chapter 5) yielding the required mode. Most often it will be some form of ENCLOSED-clause (3.2.4), as in the following example in which we also illustrate a routine-text returning **void**:

(E14) **proc** *pqrs* = *(***ref real** *a, b)* **void**: *(i<0 | a := 3.14 | b := 3.14);*

Note that in a routine-declaration the RHS must always be a routine-text. If we go back to the uncontracted identity- or variable-declaration, this restriction does not apply and we can, for example, make the routine to be ascribed or assigned dependent upon some condition:

(E15) **proc void** *pq* = *(i<0* | **void**: *a := 3.14* | **void**: *b := 3.14);*

Here we have two routines available. Which of the two is ascribed to *pq* will depend on the value of *i* at the time E15 is encountered. Please compare this example carefully with E14, in which there is only one routine which tests *i* each time it is called.

A routine-text is a quaternary (5.1.0.1), and this determines where it may

be used (thus if it was needed as the operand of a formula (5.1.3) it would need to be enclosed between "*(*" and "*)*".

4.2.2.2. Calling

A routine with formal-parameters can be called by providing it with actual-parameters to match its formal ones. Within the context of E10 we could put:

(E16) *a := sum (x, y)*

x and *y* are the actual-parameters of this call. What happens next is just as if the routine-text had been transformed into a cast containing an identity-declaration to match each formal-parameter with its actual counterpart. The elaboration of the call is equivalent to the elaboration of that cast. Applying this process to E16 and E10, we get:

real *(*
real *a = x, b = y;*

after which comes the body of the routine:

a+b
)

Observe how the *(* and the *)* demarcate a new range, so that the formal-parameters *a* and *b*, to which **real** values have been ascribed for the duration of this call, may not be confused with any occurrences of *a* and *b* elsewhere. Observe also that **real** *a = x, b = y* is a contracted collateral-declaration, so that *x* and *y* are elaborated collaterally (3.7.1).

In this example, **real** values were ascribed to the formal-parameters *a* and *b*, and so it would have been illegal to try to assign to them from within the routine. In ALGOL 60, this would have been known as "call by value". If we do wish to alter the value referred to by a formal-parameter, then that parameter must be of a mode that refers to something, as in E14 which permits the following call:

(E17) *pqrs (x, y)*

which will assign *3.14* to either *x* or *y*, according to the value of *i* at the time. We term this "call by reference". In ALGOL 60, you would have had to use (or misuse) "call by name" for that one.

To get some other effects of the ALGOL 60 call by name, however, you must declare your procedure with **proc** mode parameters:

(E18)
```
proc series = (int k, proc (int) real term) real:
        begin real sum := 0;
        for j to k do sum +:= term (j) od;
        sum
        end;
```

This sums the terms of some series from *1* to *k*. When it is called, the actual-parameter provided for the term can be any unit that yields a **proc (int) real**, and very commonly this will be a routine-text.

(E19)
```
x := series (100, (int i) real: 1/i)
```

During a call of this routine, the procedure ascribed to *term* (in this case the routine-text) is called once each time round the loop-clause. This is how, in ALGOL 68, we achieve the effect of Jensen's device.

See 5.2.1 also for other examples and further discussion of calls.

A routine without formal-parameters is called by means of a coercion known as deproceduring. This is described fully in 5.2.1, but here is a brief example:

(E20)
```
begin
real x;
proc pp = void: x := 3.14;
        begin
        real x := 0;
        pp;             ¢ pp is called ¢
        print ( x )     ¢ prints 0.0 ¢
        end;
print ( x ) ¢ prints 3.14 ¢
end
```

When *pp* is called, the routine from **void**: *x := 3.14* is entered. Note, however, that the name *x* assigned to by this routine is (as we hope you would expect) the one declared in the outer range, and not the one declared just before the call.

4.2.2.3. Recursion

Suppose, now, that a routine happens to contain a call on itself (either directly, or via a chain of calls on other routines which eventually calls the same one again). Are there any problems? In some programming languages there may be, but not in this one. It all works out normally, just like you

would expect. You will find several examples of such recursion in this book, notably in 8.7.1. Here is another one:

(E21)
```
begin
proc blocked = (int x, y) bool: c A description of a maze, centred
                                  at (0, 0) with entrance at (0,100).
                                  Yields true if the point (x, y) is
                                  inaccessible (part of the walls). The
                                  maze is presumed to contain no
                                  cycles. c;
int x := 0, y := 100, d := 0; ¢ starting coordinates and direction ¢
proc maze = bool :
    if    blocked (x, y) then false
    elif  x = 0 ∧ y = 0 then true
    else  int presx := x, presy := y, presd := d, i := 0;
    loop: i := i + 1;
          x := presx + ((d := (presd + 2 + i) mod 4) + 1 | 0, −1, 0, +1);
          y := presy + (d + 1 | −1, 0, +1, 0);
          if maze then true else (i < 3 | go to loop); false fi
    fi;
print (if maze then "Maze is solved" else "No route to centre "fi )
end
```

Clearly, *maze* can call itself recursively to a considerable depth. Now *maze* contains declarations for the variables *presx, presy, presd* and *i*, which must be elaborated whenever it is called (two trivial cases apart). This means that four locations must be reserved on the stack (1.2.2.3). Next, *maze* calls itself recursively, and we meet these declarations again. Do we get the same four locations? Of course not! We reserve another four on the stack, and the first four become inaccessible until such time as we return to the particular call of *maze* in which they were created. Then we will find that their values have not been touched since we left them.

So, by the time *maze* has called itself to a depth of 20, there are 20 instances of these four variables on the stack, and their values form a complete record of how we got from the entrance to where we are. So, if we replace the last line but one of *maze* by:

```
if maze then print ((presx, presy, newline)); true else (i < 3 | ...
```

then we shall get printed a complete set of directions showing how to get out again.

4.2.3. Scopes of routines

The following example should be compared carefully with 3.2.2. E3:

```
(E22)  begin proc void pp; real y;
           begin real x; proc p = void: y := x;
           x := 2.0;
           p;        ¢ this is all right. 2.0 is assigned to y ¢
           pp := p   ¢ this one is going to cause trouble ¢
           end;
       print (y); ¢ no complaints ¢
       pp;        ¢ tries to assign x to y, but who is x? ¢
       print (y)  ¢ now what? ¢
       end
```

In an assignation, the scope of the RHS must be at least as old as that of the LHS, and in the case of *pp* := *p* above, it was obviously the scope of the routine ascribed to *p* (i.e. **void**: *y* := *x*) that was too new.

In fact, the scope of a routine corresponds to the smallest range containing a declaration of an identifier or a bold word (2.3 or 4.3.2) which is used inside that routine [R 7.2.2.c] (i.e. the inner range in the above example because the routine contained an *x*). There are two small exceptions. A mode-indication used inside the routine as a formal-declarer (i.e. not as an actual one) does not count for this purpose, neither does an applied-operator whose only crime is to identify a priority-declaration (4.3.1) outside.

In both this example, and in 3.2.2. E3, the trouble could have been caught by a check at compile time, but in the following example a run time check would be necessary:

```
(E23)  begin ref real xx, proc copy = (ref real a) ref real: a;
           begin
           real x := 2.0;
           xx := copy (x)
           end;
       print (xx)
       end
```

Vertical readers, please turn to 5.2.

4.3. Operation declarations

The operators used in formulas (5.1.3) are either symbols built in to the language for the purpose, or bold words (1.3) invented by the user. (Note that a bold word used in a mode-declaration (2.3) may not, in that reach (3.2.1), be used for an operator.) (Note also that the built in symbols may all be used for either monadic- or dyadic-operators, except for the symbols <, >, /, =, ×, and * which may not be monadic [R 9.4.2.1].)

In order to be used, an operator must yield a routine, and if it is to be used as a dyadic-operator it must have a priority too. Now several routines may, at one and the same time (subject to a restriction that will be discussed below in 4.3.3), be ascribed to a given symbol (or bold word), but that symbol (or bold word) may only have one priority. Therefore, before a symbol can be used as a dyadic-operator, it must be given a priority (unless it has already acquired one in the standard-prelude [R 10.2.3.0]).

4.3.1. Priority declarations

There are 9 available priority levels for symbols to be used as dyadic-operators (for convenience, we classify monadic-operators as having priority 10 in Chapter 6, but this is purely our own convention). We associate a priority with a symbol (for the duration of some range) thus [R 4.3.1]:

(E1) **prio min** = *9;*

Priority-declarations may be incorporated into collateral-declarations:

(E2) **prio min** = *9*, **prio max** = *9;*

and this may be contracted into:

(E3) **prio min** = *9*, **max** = *9;*

4.3.2. Operation declarations

An operation-declaration looks rather like an identity-declaration:

(E4) **op** *(***real, real***)* **real min** = *(***real** *a, b)* **real**: *(a < b | a | b);*

The RHS of this one is a routine-text (4.2.2.1), but in general it is an actual-parameter (2.2.1) – so if you were trying to be very posh you might try to organise yourself some other unit which (after suitable coercion, of course) would yield a **proc** *(***real, real***)* **real**. Normally, however, a routine-text is as far as you will ever need to go, in which case you may then immediately contract

it as with routine-declarations (cf 4.2.2.1. E10):

(E5) **op min** = *(***real** *a, b)* **real**: *(a <b | a | b);*

The operator **min** now works for pairs of **reals**. However, we may wish it to work for other combinations:

(E6) **op min** = *(***int** *a, b)* **int**: *(a < b | a | b),*
(E7) **min** = *(***int** *a,* **real** *b)* **real**: *(a < b | a | b),*
(E8) **min** = *(***real** *a,* **int** *b)* **real**: *a* **min real** *(b);*

E8 was trying to be clever by using the version of **min** already declared in E5. It is an interesting example of the use of a cast (5.1.1.3) but not an efficient way of doing a job, as E7 was. Note the contraction whereby the **ops** have been gathered together (cf 4.2.2.1. E13).

Now **min** yields four routines, but this is only the start of it. Now there are all the combinations of **long reals** and **long ints** (2.7.2), and no doubt sensible meanings could be found for **min** when operating upon **chars**, **strings** and all sorts of things.

min is a dyadic-operator (so far), and as such yields routines which have two formal-parameters. Monadic-operators, of course, yield routines with one formal-parameter:

(E9) **op min** = *(*[ ] **real** *a1)* **real**:
begin real *x* := *max real* ¢ 6.2.1 ¢;
for *i* **from lwb** *a1* **to upb** *a1* **do** *(a1* [*i*] <*x* | *x* := *a1* [*i*] *)* **od**;
x **end**;

Routines yielded by operators are entered when those operators are encountered in the elaboration of formulas [R 5.4.2]. For a full understanding of formulas, you should consult 5.1.3. It will suffice here to show how the operands of the formula are matched up to the formal-parameters of the yielded routine (having selected the right routine to match the modes of the operands, of course). This process is identical to that used when procedures with parameters are called (4.2.2.2). So, if we have the formula:

(E10) *i* **min** *x*

we first select the E7 version of **min**, and then construct the following collateral-declaration:

int *a* = *i,* **real** *b* = *x;*

In the reach of this declaration, *(a < b | a | b)* is elaborated, and the **real** result becomes the value yielded by the formula.

4.3.3. Identification of operators

The identification of identifiers was described in 3.2.3, and the purpose and method of identification of operators are essentially similar. Consider the identification of **min** in:

(E11)
```
begin
    prio min = 9;
    op min = (real a, b) real: (a < b | a | b),
       min = (int a, b) int: (a < b | a | b);
    a := x min y;
        begin
            prio min = 8;
            op min = (ref real a, b) ref real: (a < b | a | b);
            xx := x min y
        end;
    k := i min j
end
```

Firstly, you must identify all the applied occurrences of **min** in the formulas with the defining occurrences of **min** in the priority-declarations. This results in the dotted lines to the left hand side of E11.

Secondly, you must identify all the applied occurrences of **min** in the formulas with the defining occurrences of **min** in the operation-declarations. But you must only accept, in your search, a defining occurrence the modes of whose formal-parameters can be firmly coerced from the modes of the operands of the formula [R 7.2.1]. This results in the lines on the right of E11.

However, let us now try to declare another version of **min**:

(E12) **op min** = (**ref real** *a, b*) **ref real**: (*a* < *b* | *a* | *b*);

The purpose of this one is to determine which of two names (of mode **ref real**) refers to the smaller value. Let us use it in a formula:

(E13) *x* **min** *y*

But Oh dear! Doesn't this also identify the version of **min** declared in E5 (in a formula, the operands are firm, and so *x* and *y* can be dereferenced in this example (5.1.0))?

Clearly, it must be forbidden for E5 and E12 to occur in the same reach. This is expressed by saying that the modes of the formal-parameters of two declarations of the same operator in the same reach must not be "firmly

related" [R 7.1.1], i.e. one of them, or a component mode of one of them (if it is a **union**), must not be firmly coercible to the other. Since **ref real** (in E12) is firmly coercible to **real** (in E5), these certainly do not pass the test.

However, even if two declarations are in different reaches, as they are in E11, and they are firmly related, then the one in the inner reach renders invisible the one in the outer reach. Thus you could not have written, immediately after *xx* := *x* **min** *y* in E11, *a* := *x* **min** *2.0*. It certainly could not have identified the **min** declared for two **ref real** parameters, and it is not allowed to see the other one. On the other hand, *k* := *i* **min** *2* would have been all right in this position. This additional restriction simplifies implementation of the language and is unlikely to affect the average user, since it is more likely that such an identification would indicate an error on his part, than that it would be his real intent.

Vertical readers, please turn to 6.3.

4.5. Row-of parameters

Consider routines of modes such as **proc** *(*[ ] **real***)* **void**. When such a routine is called, the compiler may be obliged to take a copy of the value of the actual-parameter – in general a time consuming operation – just in case the body of the routine should contrive in some way to alter the original:

(E1)
```
[0 : 99] real x1;
proc a2 = (ref [ ] real b1) void: for i to 99 do x1 [i] := b1 [i–1] od;
a2 (x1)
```

The intention and effect of this example is not to make every element of *x1* a copy of *x1*[*0*], which is what would have happened if *b1* had not yielded a copy of the value of *x1* at the start of the call.

However, cases such as this are rare, and a decent implementation will postpone making the copy until the problem actually arises, if at all. But not all implementations are decent, and so it may be wiser to declare your formal-parameter as a **ref** [ ] **real**. Then, as call, it is only a name which has to be passed to the routine, which uses it to access the original multiple value that it refers to. However, if a routine is provided with a name, it is entitled to be told whether the multiple value referred to is fixed or **flex**ible:

(E2)
```
proc a5 = (ref [ ] real b1) void: XXXXX;
proc a6 = (ref flex [ ] real b1) void: XXXXX;
```

For further discussion of this point, see 8.5.

Vertical readers, please turn to 5.5.

4.7. Jumps

We have informally been using jumps in this book right from the start, but mainly only in places where we had not yet introduced choice-clauses (3.2.4) or loop-clauses (3.5.2). Since these already provide powerful and adequate facilities for controlling the flow of your program, jumps are hardly necessary in ALGOL 68. Their main legitimate uses are for premature exits from loop-clauses and from event routines (7.4.4).

4.7.1. Simple jumps

We have already seen (3.1.2) that labels may appear anywhere in a serial-clause provided no declarations come afterwards:

(E1) *labl*:

A jump to a label consists of **goto** (or **go to**) followed by the label that is to be identified (3.2.3), or simply just of that label by itself:

(E2) **goto** *labl*
go to *labl*
labl

The effect of the jump is to cause the unit following that label to be elaborated next. A jump may only occur in a strong context (its mode is irrelevant, but is automatically regarded as the mode expected for syntactical purposes).

4.7.2. Procedured jumps

In some other languages, there is a mode **label**, and one may assign labels to **label** variables and subsequently jump to them. In ALGOL 68, however, **proc**s are used for this purpose [R 5.4.4.2]:

(E3) **proc void** *ppp;*
ppp := **goto** *stop*

The elaboration of this constructs the routine

void: goto *stop*

and assigns it to *ppp*, just as if we had written

(E4) *ppp* := **void: goto** *stop*

An interesting application of this facility can be used to realise the equivalent of the ALGOL 60 **switch** facility:

(E5) [ ] **proc void** *switch* = *(e1, e2, e3);* ¢ a multiple of **procs**.
e1, e2 and *e3* are labels ¢
switch [*i*] ¢ jumps to the label selected by *i* ¢

In fact, if a jump appears in any strong context where a procedure without parameters is expected (no coercions allowed), this is what happens. In all other contexts, the jump is simply obeyed immediately. Thus we may distinguish between

(E6) *x* := **if** *p* **then** *y* **else goto** *stop* **fi**

and

(E7) *task1* := **if** *p* **then void**: *print ("P WAS TRUE")* **else goto** *stop* **fi**
¢ see Appendix 2 for *task1*.
if *p* is **false**, we assign **void**: **goto** *stop* to *task1*,
but we do not go there yet ¢

Vertical readers, please turn to 5.7.

5. UNITS

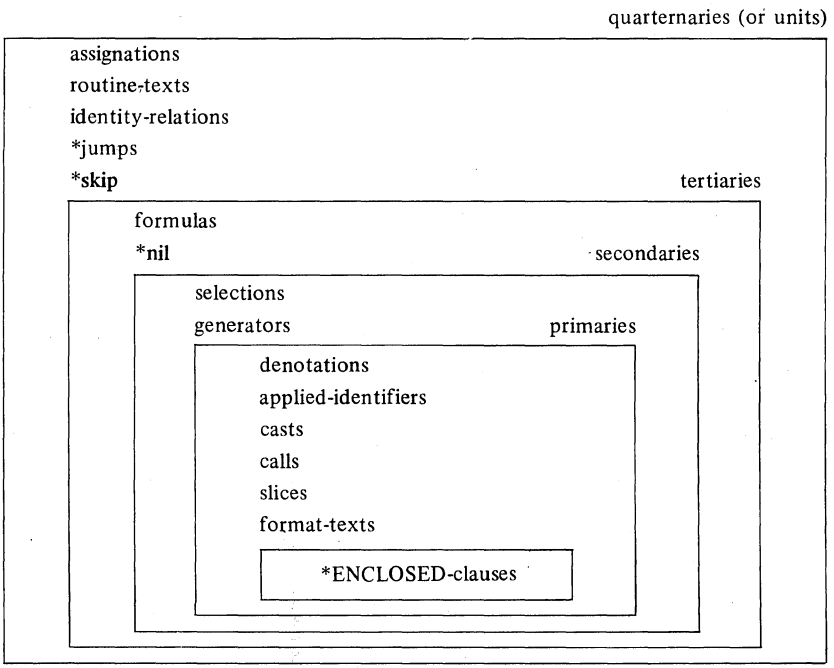
5.1. Simple units

'Units' (also termed 'unitary-clauses') are the entities in the language which actually get things done. The simplest example of a unit is the simple type of assignation (e.g. *x* := *y* + *2.4*) which we have used many times already. However, examples much more complex than this can be constructed in accordance with a set of rules which it is the purpose of this chapter to describe. (The corresponding definitions are mostly to be found in R5.)

5.1.0. Coercion

5.1.0.1. Coercends

The basic building blocks out of which units are made are known as 'coercends', of which there are 16 types, arranged in a hierarchy as follows:



* Strictly speaking, these units are not coercends, since they cannot be coerced.

A unit can be any quaternary. Right at the bottom are ENCLOSED-clauses (as *(a+b)* in *x := y × (a+b)*) which are themselves made up of further units (as described in 3.2.4), so that the definition of the whole setup is recursive.

Now the elaboration of a unit (i.e. of a coercend or of an ENCLOSED-clause) has two effects.

Firstly, it must yield a value (e.g. the value of the formula *2+3* is *5*). This value will be of some mode, possibly **void**, uniquely determinable at compile time. If the mode is **void**, then the unit is a 'statement'; otherwise it is an 'expression'. Secondly, some other actions (independent of what is yielded) may take place (e.g. in *x := 2.4*, the value *2.4* is assigned to *x*).

5.1.0.2. Coercion

Now the a priori mode of a coercend may have to be coerced (see 1.1.6) into the mode that is required by the "context" in which that coercend appears [R 6.1]. Thus, the a priori mode of *2* is **int**. If *2* occurs in the context *x := 2*, then the expected mode (after the *x :=*) is **real**. When this assignation is elaborated, then, the integral value *2* must be coerced into the real value *2.0* before the assignation can proceed. Fortunately this particular coercion (which is known as "widening") is permitted in this particular context and this assignation is therefore legitimate. The question as to whether any one of the 6 permissible coercions may be applied in a particular case depends upon the context. For this purpose, contexts are divided into 5 classes:

strong
firm
meek
weak
and soft

All 6 coercions may be applied in strong positions. During the rest of this chapter, as we describe each form of unit, we shall indicate the strength of the contexts occurring in it. Here, in the meantime, is a summary:

strong contexts	The RHS of identity-declarations (2.2.1)
	The actual-parameters of calls (5.2.1)
	The RHS of initialized variable-declarations (2.2.3)
	The RHS of assignations (5.1.4.1)
	The ENCLOSED-clauses of casts (5.1.1.3)
	The units of routine-texts (4.2.2.1)
	Statements (must yield **void**) (5.7.0.1)
	All constituents but one of a balanced clause (5.2.0.1)
	One side of an identity-relation (5.7.4) ←----------

firm contexts	Operands of formulas (5.1.3) In effect, the actual-parameters of transput calls (7.1.1, 7.1.2)
meek contexts	Trimscripts (must yield **int**) (5.5.1.3) Enquiries (3.2.4.2, 3.2.4.3, 3.5.2, 3.6) Primaries of calls (e.g. *sin* in *sin(x)*) (5.2.1)
weak contexts	Primaries of slices (e.g. x in $x[i]$) (5.5.1.3) Secondaries of selections (e.g. z in *re* **of** z) (5.4.2)
soft contexts	The LHS of assignations (5.1.4.1)
---------->	The other side of an identity-relation (5.7.4)

A complete chart of all the coercions is given below. The way to use this chart is as follows. Consider first the mode (a priori) of the available coercend and secondly the mode (a posteriori) required by the context. Then you must find a route following the arrows on the chart that will, through a sequence of intermediate modes, take you from the first to the second. If the coercion is possible, then there will be such a route (if there are several routes, they will always be found to be equivalent).

Suppose, for example, that in a strong context you have (a priori) a coercend of mode **proc ref bool** and what you really need (a posteriori) is a value of mode [ ] **union** *(***real, int, bool***)*. Then there is indeed a route between them, but the simplest way of describing it to you will be to introduce a fictitious operator to represent each coercion on the way, as suggested in 1.1.6 (of course these operators are not really part of ALGOL 68). Thus the required value is obtained by the following operations on the coercend:

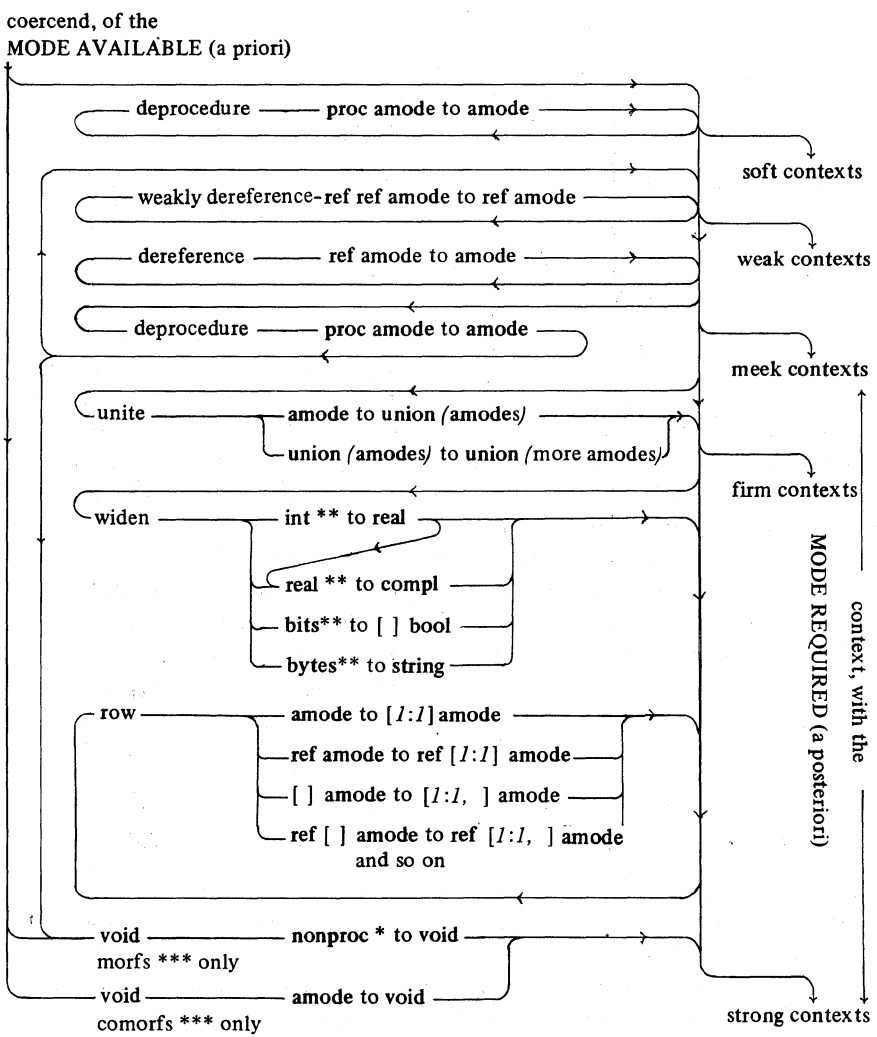
ROW*(***UNITE***(***DEREFERENCE***(***DEPROCEDURE***(***coercend*)))*)*

and you will encounter these operations (from the innermost to the outermost) as you follow the route. Doubtless you will be relieved to hear that coercions do not invariably get so complex.

The change of mode brought about by each of these fictitious operators is given in the chart. Their effect upon the elaboration of the program will be found at appropriate points in this introduction, as follows:

voiding	5.7.0.1
rowing	5.5.0
widening	5.1.0.4, 5.4.0, 5.7.0.2
uniting	5.6.0
dereferencing	5.1.0.3, 5.4.2, 5.5.1.3
deproceduring	4.2.2.2, 5.2.0.2, 5.2.1

COERCION CHART



* **nonproc** is all modes except **proc amode** and **refs proc amode.**

** The corresponding **longs** and **shorts** versions can also be widened.

*** Comorfs are assignations and casts. The rest are morfs. See 5.2.1.

If you cannot find a suitable route through the maze simply because your context is not strong enough, then all is not lost. A device known as a 'cast' has been provided wherein you first state the mode you would like to have (a posteriori), and then strongly coerce yourself into it regardless. Casts are described in 5.1.1.3 below. They are particularly useful for dereferencing in soft contexts and for widening in firm ones.

5.1.0.3. Dereferencing

We have already explained in 1.1.2.1 and in 2.1.2 how a declaration such as **real** *x;* causes a location in the memory to be reserved (or "generated") for a **real** value, the name which refers to that value being ascribed to the identifier *x*. Thus, the value yielded by *x* is a name of a priori mode **ref real**.

Now, very frequently, what we want is the **real** value stored in the location referred to (as *x* in the assignation *y* := *x*) and what we have got is its name. We must therefore have resort to the coercion known as "dereferencing".

Dereferencing [R 6.2] is permitted in any context that is strong, firm, meek or weak (which is almost everywhere). (A slight restriction in the case of weak positions will be discussed in 5.4.2.) The effect is to remove one **ref** from the a priori mode, and to yield the value of the thing that was named. If this value is in turn another name, then further dereferencing may be required.

Thus if the **ref real** identifier *x* stands in a context that is strong, firm or meek, and if the expected mode is **real**, then the value yielded is the **real** value that *x* refers to.

5.1.0.4. Widening

Consider:

(E1) *x* := *i*

Widening [R 6.5] is used to turn an **int** value into a **real** value (also a **real** value into a **compl** value (see 5.4.0) and some further cases in 5.7.0.2). In this example, therefore, the **ref int** *i* is first dereferenced to yield an **int** value, which is then widened to yield the corresponding **real** value, which can then be assigned to the name *x*.

We shall now consider the simpler forms of coercend, starting with the primaries:

5.1.1. Primaries

We shall consider three forms of unit here—denotations, identifiers and casts.

5.1.1.1. Denotations

Denotations are those entities which, in earlier languages, would have been known as "literals" or "constants". The following examples show some legitimate denotations, together with the a priori modes of the values that they yield [R 8.1].

Yielding **int**	*2* ; *1024* ; *123* ; *0123*
Yielding **real**	*12.3* ; $1.23_{10}1$; $.123_{10}+1$; *0.123e+1* ; *1230e–1* *0.00123* ; *.00123* ; *123.0* ; but not *123.*
Yielding **bool**	**true** ; **false**
Yielding **char**	*"a"* ; *"B"* ; *"1"* ; *","* ; *"."* for a space symbol; *""""* for a (single) quote symbol

From a study of these examples, you should be able to construct any other denotation that you might require.

For **string** denotations see 5.5.1.1, for the **void** denotation see 5.6.1 and for **bits** and **long** denotations see 5.7.1.

5.1.1.2. Applied identifiers

An identifier standing as a unit constitutes an applied occurrence (1.1.5) of that identifier. Somewhere, that same identifier will have been declared (at its defining occurrence). These two occurrences must be correlated since the a priori mode of the value yielded when it stands as a unit is the same as the mode with which it was declared. The exact method of correlation is considered in 3.2.3.

5.1.1.3. Casts

Suppose the operator ◊ had been defined for pairs of **reals**, but not for **ints**, and suppose you wanted the formula:

(E2) $x \Diamond i$

which would not then be allowed. *i* cannot be widened because its context is only firm. If only it were strong. Let us therefore make a mould the shape of a **real**, and melt up our **int**, and cast it into the required shape:

(E3) $x \Diamond$ **real** *(i)*

This is all right. **real***(i)* is a 'cast' [R 5.5.1]. The formal-declarer **real** specifies that it shall yield a **real**, which suits the ◊ operator. Immediately after the

formal-declarer the context is strong, and an ENCLOSED-clause is expected. Thus the **int** *i* may be widened to **real** and everyone is happy.

Other examples of the use of casts will be given in 5.2.4. Ingenious users will find many other applications. For example, in transput (see 7.1), given *i = 1234*:

(E4) *print(i);* ¢ will print + 1234 ¢
*print(***real***(i))* ¢ will print + $1.234_{10}+3$ ¢

5.1.2. Secondaries

Discussion of secondaries will be postponed until Section 5.4.2.

5.1.3. Tertiaries–formulas

(The reason why we sometimes prefer to talk of formulas rather than formulae is to be found in the Report at 1.1.4.2.a, but we would not recommend that you should read that just yet.)

The following are examples of 'formulas':

(E5) $x - 2$; $x \diamond y$ ¢ if a meaning for ◊ has been declared ¢ ;
$x \times a + b$; $x \times (a + b)$; -2

It will be seen that the essential feature of formulas is that they contain operators which operate upon operands [R 5.4.2].

If a formula contains more than one operator, then there is an implied bracketing which ensures that each dyadic-operator has two clearly defined operands, and each monadic has one. In order to assist with the implied bracketing, each dyadic-operator has an associated priority in the range of *1* through *9*, all monadic-operators effectively having priority 10. For example, "↑" has priority *8*, "x" and "/" have *7*, and "+" and "–" have *6*. The rule is that the operators with the highest priority are always considered first. Thus:

(E6) $x \times a + b$ means $(x \times a) + b$
(E7) $-a + b$ means $(-a) + b$ (because the "–" here is monadic)
(E8) $+4 - 2 \uparrow 2$ means $(+4) - (2 \uparrow 2)$ (and yields *0*)
(E9) $-2 \uparrow 2 + 4$ means $((-2) \uparrow 2) + 4$ (and yields *8*)
(E10) $4 + -2 \uparrow 2$ means $4 + ((-2) \uparrow 2)$ (and yields *8*).

We agree that E8 and E9 are confusing, but it was thought that to have some dyadic-operators of priority higher than the monadics would have been

even more so. The operator "↑", as in $a \uparrow b$, should not be thought of as equivalent to the usual mathematical notation for "to the power" as in a^b, which is itself a notation for a function such as *pow (a, b)*.

Where several operators of the same priority occur together, an additional rule is needed. Thus for dyadic-operators:

(E11) $i - j + k - m + n$

means

(E12) $(((i - j) + k) - m) + n$

Likewise for monadic-operators:

(E13) **+ abs entier** $- x$

means

(E14) **+(abs (entier** $(-x)))$

The priority and meaning of each operator are either built into the standard-prelude (6) or library-prelude (1.1) or are defined by the user (1.3.3.2 and 4.3).

The mode(s) of the operand(s) in a formula must match the mode(s) for which its operator has been defined. For example, the dyadic-operator "+" is defined (6.1.2) to do a variety of things, amongst which is a definition which states that if its first operand is **real** and its second operand is **real**, then it yields a **real** value which is to be the sum of its two operands (within the accuracy permitted by the implementation). A separate definition states that if its first operand is **int** and its second is **int**, then it yields an **int** value, and there are ten other similar definitions, not to mention three more for its monadic counterpart (6.1.1).

An operand can be any tertiary except **nil**, provided it is of the required mode. E.g. it can be another formula (as in the implied bracketing examples above), a selection, a denotation or an ENCLOSED-clause, but it cannot be an assignation because an assignation is not a tertiary (see 5.1.0.1). Thus if you wanted to operate upon an assignation, you would have to make it into a closed-clause thus:

(E15) $x + (b := a \times y)$

The tertiary which constitutes an operand is in fact firm, and the permitted coercions therefore include dereferencing, but not widening. Thus in:

(E16) $x + y$

the names *x* and *y* are first dereferenced to yield two **real** values, which are then added to yield the **real** value of the formula.

Consider also:

(E17) $x := i + j$

i and *j* cannot be widened because of the firm context, but the version of "+" to add two **int**s can be used. Then the value of the whole formula *i+j*, being of mode **int**, can be widened to **real**.

(E18) $x := i + y$

Here again, *i* cannot be widened in order to be added to *y*. The formula *i+y* is only valid by virtue of the fact that the operator "+" is also defined (6.1.2) for the case of an **int** plus a **real** yielding a **real**.

5.1.4. Quaternaries

5.1.4.1. Assignations

An 'assignation' [R 5.2.1] is one of the commonest forms of unit. It consists of two parts–a left hand side (its 'destination') and a right hand side (its 'source'). Consider the following example:

(E19) $x := y + 3.14$

The LHS (*x* in this example) is subject to the following restrictions:

a) It must yield a name (i.e. its mode must be **ref amode**; in the example above *x* was **ref real**).
b) It must be a tertiary (in the example *x* was an applied-identifier; a formula is also possible but in fact few operators yield names (but see 6.3)).
c) Its context is soft, which means in particular that no dereferencing is allowed (unless you use a cast).

Application of these rules completely determines the mode of the value referred to by the name yielded.

The RHS (*y + 3.14* in the example) therefore has considerable freedom, the rules being the following:

a) It must yield a value whose mode is the same as that of the value referred to by the left hand side (in the example a **real** value is yielded, which is in agreement with the **ref real** mode of the left hand side).
b) It can be any quaternary, which means it can be any coercend or any ENCLOSED-clause, provided a suitable mode is yielded (in the example it was a formula).

c) Its context is strong, which means that any known coercion can be applied in order to obtain the required mode (in particular, both widening and dereferencing can be used).
d) It has a scope restriction, but this is described in 3.2.2.

Consider:

(E20) $x := y$

Both x and y are, a priori, names (of mode **ref real**). y must be dereferenced to yield a **real** value. The value formerly referred to by x is then superseded by this **real** value.

$x := a + b$

In this case the RHS is the formula $a + b$, which already yields a **real** value. No dereferencing is therefore needed (note, however, that a and b were in fact dereferenced during the elaboration of the formula itself).

Since any quaternary can stand for the RHS of an assignation, it follows in particular that another assignation can so stand. It is therefore necessary to specify what value (and of what mode) is to be yielded. In fact, the value yielded by an assignation is the value yielded by its LHS, which is always of mode **ref amode**. Consider the following:

(E21) $a := b := x := 2.4$

Let us first insert the implied bracketing (which, as you will observe, is not that of a dyadic-operator (see 5.1.3. E12)):

(E22) $a := (b := (x := 2.4))$

First of all, the **real** value *2.4* is assigned to x. The value of $x := 2.4$ as a whole is the name x which, being of mode **ref real**, must be dereferenced before the value to which it refers (which is now *2.4*, of course) can be assigned to b. The value of this assignation is the name b, the value referred to by which (*2.4* again) can now be assigned to a. Thus, everybody lands up by referring to his private instance of *2.4*. The formula $x + (b := axy)$ given in example E15 causes the product axy to be assigned to b. x and the new value now referred to by b are then added together.

5.1.4.2. skip

skip is a special form of unit. As a statement, it is a dummy. In other strong contexts, it yields an undefined value of the mode demanded.

Note that it is never subject to coercion and that it may only occur where the context is strong.

Vertical readers, please turn to 6.1.

5.2. Balance and call

In this section we consider balancing, procedure calls, and also some further examples of coercends involving names.

5.2.0. Coercion

5.2.0.1. ENCLOSED clauses and balancing

Any form of ENCLOSED-clause (3.2.4) may stand as a primary. Often, the effect is straightforward:

(E1) *y* := *x* × *(a+b)*

However, let us consider again the example E9 from 3.2.4.1:

(E2)
```
begin
print (4 × ( real w := 0, int i := 1; real z = sqrt (small real/2);
             loop: w := w + 2/(i × (i + 2)); i := i + 4;
             if 1/i > z then loop fi;
             w ) )
end
```

In this example, the value of the serial-clause within the parentheses is, a priori, the name *w* (of mode **ref real**) which will have vanished by the time we are outside the clause. Fortunately it is also clear that, if the value of this serial-clause is a name, it ought to have been dereferenced (because the operator "×" is expecting a **real**). However, the rules provide that an ENCLOSED-clause cannot be dereferenced (it is not a coercend). Instead, the required mode and the strength of the context are passed on to the unit which is to be yielded, which in this case is the identifier *w*. Therefore, it is *w* that gets dereferenced, right at the last moment before it disappears, and the resultant **real** value is passed on. The following piece of syntax (which is not the complete syntax for a serial-clause) may make this clearer to those who have some familiarity with the Report [R 3.2.1].

SORT: strong; firm; weak; meek; soft.
SORT MOID serial clause:
strong void unit, go on token, SORT MOID serial clause;
declaration, go on token, SORT MOID serial clause;
SORT MOID unit.

The ENCLOSED-clause might well be a conditional-clause:

(E3) $x := (i<0|-i|i)$

or it might be a case-clause:

(E4) $x := (i|j,k,m,n)$

In these cases as well, any coercion that might appear to be needed on the ENCLOSED-clause as a whole is instead performed on the unit(s) inside, as the context may permit. Indeed, different coercions may be applied to different internal units;

(E5) x := **case** i **in** j, k, x, y **esac**

The first two alternatives in E5 would have to be widened – the last two are already **real**. Widening is possible because the case-clause occurs in a strong context. However, even if the context had been firm, widening would have been permitted provided that at least one of the alternatives had been **real**, e.g.:

(E6) $a := x \times (i<0 \mid j \mid y)$

The fact that y is **real** shows that the version of the operator "x" requiring a **real** is intended, and therefore the context of j can be promoted to strong. The same holds for

(E7) $a := x \times (i<0 \mid x \mid k)$

This principle is known as "balancing". However:

(E8) $a := x \times (i<0 \mid j \mid k)$

involves the multiplication of a **real** with an **int**, and is not balanced.

Balancing is permitted between:

a) The completion points of a serial-clause [R3.2.1] (i.e. the **exits** and the final unit (3.1.4)), e.g.

(E9)
```
a := b + ( real x, y, z; z := 1 − 3 × sqrt(small real); x := i;
           if x ⩽ 0 then go to libl fi;
     lobl: y := i/x↑2; x := (2 × x + y)/3;
           if y/x < z then go to lobl fi;
           if x ⩾ 10.0 then go to libl fi;
           x exit
     libl: print ("out of range");
           10)
```

Please compare that carefully with 3.1.4. E7.

b) The alternatives of a conditional-clause [R 3.4.1], following **then** or **else** (3.2.4.2 and examples E6 and E7 above).

c) The alternatives of a case-clause, including the **out** option (3.2.4.3), e.g.:

(E10)
```
a := y × (i |j, k, x, y)
```

d) The alternatives of a conformity-clause, including the **out** option (3.6).

e) The LHS and RHS of an identity-relation (5.7.4). Experienced readers might like to consider the rather delicate example 5.7.4. E28.

See also 5.5.1.3 for the balancing of transient names.

5.2.0.2. Deproceduring

Deproceduring is a method of calling routines not having parameters, and has already been introduced in 4.2.2.2. We think it best, however, to consider it alongside calls of routines which do have parameters, which brings us to:

5.2.1. Primaries – procedure calls

It will have been seen (1.2.3.1 and 4.2) that procedures are declared in much the same way as other objects, and that they yield values (i.e. their routines). Thus **proc void**, **proc real**, **ref proc** *(* **int**, **real***)* **int** are perfectly valid modes. The consequence of which is that if *random* (which is declared to be of mode **proc real** (see 6.2.2)) appears as a coercend, it is not immediately apparent whether its value (i.e. its routine, which is of mode **proc real**) is to be yielded, or whether the intention is to elaborate the routine and to yield its **real** result (although the latter may well be intended 99% of the time, the former facility is needed whenever a procedure is to be assigned, or operated upon in a formula, or yielded by another procedure – all of these things being quite allowable).

The distinction between these two interpretations can only be made by

context. There are two cases:
a) Calls [R 5.4.3]

(E11) **proc** *(***real***)* **real** *p;* **proc** *sinh* = *(***real** *x)* **real:** *(exp(x) – exp(–x))/2;*
y := *sinh(x);*
(E12) *p* := *sinh*

sinh has been declared to be a procedure requiring a parameter. Therefore, in E11 where an actual-parameter is indeed present, the intention is clearly that the procedure should be called. If there are no parameters, as in E12, then its body must be yielded. Thus the problem does not arise.

b) Deproceduring [R 6.3]

proc real *q;* **proc** *sinh x* = **real:** *(exp(x) – exp(–x))/2;*
(E13) *y* := *sinh x;*
(E14) *q* := *sinh x*

sinh x does not require parameters. Therefore, we must see which mode is required by the context. In E13, **real** is required, which is what the routine yielded by *sinh x* returns. Therefore we must employ the coercion known as "deproceduring".

The effect of deproceduring is always to convert the mode of a coercend of **proc amode** into **amode** (including **proc void** into **void**), at the same time calling the value (i.e. the routine) that the coercend yields.

In E14, on the other hand, the mode required is **proc real** and so the routine yielded by *sinh x* is assigned without any coercion.

Deproceduring can be used in any context*, including the LHS of an assignation.

In elaborating a call of either sort, it is first necessary to establish what is to be called. In case (a) this is specified by a meek primary yielding the required routine, and the primary is followed by the actual-parameters. The interpretation of these has already been described in 4.2.2.2. Usually, the primary will be an identifier, as *ncos* (see Appendix 2) in:

(E15) *x* := *ncos (i)*

* In strong **void** contexts, there may be some doubt as to whether deproceduring or voiding (5.7.0.1) is to be used. The coercion chart (5.1.0.2) shows that deproceduring is always to be preferred to voiding, except where the coercend is an assignation or a cast. In E14 it was an assignation yielding (a priori) the **ref proc real** value *q*. Clearly, to have now called this would have been ridiculous, and therefore immediate voiding was appropriate. If *q* had stood as an applied-identifier on its own, however, it would have been dereferenced, then deprocedured (so that any side effects of *sinh x* could happen), and finally voided.

However, it could be an ENCLOSED- (e.g. conditional-(3.2.4.2)) clause, as in:

(E16) *x* := *(p | ncos | nsin) (i)*

In case (b), where deproceduring is to be used, the required routine must be yielded by a suitable unit (but not an assignation or a cast). Again, usually, it will be an identifier as in:

x := *random*

It cannot, however, be an ENCLOSED-clause (which is not a coercend). If *proca* and *procb* are both of mode **proc real**, then:

(E17) *x* := *(p | proca | procb)*

is legitimate, but the deproceduring of *proca* and *procb* takes place in situ, as you have already seen in 5.2.0.1, and the conditional-clause as a whole yields **real** without further coercion.

The corresponding situation in the case of calls with actual-parameters arises in:

(E18) *x* := *(p | ncos(i) | nsin(i))*

which should be compared with example E16 in which the actual-parameter (*i*) appeared only once.

The modes yielded by the actual-parameters in a call must, of course, match those of the formal-parameters (4.2.2.2) of the routine. However, the context of an actual-parameter is strong, so that all the coercions are available.

5.2.3. Tertiaries – **nil**

nil is a special tertiary of mode **ref amode** which yields a name which does not refer to any value. See 5.2.4.E27 for an example. Note that **nil** is never subject to coercion and that it may only occur where the context is strong.

5.2.4. Quaternaries – assignations involving names

Here are some examples of assignations involving names. Remember that *xx* and *yy* are of mode **ref ref real** and that *a*, *x* and *y* are merely **ref real**:

(E19) *xx* := **if** *i*<*0* **then** *x* **else** *y* **fi**;

The value of *xx* is therefore the name *x* or the name *y*.

(E20) *yy* := *xx;*

and so is the value of *yy*.

(E21) *a* := *xx;*

The value currently referred to by *x* or *y* (whichever was assigned to *xx*) is assigned to *a*. Note that *xx* is dereferenced twice in this example.

(E22) **ref real** *(xx)* := *a;*

The value referred to by *a* is assigned to *x* or to *y*. *xx* is here put in a cast, so that it may be dereferenced. There is normally little point in using a cast as the RHS of an assignation, since:

(E23) *y* := **real** *(x);*

means the same as:

(E24) *y* := *x;*

(E25) *a* := *xx* := *x;*

means the same as *a* := **real** *(xx* := *x)*, in which the name *x* is assigned to *xx*, the value referred to by the value referred to by which (i.e. it is dereferenced twice) being then assigned to *a*. On the other hand:

(E26) *xx* := *a* := *x;*

means the same as *xx* := *(a* := *x)*, in which the value referred to by *x* is first assigned to *a*, and the name *a* is then assigned to *xx* (but the resultant value of *xx* is no way dependent upon *x*, so that one might just as well have written *a* := *x;xx* := *a;*).

(E27) *xx* := **nil**

means that the value referred to by *xx* is a name which does not refer to anything.

Vertical readers, please turn to 6.2.

5.4. Units and structures

5.4.0. Coercion – complex widening

The widening of **ints** into **reals** was introduced in 5.1.0.4. In a similar fashion, it is possible to widen a **real** into a **compl**, provided the context is strong. Thus:

(E1) $z := x$

x is first dereferenced into a **real** and then widened.

(E2) $z := 2$

Here the **int** *2* is first widened into a **real**, and then widened again into **compl**.

Even an implementation (of a sublanguage) which does not include the **compl** operators in its standard-prelude ought to implement this particular widening (i.e. of a **struct** *(***real** *re, im)*), in order that a user may then declare these operators himself, and use them in the normal fashion.

5.4.1. Primaries – applied identifiers

Clearly, once a **struct** (or a **ref struct**, etc.) has been ascribed to an identifier, then that identifier can stand as a unit, and the value yielded is the whole of some structure (however complicated) of the appropriate mode (or the name of such a structure). For example, in:

(E3) *v1* := *v2*

v2, which is of mode **ref struct** *(***real** *xcoord, ycoord, zcoord)* (see Appendix 2) is dereferenced to yield a value (consisting of three **real** quantities) which is of mode **struct** *(***real** *xcoord, ycoord, zcoord).* Such identifiers can occur in assignations (as above), or in formulas, as in:

(E4) *v1* := *v2* + *v3*

or in selections, as will now be discussed.

5.4.2. Secondaries – selections

A 'selection' is a form of secondary which can be used to obtain an individual field out of a structure, thus:

(E5) *xcoord* **of** *v1; re* **of** *z*

Here, *xcoord* and *re* are field-selectors (see 2.4.1) and *v1* and *z* are identifiers.

More specifically, any weak secondary can be selected from, and this has various consequences as follows:

a) Because it is weak, the secondary can be dereferenced, but a special provision attaches to weak dereferencing. If we have a name referring to a structure, we have the right to expect, from our selection, a name referring to the selected field, and so the dereferencing of the secondary must yield the name of the structure, never the structure itself. Therefore, in weak dereferencing, succeeding **ref**s may be removed from the mode of the secondary until one remains, but this last one cannot go.

A special rule now provides that where the secondary thus yields the name of a structure, the selection as a whole yields the subname (1.1.4.2) referring to the selected field (but if the secondary (being of a non-**ref** mode) yields the structure itself, then the selection yields the field itself). For example in:

(E6) *xcoord* **of** *v1*

v1 is of mode **ref struct** *(***real** *xcoord, ycoord, zcoord)* and hence the example as a whole yields a name of mode **ref real**. This selection as a whole may now be dereferenced if its context is strong, firm or meek, as happens in:

(E7) $x := y +$ *xcoord* **of** *v1*

However, if we declare:

(E8) **vec** *w1 = (1, 2, 3);*

in which *w1* is of mode **vec**, then:

(E9) *xcoord* **of** *w1*

is of mode **real**. Thus both *v1* and *xcoord* **of** *v1* are acceptable on the LHS of an assignation, but *w1* and *xcoord* **of** *w1* are not.

For another viewpoint over this whole matter, you are invited to re-read 1.4.3.

b) Because it is a secondary from which the selection is made, and because a selection is a secondary, it follows that selections can be selected from. If we declare:

(E10) **mode tens = struct** *(* **vec** *xlevel, ylevel, zlevel);* **tens** *u1;*

then we may call upon:

(E11) *xcoord* **of** *ylevel* **of** *u1*

and the result is of mode **ref real**. Moreover, if we declare:

(E12) **mode man = struct** *(* **int** *age,* **ref man** *father);* **man** *jones;*

then we may call upon:

(E13) *age* **of** *father* **of** *father* **of** *father* **of** *father* **of** *father* **of** *jones*

and the result is of mode **ref int**. Each time we select a *father* in this example, we obtain a **ref ref man** which, since it occurs in a weak position, can be dereferenced as far as **ref man** which is just what we need in order to be able to select another **ref ref man** from it.

5.4.3. Tertiaries – formulas with complex operators

It will be recalled (2.4.4) that **compl** is really a structure of mode **struct** *(***real** *re, im)*. Six special operators are provided for use with **compl**s. They are **re, im, abs, arg, conj** and **i**. The first four operate on **compl** and yield **real**. For example:

(E14) **re** z means the same as **real** *(re* **of** *z)*

(E15) **re** *(w + z)* means the same as *re* **of** *(w + z)*

(E16) **abs** *z* means the same as *sqrt (***re** *z*↑*2* + **im** *z*↑*2)*

Contrariwise:

(E17) **re** *w* := *x*

is not permitted, although:

(E18) *re* **of** *w* := *x*

is.

(E19) **conj** *z* means the same as **re** *z* **i** – **im** *z*

i is a dyadic-operator which produces a **compl** out of two **reals** or **ints**. Thus:

(E20) *z* := *x* **i** *y* means the same as *z* := *(x, y)*

However:

z := *w* + *(x, y)*

is not legitimate because the context of *(x, y)* is not strong.

(E21) *z* := *w* + *x* **i** *y*

is all right because **i** is of higher priority than "+".
N.B. Those still unhappy with the interpretation of $-x\uparrow 2$ should pay some attention to **re** $z\uparrow 2$, in E16 above.

Vertical readers, please turn to 7.4.

5.5. Units and multiples

5.5.0. Coercion – rowing

The restriction that a row-display (3.5.1) should contain either zero or at least two units is necessary in order to avoid ambiguity. However, a multiple value is perfectly entitled to contain only one element, and in order to be able to assign values to such multiples a coercion known as "rowing" is provided. This may be used in any strong context. Thus, given:

(E1) **flex** [*1 : 2*] **real** *w1;*

then

(E2) *w1* := *2.4;*

causes a multiple (with bounds [*1 : 1*]) of one element (i.e. *2.4*) to be assigned to *w1*. Also:

(E3) *w1* := *(1, 2, 3);*
flex [*1 : 2, 1 : 4*] **real** *w2* := *w1;*

gives *w2* the bounds [*1 : 1, 1 : 3*].
Likewise:

(E4) **flex**[*1 : 2*] [*1 : 3*] **real** *w3* := *w1;*
gives *w3* the bounds [*1 : 1*] [*1 : 3*].

Thus, rowing consists of adding one 'row' or one 'row of' to the mode of a value, at the same time providing bounds [*1 : 1*] (several 'row's or 'row of's may be added by repeated rowing). See also 5.5.1.3. E20 for how to produce names by rowing

5.5.1. Primaries

5.5.1.1. String denotations

A **string** is of course of mode [ ] **char**, and therefore a literal string could

be provided by means of a row-display:

(E5) *s* := *("T", "H", "E", "␣", "Q", "U", "I", "C", "K")*

However, a more compact denotation is provided:

(E6) *s* := *"THE␣QUICK␣BROWN␣FOX␣JUMPS␣OVER␣THE␣LAZY␣DOG"*

The value of the RHS here is a multiple value of mode **string**, and with bounds [*1:43*]. Note the denotation "␣" for the space character. You can, if you like, use a space as the space-character:

(E6*) *s* := *"BUT IF YOU HAVE A VERY LONG STRING-*
DENOTATION WHICH CONTINUES ON TO THE NEXT
LINE, HOW CAN YOU TELL HOW MANY SPACES
THERE WERE AT THE LINE BREAK?"

Presumably as many as you actually punched when you punched the program in the first place. Note that this is an exception to the general rule (1.3.2) that blank spaces have no meaning in this language.

If the quote-sumbol itself is required to appear in the string, it must be represented by two quote-symbols (i.e. ""), thus:

(E7) *s* := *"He said ""she said ""he is a liar"""" """ "*

An empty string can also be assigned:

(E8) *s* := *""*

There is no denotation for a string of one character only. However, in strong positions the same result can be obtained by taking a character-denotation (5.1.1.1) and rowing (5.5.0) it:

(E9) *s* := *"A"*;
s := *""""*

Note that a comment may not appear inside a string- (or character-) denotation, and thus the comment symbol (¢) may safely appear, and stand for itself [R 8.1.4.1].

5.5.1.2. Applied identifiers

Clearly, once a multiple (or a **ref** to a multiple, etc.) has been ascribed to an identifier, then that identifier can stand as a unit, and the value yielded is the whole of some multiple value of the appropriate mode (or the name of such a value). Note that the value so yielded includes a descriptor. For

example:

(E10) *x1* := *y1*

(in which *y1* has to be dereferenced before a multiple value is obtained).

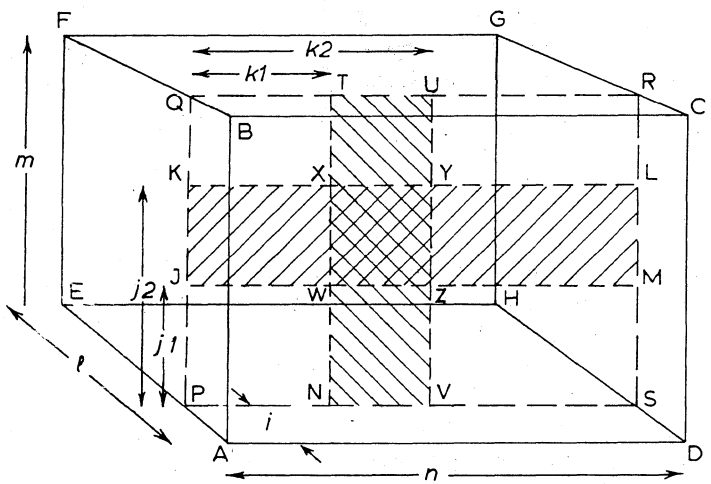
5.5.1.3. Slices

Slices are used in order to dissect multiple values. They consist of a weak primary, which yields a multiple value, followed by an 'indexer' containing a series of 'trimscripts' which specify which parts of that multiple value are required. Trimscripts may be either 'subscripts' or 'trimmers'.

Let us declare:

(E11) [,,] **real** *x3* = (c *some multiple with bounds* [*0:l–1, 0:m–1, 0:n–1*] c);

which is of mode [, ,] **real**. It can be represented thus:



ABCDEFGH represents the whole value, *x3*. From this, we may select the plane PQRS by writing:

(E12) *x3*[*i*, ,]

which yields a value of mode [,] **real**. This could now be assigned to any doubly subscripted variable which it happened to fit. Here, *i* is a subscript. Further subscripts can be used to yield the row JM (of mode [ ] **real**) and the element W (of mode **real**):

(E13) *x3[i,j1,]; x3[i,j1,k1]*

To obtain the column TN (of mode [ ] **real**) we write:

(E14) *x3 [i, , k1]*

Trimmers are used to obtain a part (a "subvalue") of a row, column, etc. The required lower- and upper-bounds (both inclusive) are given, separated by a colon. Additionally, the 'revised-lower-bound' from which the yielded bounds are to run is also specified following an @ (effectively, the subvalue between the specified lower- and upper-bounds is extracted, and its bounds are then "slid down" until the revised-lower-bound is reached). If no revised-lower-bound is specified, *@ 1* is assumed. The following examples should make this clear:

(E15)

slice	value yielded	mode yielded	bounds yielded
x3[i, j1:j2@j1 ,]	JKLM	[,] **real**	*[j1:j2, 0:n−1]*
x3[i, j1:j2 ,]	JKLM	[,] **real**	*[1:j2−j1+1, 0:n−1]*
x3[i, :j1 ,]	PJMS	[,] **real**	*[1:j1+1, 0:n−1]*
x3[i, : ,]	PQRS	[,] **real**	*[1:m, 0:n−1]*
x3[i, j2: @j2 ,]	KQRL	[,] **real**	*[j2:m−1, 0:n−1]*
x3[i, , k1:k2]	NTUV	[,] **real**	*[0:m−1 , 1:k2−k1+1]*
x3[i,j1:j2@j1, k1:k2@k1]	WXYZ	[,] **real**	*[j1:j2, k1:k2]*
x3[i, j1, k1:k2]	WZ	[ ] **real**	*[1:k2−k1+1]*
x3[i,j1 :j2, k1]	WX	[ ] **real**	*[1:j2−j1+1]*
x3[i, j1, k1]	W	**real**	–
x3[, ,]	ABCDEFGH	[,,] **real**	*[0:l−1, 0:m−1, 0:n−1]*
x3[: , : , :]	ABCDEFGH	[,,] **real**	*[1:l, 1:m, 1:n]*
x3	ABCDEFGH	[,,] **real**	*[0:l−1, 0:m−1, 0:n−1]*

(The last line is not strictly a slice at all).

Note that, if a bound is omitted from a trimmer, the bound currently existing in that multiple is implied. Moreover, if both bounds are omitted, both existing bounds are taken (but *@ 1* is still implied). Alternatively, when both bounds are absent, the colon may be omitted as well, but now the existing lower-bound is assumed (i.e. there is no sliding). This accounts for the difference between *x3*[, ,] and *x3*[: , : , :] in E15. The first is the same as *x3* (no trimmers, existing bounds). The second is an abbreviation for *x3[0:l−1, 0:m−1, 0:n−1]* (full set of trimmers, with *@1* implied for each).

Subscripts, and bounds occurring in trimmers, are meek **int** units, and therefore expressions of considerable complexity can be used, including any coercion that is able to yield **int**. The use of an assignation will be quite common:

(E16) *x3*[*j* : = *i*, ,]

A slice consists of a weak primary, followed by an indexer. The weak primary leads to the following consequences:

a) Because it is weak, it can be dereferenced, but only until one **ref** is left (see 5.4.2 for the corresponding phenomenon in connection with selections). Thus dereferencing can never yield the multiple value itself.

A special rule now provides that where the primary thus yields the name of a multiple value, the slice yields the subname which refers to the element or subvalue that has been sliced (but if the primary already yields a multiple value – as *x3* in the examples above – then the slice yields the element or subvalue itself). The similarity between this and the corresponding provisions for selections (5.4.2) should be noted.

However, names referring to multiple values may be either fixed or flexible (2.5.2.1) so that there are two cases to consider. For example, in:

(E17) *x1*[*i*]

x1 is a fixed name of mode **ref** [ ] **real**, and hence the example as a whole yields a name of mode **ref real**, which may itself now be dereferenced if its context is strong, firm or meek, as happens in:

(E18) *x* := *y* + *x1*[*i*]

whereas in:

(E19) *xx* := *x1*[*i*]

we have obtained, in *xx*, a pointer to a **real** value, which just happens to be a particular element of a [ ] **real**.

(The converse operation, in which a pointer intended for a multiple value can instead point to a single value is also possible:

(E20) **ref** [ ] **real** *xx1* := *x;*

This involves rowing (5.5.0), and the bounds, when the **real** value referred to is accessed via *xx1*, are [*1* : *1*]).

A pointer to a subvalue can also be obtained:

(E21) **ref** [ ] **real** *xx1* := *x2*[*2* : *4*, *i*];

in which *xx1* is made to point to part of the *i*th column of *x2*.

If, however, the primary yields a flexible name, as in:

(E22) *a1*[*i*]

where *a1* is of mode **ref flex** [ ] **real**, we have to be careful. We indeed get a

name, and its mode is **ref real** as in E17, but it is said to be a "transient name" [R 2.1.3.6.c] because it is only meaningful so long as the flexible *a1* stays the same size. If the whole of *a1* is subsequently assigned to, it may grow or contract and, in the process, be re-incarnated at a different address in the store. What happens now if a subname referring to an element or subvalue in the old store address has been preserved somewhere?

Clearly, transient names are undesirable things to hang on to, and it is therefore forbidden for them to be either assigned, or ascribed, or passed to or returned by a routine. They may be dereferenced, or stand on the LHS of assignations, so that we may have

(E23) $x := y + a1[i]; a1[2:5] := (a, b, x, y);$

but not

(***) $xx := a1[i]$ ¢ cf. E19 ¢

Care must therefore be taken, when using the **flex** feature (and especially the mode **string**), to avoid these situations. Indeed, these unfortunate, but necessary, restrictions are a significant limitation on the usefulness of flexibility.

(Transient names can also arise in rowing, with the same restriction. Thus we may have

(E24) $[1:1, 1:4]$ **real** *b2;*
$(p \mid b2 \mid a1) :=$ [ ] **real** (a, b, x, y) ¢ *a1* is rowed to **ref** [,] **real** ¢

but not

(***) **ref** [,] **real** *xx2* := *a1;* ¢ cf. E20 ¢

nor even

(***) **ref flex** [,] **real** *xx2* := *a1;*

Note how E24 illustrates the balancing of a fixed name (*b2*) against a transient name (the rowing of *a1*) to give a transient name.)

b) Because it is a primary, it follows that any suitable ENCLOSED-clause can be used, thus:

(E25) **if** $i < j$ **then** *x1* **else** *y1* **fi** $[2:n-1]$

Even a **string** denotation can be trimscripted:

(E26) *"abcd"* [2]

yields *"b"*. However, a row-display cannot be used because its context must

be strong, not weak (3.5.1). However, it can always be cast:

(E27) [ ]**int** *(j, k, l, m)*[*i*]

Of course, slices can always be sliced again:

(E28) *w3*[*i*] [*j*] ¢ see 5.5.0.E4 ¢

c) Because a selection is a secondary, parentheses may be needed when slices and selections are to be combined:

(E29) *p* **of** *q*[*i*]

is only meaningful if *q* is of some mode such as [ ] **struct** *(***amode** *p, ...)*, in which case the *i*th structure is to be sliced from the multiple, and then the field *p* is to be selected from it.

(E30) *(p* **of** *q)* [*i*]

is only meaningful if *q* is of some mode such as **struct***(*[ ] **amode** *p, ...)*, in which case the field *p* (which is a multiple) is to be selected from the structure *q*, and the *i*th element is to be sliced from it. See 1.4. E13 for a similar case concerning a selection which yields a procedure.

5.5.2. Secondaries – multiple selections

Suppose we have a row of structured values (e.g. complex numbers). We may select a row of fields:

(E31) *x*1 := *re* **of** *z1;*
ref [ ] **real** *xx1* := *im* **of** *z1*

Observe that *(re* **of** *z1)*[*i*] and *re* **of** *(z1*[*i*]*)* therefore both select the same **real** value, but by completely different mechanisms!

Again, the possibility of transient names arises, so preventing

(***) **flex** [*1* : *n*] **compl** *c1;*
ref [ ] **real** *xx1* := *re* **of** *c1*

5.5.3. Tertiaries – bound interrogations

It is useful to be able to discover the actual value of the bounds of a multiple which is on hand, particularly so when it is a formal-parameter of a routine, and the bounds of the actual-paramter are needed inside the routine. Two special operators are provided for this purpose:

(E32) *n* **lwb** *x3;*
n **upb** *x3*

These two formulas yield, respectively, the lower- and upper-bounds of the *n*th boundpair (see 1.5.1) of the multiple *x3*. (For example, with *x3* declared as in E11, *3* **upb** *x3* would yield the value *n – 1*).

For getting at the first (or only) boundpair, monadic versions of **lwb** and **upb** are provided. Thus:

(E33) **upb** *x1*

means the same as

(E34) *1* **upb** *x1*

All these operators are introduced formally in 6.5.

5.5.4. Quaternaries – assignations

There are three questions to be answered:

1) What happens when the LHS yields a flexible name (2.5.2.1)?

2) What happens when the LHS is a slice?

(Note that these two questions can never arise together, because a slice cannot yield a flexible name).

3) What happens when the LHS and the RHS involve the same multiple value?

5.5.4.1. Flexible assignations

In the first place, it must be stated that the bounds on the two sides of an assignation must match exactly. Thus:

(***) [*1 : 3*] **real** *xa*, [*2:4*] **real** *xb ; xa* := *xb*

can never be legitimate under any circumstances.

However, if the LHS yields a flexible name (and this can only occur when the whole of some multiple is being assigned to), then the bounds from the RHS are copied across. This means that a flexible multiple may well change its size when the whole of it is assigned to. Given the declaration:

(E35) **flex** [*1 : 0*] **real** *c1*, [*0:n–1*) **real** *d1;*

(E36) *c1* := *d1*

causes *c1* to acquire the bounds [*0*:*n*–*1*], whereas:

(E37) *c1* := *d1*[*2*:*n*–*2*]

causes *c1* to acquire the bounds [*1*:*n*–*3*].

5.5.4.2 Assignation to slices

When the LHS of an assignation is a slice, then of course only the sliced part of the multiple referred to is assigned to. First, however, the bounds of the slice are elaborated, and slid down according to any @s that may be present. These bounds are then compared with those on the RHS to see whether the assignation is legitimate. If it is, the value of the RHS is assigned to the slice on the left (but as selected by the un-slid bounds, of course). Thus, given the declarations:

(E38) [*1*:*3*] **real** *xa*, [*2*:*4*] **real** *xb*, [*1*: *2*] **real** *xc*, [*2*:*3*] **real** *xd*;

the following statements are all legitimate:

(E39)
```
xa [2:3] := xc;
xa [2:3 @ 2] := xd;
xa [2:3] := xd[@ 1];
xa [@ 1] := xb [@ 1]
xa [ : ] := xb [ : ]
```

In these examples the bounds used for comparison purposes are [*1*:*2*] , [*2*:*3*], [*1*:*2*], [*1*:*3*] and [*1*:*3*] respectively, but in all of the first three cases it is [*2*:*3*] of *xa* that get altered. The fourth and fifth cases show how the presumably intended effect of *xa* := *xb* (which is not legitimate) can be achieved.

5.5.4.3. Overlapping slices

Suppose we wish to effect a cyclic permutation of the elements of *x1*. Then we may write

(E40)
```
y1[2 : n] := x1[1 : n–1];
y1[1] := x1[n];
x1 := y1
```

However, we might consider the effect of:

(E41)
```
x := x1[n];
x1[2 : n] := x1[1 : n–1];
x1[1] := x
```

Consider the second line of this:

```
x1[2 : n] := x1[1 : n–1];
```

in which the slice being assigned from overlaps the slice being assigned to. Does this work, or is it equivalent to:

```
for i from 2 to n do x1[i] := x1[i–1] od;
```

(which has rather a disastrous effect)? Fortunately, the overlapping slices of E41 do work correctly, and it is up to the implementation to ensure that it starts the copying operation at the correct end.

Vertical readers, please turn to 6.5.

5.6. Units and unions

(E1) **union** *(* **int, real***) ira, irb;*

ira and *irb* may refer to values of mode either **int** or **real**. However, this raises no problem when one is assigned to the other:

(E2) *ira* := *irb;*

since *ira* now refers to whichever mode *irb* referred to before. The modes on the two sides of E2 are both **ref union** *(* **int, real***)*, and the RHS is dereferenced as usual.

The problems do not begin to arise until we want to set the mode of *irb* in the first place (by assigning an **int** to it, for example) or until we want to get an **int** out of it again (always assuming that it happens to refer to an **int** at the time in question). The first of these problems is dealt with by a new coercion known as "uniting". The solution to the second has already been given in 3.6 (conformity-clauses).

5.6.0. Coercion – uniting

It should be emphasised that there are no built-in-operators for operating on **unions** so *ira* + *irb* is not a valid formula unless you have suitably defined "+" for yourself. This is reasonable because your compiler could not tell whether you were trying to add a **real** to an **int** or an **int** to an **int** or whatever. Therefore, all arithmetic must be done on ununited operands. Once you have a value of some definite mode, however, (and if your context is at least firm) then you may unite it to yield any **union** containing that mode which may be demanded by the context:

(E3) *ira* := *i*+2*;*

Here, *i+2* is of mode **int**. The mode required is **union** *(***int, real***)* and the context is strong (being the RHS of an assignation). Therefore *i+2* is united to be of mode **union** *(***int, real***)* and as such it can now be assigned to *ira*. Because the uniting was from an **int**, *ira* now refers to an **int** value. You might be tempted to think that this example is ambiguous, because the *i+2* might also be widened to **real** and then united. However, if you try to follow through this possibility on the coercion chart given in 5.1.0.2, then you will find that it has been carefully excluded – the only coercions that may precede uniting are the meek ones.

(E4) **union** *(***bool, int, real***)* *bira;*
bira := *ira*

Here, *ira* is united from **union** *(***int, real***)* into **union** *(***bool, int, real***).* This is quite in order because all the constituent modes of the former are also constituents of the latter.

5.6.1 Primaries – the **void** denotation

One of the few places where **void** values are actually useful is in **unions** such as

(E5) **union** *(***real, int, void***)** *riv;*

Here, *riv* may refer to a **real** value, an **int** value, or to no value at all (i.e. a **void** value). We can bring about the last state of affairs using the **void** denotation **empty**:

(E6) *riv* := **empty**;

and of course we can test for this case in a conformity-clause:

(E7) **case** *riv* **in** *(***real** *x)*: *print(x)*,
*(***int** *i)*: *print(i)*,
*(***void***)*: *print("neither")*
esac

5.6.4. Quaternaries – assignations of unions of rows

Multiple values inside **unions** are always declared with formal bounds (2.5.2.3) [R 4.6.1.u]:

(E8) **union** *(*[ ] **int**, [ ] **real**, **bool***)* *irla;*

The effect is much as if the bounds had been preceded by **flex**, insofar as a multiple value of any size (and suitable mode) may be assigned thereto:

(E9) *irla* := *y1*;
irla := *i1*[*17*:*23*];
irla := *i1*

In all these examples, the RHS is united before being assigned. The whole of the multiple value on the RHS (bounds and all) is copied across regardless [R 5.2.1.2.b]. There is no question of checking the existing bounds of the LHS (for if *irla* had previously referred to a **bool**, there would have been none).

A **union** containing multiples cannot be sliced (5.5.1.3), so there is no question of assigning to only a part of it. To get at a part of its existing value, we use a conformity-clause (3.6):

(E10) *il*[*17:23*] := (*irla* | *(*[ ] **int** *ijl)*: *ijl)*[*17:23*]

Note that if *irla* did not refer to a [ ] **int** at this time, the result of E10 would be undefined.

It is, however, possible to discover the bounds of the multiple within a **union** without all this bother, provided the **union** consists of multiples and nothing else [R 10.2.3.1.a]. Thus *irla* as declared in E8 would not do, but if we declare:

(E11) **union** *(*[ ] **int**, [,] **real***)irlb;*

then we can say:

(E12) *i* := **lwb** *irlb;* ¢ yields the lower-bound of the first or only boundpair ¢

p := *2* **upb** *irlb* ¢ yields the second upper-bound, so that it is undefined unless *irlb* is currently exercising its [,] **real** option ¢

Vertical readers, please turn to 7.6.

5.7. Bits and pieces of garbage

5.7.0. Coercion

5.7.0.1. Voiding

We have one coercion left to consider, although we have in fact been using it informally all along. Formally, it is necessary in order to satisfy the general syntactic rule that the mode of each external object must, a posteriori, be that required by its context.

The bulk of any ALGOL 68 program will consist of statements forming the bodies of serial-clauses (3.1.2). Statements are, of course, void-units, but in practice most of them will be assignations which yield a value (the name yielded by the LHS). This value is thrown away by "voiding" [R 6.7]:

(E1) **begin** *x* := *1*: *y* := *2*: *z* := *3* **end**

The first two assignations in this will certainly be voided. Whether the third one is or not depends upon whether the context in which the whole closed-clause occurs expects **void**.

Voiding can occur in strong contexts (but all contexts where **void** is required are strong anyway) and may in most cases be preceded by deproceduring (see coercion chart in 5.1.0.2):

(E2) ....; *x or y*; ¢ see Appendix 2. *x or y* is deprocedured and the next random number is taken but (the context requiring **void**) its **real** result is then thrown away by voiding ¢

However, an assignation (or a cast) must never be deprocedured and then voided, for otherwise in:

(E3) **proc void** *ppp*; *ppp* := *finish*;

we should have to assign *finish* to *ppp* and then, by deproceduring the whole assignation, call the routine now referred to by *ppp* (i.e. *finish*). The assignation is therefore voided straight away.

It is useful to note that the context immediately preceding a ";" is always **void** (and strong).

5.7.0.2. **bits** and **bytes** widening

In strong contexts, **bits** values can be widened to [ ] **bool** and **bytes** values to **string**. The bounds of the [ ] **bool** will always be [*1* : *bits width*] (see

6.2.1) and the **string** will always contain exactly *bytes width* **chars**.

Here is an example in which a slice is trimmed out of a **bits** value. Note the use of a cast to give strength to what would otherwise have been a weak context (5.5.1.3):

(E4) **bits** *t* := *2r1011100;* ¢ for **bits** denotations see next section ¢
[*1* : *3*] **bool** *b1* := [ ] **bool** *(t)* [*bits width–4*: *bits width–2*];
¢ yields (**true**, **true**, **true**) ¢

Similarly, **longs** and **shorts** **bits** and **bytes** can be widened, the upper-bound of the resultant value being given by environment enquiries such as *short bits width, long long bytes width,* etc.

5.7.1. Primaries

5.7.1.1. **bits** denotations

A special form of denotation [R 8.2] is provided for values of mode **bits**:

(E5) **bits** *bits* := *2r101101011001;*

which means that the value assigned to *bits* is the row-display:

*(***false** . . . **false, true, false, true, true, false, true, false, true, true, false, false, true***)*

Note that the correct number of **falses** is automatically inserted at the left hand end. The *2r* means that the **bits** denotation is in binary. Radices *4, 8* and *16* are also possible:

(E6) *4r231121*
8r5531
16rb59

These all yield the same value as that in E5. Note the use of the letters *a – f* to denote the "digits" 10 -15.

Thus if you like quoting your **ints** in octal, you can always write:

(E7) *i* := **abs** *8r12* ¢ meaning *i* := *10* ¢

(For the operator **abs**, see 6.1.1).

5.7.1.2. **long** and **short** denotations

There is no coercion provided in the language for converting a **real** into a

long real or a **long long real.** Therefore the a priori mode of any object must already contain the right number of **longs.** In the case of denotations (**int, real** and **bits**) this is achieved as follows [R 8.1.0.1]:

(E8) **long int** *iiiint* := **long** *122333444455555;* **short int** *it* := **short** *12;*
long long real *reaeal* := **long long** *3.1415926535 8979323846;*
long long long bits *biiiits* := **long long long**
2r10110010110101110001011001011001011011010 11;

There are no **long** forms of **bool, char,** or **string** denotations.

5.7.2. Secondaries – generators

'Generators' [R 5.2.3] are used to make available to the user regions of store where values may be put. They yield the names of those regions. A generator consists of an actual-declarer (2.1.2 and 2.5.2.2), preceded by **loc** or **heap.**

5.7.2.1. **loc** generators

In the case of **loc** generators, the scope (3.2.2) of the name thus created is the lifetime of the "local range" in which the generator appears. The "local range" is the smallest enclosing range (3.2.1) containing the generator which is either

a) a routine-text, or
b) a serial-clause with at least one declaration (3.1.1), except that priority-declarations (4.3.1) do not count, or
c) an enquiry-clause together with the remainder of its choice-clause (3.2.4.2, 3.2.4.3, 3.6) or loop-clause (3.5.2), if that enquiry-clause contains at least one declaration (priority-declarations again excepted).

(E9)
```
begin
[1:n] ref real xx1;
for i to n
    do xx1[i] := case sign i1[i] +2 in x1[i],
                                       nil ,
                                       loc real := 0 esac
    od;
comment At this point, each element of the multiple value
xx1 has been set up referring to either an element of x1 (if the
corresponding element of i1 was negative), or to no value at
```

all (if zero), or to some value specially created for the purpose by the generator **loc real**, and initially set to zero. The number of these special values is determined at run time, according to the values of the elements of *il*. The only way to gain access to them, at the moment, is via the elements of *xx1*. Because a **loc** generator was used, they will all disappear when we leave this range. **comment**
skip
end
¢ At this point, *xx1* has disappeared, and so have any **locally** generated values to which it referred ¢

In this example, the generator **loc real**, each time it was encountered, would reserve storage for one **real** value (presumably on the same stack as *xx1* and *i*) and yield the name referring to that value. *0* would then be assigned to that name, and the name itself would be assigned to *xx1*[*i*].

Although the generator **loc real** in this example is contained within the range lying between the **case** and the **esac**,
which is in turn contained within the range lying between the **do** and the **od**,
which is in turn contained within the range lying between the **for** and the **od** (but excluding the **to** *n*, see 3.5.2),
which is in turn contained within the range lying between the **begin** and the **end**,
only the last range of the four is the local range in question, since the others do not declare anything, neither are they routine-texts.

loc generators are sometimes useful for creating triangular and other oddly-shaped multiples:

```
(E10)   begin
        flex[1 : 0] ref flex [ ] real triangle;
        mode array = flex [1 : 0] real; ¢ to save ink ¢
        triangle := (loc array := 1,
                     loc array := (1,2),
                     loc array := (1,2,3),
                     loc array := (1,2,3,4) );
        for i to 4 do print (triangle [i] [i]) ¢ prints the diagonal ¢
        end
```

Outside the range of E10, both *triangle* and the **arrays** to which it referred will have vanished.

The slice *triangle* [*i*] [*i*] is worthy of further examination. *triangle* itself is of mode **ref flex** [ ] **ref flex** [ ] **real**. *triangle* [*i*] is a slice of mode **ref ref flex** [ ] **real** (for the reasons explained in 5.5.1.3). It yields the name of the name of the *i*th row of *triangle* (it is impossible to get hold of the columns). In order to be able to take a further slice out of *triangle* [*i*], we must demonstrate that it is a weak primary of mode **ref flex** [ ] **real**. Now a slice is a primary (5.1.0.1) and a weak primary of mode **ref flex** [ ] **real** can be obtained by dereferencing a slice of mode **ref ref flex** [ ] **real**, such as *triangle* [*i*], and there we are. We can make the further slice *triangle* [*i*] [*i*], and the mode it yields is **ref real**. In E10 this was then dereferenced once more so that a **real** value could be printed.

Now, we shall remind you for the last time that:

(E11) **real** *x* ;

means exactly the same (2.2.2) as:

(E12) **ref real** *x* = **loc real**;

x is here declared to be of mode **ref real** and the name of the piece of store made available by the generator **loc real** has been ascribed to it. Since it is a **loc** generator, the piece of store will cease to be available outside the range in which E11 or E12 appeared. Also, outside this range, the identifier *x* cannot occur (or if it does, it identifies something completely different). Thus *x* and the name which it yields rise and fall together.

5.7.2.2. **heap** generators

In the case of **heap** generators, the scope (3.2.2) of the name that is created is not restricted to the lifetime of any range:

```
(E13)   begin
        real w;
        w := 10.5;
        xx := heap real := w ¢ creates an extra instance of 10.5
                                on the heap ¢
        end;
          comment now we are outside the reach of w and of the
              first instance of 10.5 to which it referred. However, the
              second instance of 10.5 is still intact, and is accessible
              via the variable xx comment
        print (xx); ¢ prints 10.5 (after dereferencing xx twice) ¢
```

```
xx := x    ¢ the instance of 10.5 on the heap is now quite
             inaccessible, because no-one now refers to it ¢
```

In this example, when the **heap** generator **heap real** was encountered, storage for one **real** value was reserved (but not on the main stack such as was used by *w* – thus a different region of store, usually termed the "heap", is involved). The generator yielded the name referring to this piece of storage, the value *10.5* was assigned to it, and it was assigned to *xx* (which is of mode **ref ref real** – see Appendix 2). Both *xx* and this value remained fully available outside the range in which **heap real** occurred, and were used in the *print*. However, after this, *xx* was used for something else and the *10.5* was just left sitting there.

Thus it is very easy to waste large amounts of the heap:

```
(E14)    to 10000 do heap real od
```

this will reserve 10,000 words on the heap, and there will be no way of accessing any of them – they will in fact be "garbage". Therefore it will be necessary for your implementation to include in your run-time program a "garbage collection routine" which will be called in whenever the size of the heap has become embarrassingly large. How this works in detail is your implementor's worry, but it will go something like this:

1) Consider all the values (on the stack) which are names that have been ascribed or assigned to identifiers (i.e. all identifiers declared with mode **ref ref amode** and some with **ref amode** within the current range, or its surrounding ranges).
2) If any such name refers to a value on the heap, mark that area of the heap as useful (this could be done by a vast array of bits, one for each word on the heap). Since the mode of the name is always known, the size of the value can easily be determined.
3) If the value referred to by any such name contains further names within itself (again, this will be apparent from the known mode of the given name), then consider these names also.
4) Go through the array of bits searching for areas of the heap that have not been marked as useful. These areas can now be made available for further use.

This process will be recognised as being similar to that employed in list-processing languages such as LISP, and it is for applications in which list-processing would otherwise have had to be used that **heap** generators are primarily intended.

You can, if you like, do fairly conventional list-processing in this language:

(E15)
```
mode atom = union (char, int);
mode cons = struct (union (atom, ref cons) car, ref cons cdr);
proc list = ( [ ] union (atom, ref cons) item) ref cons:
        begin
        ref cons a := nil;
        for i from upb item by -1 to 1
                do a := heap cons := (item [i], a) od;
        a
        end;
ref cons expression := list (("X", "+", list(("Y", "x", 2))));
```

However, if you intend to create many lists with the same layout, it is better to declare them as **structs**, and generate them as such:

(E16)
```
mode operand = union (char, int, ref expression);
mode expression =
        struct ( operand left, char operator, operand right);
ref expression expression := heap expression :=
        ("X", "+", heap expression := ("Y", "x", 2));
```

This version is more convenient to write, will use less storage space, and will have its garbage collected more speedily (since a complete **expression** can be removed at one go).

As was mentioned in 2.7.3, the declaration:

(E17)
```
heap real x;
```

means exactly the same as:

(E18)
```
ref real x = heap real;
```

x is here declared to be of mode **ref real** and to yield the name of the piece of store made available by the generator **heap real**. This piece of store will still be available outside the range in which E17 or E18 appeared, even though the identifier *x* cannot occur there (or if it does, it identifies something completely different). It could be accessed in the following circumstances, which should be compared with E13:

(E19)
```
begin
heap real w;
w := 10.5;
xx := w
end ;
print(xx); ¢ prints 10.5 ¢
xx :=x ¢ the instance of 10.5 on the heap is now garbage ¢
```

5.7.3. Tertiaries – order of elaboration of operands

The elaboration of a dyadic-formula involves the elaboration of two operands (these are either other formulas or secondaries). These two operands are elaborated collaterally (see 3.7.1). The following dangerous example illustrates this, and should be compared with 3.7.1. E2:

(E20)
```
begin int i;
proc side = int: (i:=1; i:=2, i);
print ( side + side )
end
```

This will print either 3 or 4, for the reasons given in 3.7.1.

The advantage of this collateral elaboration from the point of view of implementation is that the order of elaboration can be chosen to be that which yields the minimum number of compiled instructions. For example:

(E21) $y := x + a \times b$

Most compilers will choose to fetch and multiply a and b before getting hold of x.

5.7.4. Quaternaries – identity relations

Identity-relations are used to detect whether two names are identical:

(E22)
```
ref real anotherx = x; real y;
anotherx :=: x;  ¢ always yields true ¢
anotherx :≠: x;  ¢ always yields false ¢
anotherx :=: y   ¢ always yields false ¢
```

This example is quite trivial, because *anotherx* and x, by virtue of the identity-declaration, both yield the same name.

(E23) *xx* := *yy* := *x*;
(E24) *xx* :=: *x* ; ¢ yields **true** ¢
(E25) *x* :=: *xx*; ¢ yields **true** ¢
(E26) *xx* :=: *yy* ¢ yields **false** ¢

In E24 and E25, *xx* was dereferenced to yield the name to which it currently referred (i.e. *x*). It is permissible to dereference on one side of an identity-relation, but not on both, and it is for this reason that E26 did not work. E26 could never yield **true**, whatever assignations we might make to *xx* and *yy*. However, by the use of casts we can achieve the result presumably intended:

(E27) **ref real** *(xx)* :=: **ref real** *(yy)* ¢ yields **true** (in the context of E23) ¢

The rules governing the use of an identity-relation are the following:

It has two sides – a strong side and a soft side. One of these can stand as the LHS, in which case the other must stand as the RHS. The symbol in between is either ":=:" or ":≠:" (which may alternatively be written as **is** or **isnt**).

For its strong side:
a) It must yield a name (i.e. its mode must be **ref amode**).
b) It must be a tertiary.
c) Its context is strong (so that dereferencing is allowed, and also **nil** (5.2.3)).

For its soft side:
a) It must yield a name, of the same mode as that yielded by the strong side.
b) It must be a tertiary.
c) Its context is soft (so that deproceduring is the only possible coercion, and **nil** is not permitted).

As a whole, an identity-relation yields a **bool** value – **true** if the names match and the symbol in between is ":=:" or if the names do not match and the symbol is ":≠:".

Here is a delicately balanced example:

(E28) **case** *i* **in** *xx*, *x* **or** *y* **out nil esac**
:=:
case *j* **in** *yy*, **skip, heap ref real** := *x* **esac**

First, let us rewrite the example showing the a priori modes of all the items:

```
case i in xx,                ¢ ref ref real    ¢
        x or y               ¢ proc ref real   ¢
     out nil                 ¢ wait and see    ¢
esac :=:
case j in yy                 ¢ ref ref real    ¢
        skip,                ¢ wait and see    ¢
        heap ref real:=x     ¢ ref ref real    ¢
```

Which is the soft side, and what is the mode of the name that it yields? Well, to put you out of your misery, the LHS is the soft one and the mode is **ref real**, but to obtain it we have to recall that case-clauses can be balanced (5.2.0.1) so that, even on the soft side, all but one of the items can be strongly coerced. Here then is the example again with all made clear:

```
case i in xx,                ¢ strongly dereferenced ¢
        x or y               ¢ softly deprocedured ¢
     out nil                 ¢ a strong context ¢
esac :=:
case j in yy,                ¢ strongly dereferenced ¢
        skip,                ¢ a strong context ¢
        heap ref real :=x    ¢ strongly dereferenced ¢
```

For an application of identity-relations, let us return to our list processing in E15. Let us assume all the declarations of E15 to have been made, and now continue thus:

```
(E29)   op eql = (union( atom, ref cons ) a, b) bool:
                    case a
                    in (char c): (b|(char d): c=d, (int): false|b eql c),
                      (int i): (b|(int j): i=j, (char): false|b eql i),
                      (ref cons rc): car of rc eql b
                    esac;
        comment     this recursively defined (and probably not very
                    efficient) operator compares two cars (or cdrs)
                    of conss and yields true if they are, or refer to
                    via a chain of cars, identical atoms comment
        ref cons a := list(("A", "X"));
        ref cons b := list(("B", "X"));
        cdr of cons(a) eql cdr of cons(b);
          ¢ yields true because both sides refer to the value "X" ¢
        cdr of cons(a) :=: cdr of cons(b)
          ¢ yields false because both sides refer to different
            instances of the value "X" ¢
```

The distinction here illustrates how, in list processing, it is often important to distinguish between a pointer to a list which is merely a copy of a given one, and a pointer which points to the given list itself. The identity-relation should be used to make this test.

Vertical readers, please turn to 6.7.

6. STANDARD PRELUDE

6.1. Operators

Each particular-program written by a user is presumed to be included within an "outer range", at the head of which is the standard-prelude (1.1) in which various standard declarations are made. These include:

Standard constants (see 6.2 and 6.7)
Standard procedures (see 6.2 and 6.7)
Standard operators (see this section, and 6.3, 6.5 and 6.7)
Additional constants and procedures required for transput (see 7).

Likewise, at the tail of the outer range, is a label *stop*:, to which you may jump in order to terminate the elaboration of your program, and which is followed by the standard-postlude (see 1.1 and Appendix 3).

We now set out, in tabular form, details of all the common operators (for the manner in which they are used in formulas see 5.1.3). The tables include, for completeness, all of the meanings which each operator can have, even though you may not yet be familiar with all of the modes involved. The meaning of each operator is given in the last column for those operators whose meaning is not obvious. If nothing appears in this column, it means that the generally accepted meaning applies, or that a similar operator has already been explained higher up the column.

There are sometimes several operators which perform the same function, in which case they are all given in the first column. Not all implementations will provide all the versions, however.

6.1.1. Monadic operators

operator	priority	mode of *a*	mode of result	meaning
¬ ~ **not**	10	**bool** **bits**	**bool** **bits**	
+	10	**int** **real** **compl**	**int** **real** **compl**	
–	10	**int** **real** **compl**	**int** **real** **compl**	

operator	priority	mode of *a*	mode of result	meaning
bin	10	**int**	**bits**	the binary digits representing the positive integer *a*
re	10	**compl**	**real**	the real part
im	10	**compl**	**real**	the imaginary part
conj	10	**compl**	**compl**	**re** *a* **i** – **im** *a*
abs	10	**bool**	**int**	*1* for **true** and *0* for **false**
		int	**int**	
		real	**real**	
		compl	**real**	*sqrt(***re***a*↑*2* + **im***a*↑*2)*
		bits	**int**	the opposite of **bin**
		char	**int**	a unique integer for each permissible value of **char**
arg	10	**compl**	**real**	the argument of *a*, $-\pi < \textbf{arg}\ a \leqslant \pi$
odd	10	**int**	**bool**	**true** if odd, **false** if even
sign	10	**int**	**int**	yields *–1*, *0*, or *+ 1*
		real	**int**	
round	10	**real**	**int**	the nearest integer
entier ⌊	10	**real**	**int**	the integer equal to *a*, or the next integer below (more negative than) *a*
repr	10	**int**	**char**	the opposite of **abs** of a **char**

6.1.2. Dyadic operators

operator	priority	mode of *a*	mode of *b*	mode of result	meaning
i	9	**real**	**real**	**compl**	*a* plus i times *b*
⊥		**int**	**int**	**compl**	
+×		**int**	**real**	**compl**	
+*		**real**	**int**	**compl**	

oper-ator	prior-ity	mode of *a*	mode of *b*	mode of result	meaning
↑ ****** **up**	8	**int** **real** **compl**	**int** **int** **int**	**int** **real** **compl**	a^b where $b \geqslant 0$ a^b a^b
↑ **shl** **up**		**bits**	**int**	**bits**	*a* shifted left *b* places (or right for *b* negative)
↓ **shr** **down**	8	**bits**	**int**	**bits**	*a* shifted right *b* places (or left for *b* negative)
÷ % **over**	7	**int**	**int**	**int**	**abs** *(a ÷ b)* = **entier abs** *(a/b)* i.e. truncation towards zero
mod ÷x ÷* %x %*	7	**int**	**int**	**int**	$0 \leqslant a$ **mod** $b < b$
× *	7	**int** **real** **compl** **real** **int** **compl** **compl** **int** **real** **string** **int** **char** **int**	**int** **real** **compl** **int** **real** **int** **real** **compl** **compl** **compl** **int** **string** **int**	**int** **real** **compl** **real** **real** **compl** **compl** **compl** **compl** **string** **string** **string** **string**	*a* replicated *b* times *b* replicated *a* times
/	7	**int** **real** **compl** **real** **int** **compl** **compl** **int** **real**	**int** **real** **compl** **int** **real** **int** **real** **compl** **compl**	**real** **real** **compl** **real** **real** **compl** **compl** **compl** **compl**	
elem □	7	**int** **int**	**bits** **bytes**	**bool** **char**	the *a* th bit of *b* the *a* th **char** of *b*

operator	priority	mode of *a*	mode of *b*	mode of result	meaning
+	6	int	int	int	
		real	real	real	
		compl	compl	compl	
		real	int	real	
		int	real	real	
		compl	int	compl	
		compl	real	compl	
		int	compl	compl	
		real	compl	compl	
		string	string	string	the concatenation of *a* and *b*
		string	char	string	
		char	string	string	
		char	char	string	
–	6	int	int	int	
		real	real	real	
		compl	compl	compl	
		real	int	real	
		int	real	real	
		compl	int	compl	
		compl	real	compl	
		int	compl	compl	
		real	compl	compl	
< lt	5	int	int	bool	
		real	real	bool	
		real	int	bool	
		int	real	bool	
		char	char	bool	true if abs *a* < abs *b*
		string	string	bool	true if the first character in *a* that differs from the corresponding character in *b* is less than same
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	
> gt	5	int	int	bool	
		real	real	bool	
		real	int	bool	
		int	real	bool	
		char	char	bool	
		string	string	bool	
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	
≤ <= le	5	int	int	bool	
		real	real	bool	
		real	int	bool	
		int	real	bool	
		bits	bits	bool	true if each bit in *a* implies the corresponding bit in *b*
		char	char	bool	
		string	string	bool	
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	

operator	priority	mode of *a*	mode of *b*	mode of result	meaning
⩾	5	int	int	bool	
>=		real	real	bool	
ge		real	int	bool	
		int	real	bool	
		bits	bits	bool	
		char	char	bool	
		string	string	bool	
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	
=	4	bool	bool	bool	
eq		int	int	bool	
		real	real	bool	
		compl	compl	bool	
		real	int	bool	
		int	real	bool	
		compl	int	bool	
		compl	real	bool	
		int	compl	bool	
		real	compl	bool	
		bits	bits	bool	
		char	char	bool	
		string	string	bool	
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	
≠	4	bool	bool	bool	
/=		int	int	bool	
ne		real	real	bool	
		compl	compl	bool	
		real	int	bool	
		int	real	bool	
		compl	int	bool	
		compl	real	bool	
		int	compl	bool	
		real	compl	bool	
		bits	bits	bool	
		char	char	bool	
		string	string	bool	
		string	char	bool	
		char	string	bool	
		bytes	bytes	bool	
∧	3	bool	bool	bool	
&		bits	bits	bits	
and					
∨	2	bool	bool	bool	
or		bits	bits	bits	

Vertical readers, please turn to 7.1.

6.2. Constants and procedures

6.2.1. Constants

With the exception of *pi*, the purpose of these constants is to give information about the implementation upon which the program is being run, and they are therefore called "environment enquiries". They are all declared in the standard-prelude [R 10.2.1], by means of identity-declarations, to be of some mode such as **int** or **real**. Hence they are not names, and hence they cannot be altered by the user.

Some further environment enquiries are given in 6.7.1 (in connection with **long** modes) and in 7.2.2 and 7.5.3 (in connection with transput).

identifier of constant	mode	value
max int	**int**	the largest **int** value which can be represented
max real	**real**	the largest **real** value which can be represented
small real	**real**	the smallest **real** value which can be meaningfully added to or subtracted from *1*
bits width	**int**	the number of bits in **bits** (see 2.7.1)
bytes width	**int**	the number of **chars** in **bytes** (see 2.7.1)
max abs char	**int**	the largest value which **abs** of a **char** can yield
null character	**char**	some character (see *bytespack* in 6.2.2)
blank	**char**	"␣" (the space character)
pi	**real**	π

6.2.2. Procedures

The following procedures are all declared within the standard-prelude [R 10.2.3.12, 10.5.1] to be of some mode **proc amode**, rather than **ref proc amode**. Hence they cannot be altered by the user. Their meanings are those generally accepted, or as specified by the last column of the following table.

Further procedures from the standard-prelude are to be found in 6.7.2, 7.1.1, 7.1.2, 7.2.3, 7.2.4, 7.2.5, 7.4.2, 7.4.3, 7.5.1, 7.6.3 and 7.7.1.

identifier of **proc**	mode	
sqrt	**proc(real)real**	
exp	**proc(real)real**	
ln	**proc(real)real**	
cos	**proc(real)real**	
arccos	**proc(real)real**	$0 \leqslant arccos(x) \leqslant \pi$
sin	**proc(real)real**	
arcsin	**proc(real)real**	$-\pi/2 \leqslant arcsin(x) \leqslant \pi/2$
tan	**proc(real)real**	
arctan	**proc(real)real**	$-\pi/2 \leqslant arctan(x) \leqslant \pi/2$
next random	**proc(ref int** *a***)real**	The next **int** value after *a* from a pseudo-random sequence uniformly distributed in the range $0 \leqslant a \leqslant max\ int$ is assigned to *a*. The yield is a **real** value x $(0 \leqslant x < 1)$ obtained by means of some uniform mapping from *a* (such that *x* is also pseudo-random and uniformly distributed)
last random	**ref int**	an **int** variable, initialised to **round**$(max\ int/2)$, which is used by *random* (below)
random	**proc real**	a call of *next random*, using *last random* as parameter
bitspack	**proc([]bool** *a***)bits**	the multiple *a*, made up with **false**s at the left, is turned into **bits**
bytespack	**proc(string** *a***)bytes**	the string *a*, made up with *null character*s (6.2.1) on the right, is turned into **bytes**

See 6.7.2 for long(s) versions of these.

Vertical readers, please turn to 7.2.

6.3. Assigning operators

The operators in the following table all have the property that the result of the operation is automatically assigned to the name of the left hand operand, and this name is yielded as the value of the formula. Thus:

(E1) *a* **plusab** *b* or *a* +:= *b* ¢ means the same as *a* := *a*+*b* ¢
(E2) *x* := *a* **plusab** *b* ¢ means the same as *x* := *a* := *a*+*b* ¢
(E3) *a* **minusab** *b* **plusab** *c*

E3 has implied bracketing:

(E4) *(a* **minusab** *b)* **plusab** *c* ¢ and therefore means *a* := *a*–*b*; *a* := *a*+*c* ¢
(E5) *a* **plusab** *b* := *x*

E5 is legitimate example of a formula on the LHS of an assignation, but it is not very sensible since *b* does not enter into the result.

oper-ator	prior-ity	mode of *a*	mode of *b*	mode of result	meaning
timesab x:= *:=	1	**ref int**	**int**	**ref int**	*a* := *a*×*b*
		ref real	**real**	**ref real**	
		ref compl	**compl**	**ref compl**	
		ref real	**int**	**ref real**	
		ref compl	**int**	**ref compl**	
		ref compl	**real**	**ref compl**	
		ref string	**int**	**ref string**	*a* := *a*×*b* (× implying replication)
overab ÷:= %:=	1	**ref int**	**int**	**ref int**	*a* := *a*÷*b*
divab /:=	1	**ref real**	**real**	**ref real**	*a* := *a*/*b*
		ref compl	**compl**	**ref compl**	
		ref real	**int**	**ref real**	
		ref compl	**int**	**ref compl**	
		ref compl	**real**	**ref compl**	
modab ÷x:= ÷*:= %x:= %*:=	1	**ref int**	**int**	**ref int**	*a* := *a* **mod** *b*

oper-ator	prior-ity	mode of *a*	mode of *b*	mode of result	meaning
plusab	1	**ref int**	**int**	**ref int**	*a* := *a*+*b*
+:=		**ref real**	**real**	**ref real**	
		ref compl	**compl**	**ref compl**	
		ref real	**int**	**ref real**	
		ref compl	**int**	**ref compl**	
		ref compl	**real**	**ref compl**	
		ref string	**string**	**ref string**	*a* := *a*+*b* (+ implying
		ref string	**char**	**ref string**	concatenation)
plusto	1	**string**	**ref string**	**ref string**	*b* := *a*+*b*
+=:		**char**	**ref string**	**ref string**	
minusab	1	**ref int**	**int**	**ref int**	*a* := *a*−*b*
−:=		**ref real**	**real**	**ref real**	
		ref compl	**compl**	**ref compl**	
		ref real	**int**	**ref real**	
		ref compl	**int**	**ref compl**	
		ref compl	**real**	**ref compl**	

Vertical readers, please turn to 8.3.

6.5. Interrogations

The following table specifies the operators introduced informally in 5.5.3. Note that **amode** stands for any mode and ",s" stands for any number of commas (including none).

6.5.1. Dyadic operators

operator	priority	mode of *a*	mode of *b*	mode of result	meaning
lwb ⌊	8	**int**	[,s] **amode**	**int**	the lower bound of the *a* th subscript of *b*
upb ⌈	8	**int**	[,s] **amode**	**int**	the upper bound of the *a* th subscript of *b*

6.5.2. Monadic operators

operator	priority	mode of *a*	mode of result	meaning
lwb ⌊	10	[,s] **amode**	**int**	*1* **lwb** *a*
upb ⌈	10	[,s] **amode**	**int**	*1* **upb** *a*

Vertical readers, please turn to 7.5.

6.7. Long operators

6.7.1. Environment enquiries

The numbers of different lengths and shorths of **ints**, **reals**, etc. that are provided may vary between different implementations. Environment enquiries (6.2.1) are therefore provided to indicate these numbers [R 10.2.1]. Additionally, the environment enquiries introduced in 6.2.1 (*max int*, etc.) and in 7.5.3 (*int width*, etc.) have long versions of themselves. Note that the number of different "lengths" or "shorths" of each mode includes none at all, so that if the implementation distinguishes just **short int**, **int** and **long int**, then both **int lengths** and **int shorths** will have the value *2*.

identifier of constant	mode	value
int lengths	**int**	the number of different lengths of **ints**
int shorths	**int**	the number of different shorths of **ints**
real lengths	**int**	the number of different lengths of **reals** (and of **compls**)
real shorths	**int**	the number of different shorths of **reals** (and of **compls**)
bits lengths	**int**	the number of different widths of **bits**
bits shorths	**int**	the number of different shorths of **bits**
bytes lengths	**int**	the number of different widths of **bytes**
bytes shorths	**int**	the number of different shorths of **bytes**
long max int	**long int**	the largest **long int** value which can be represented
long long max int	**long long int**	
long long long max int		

and so on, up to any number of *longs*. For the rest of this table, let us introduce the convention that "*long*(s)" means any number of *longs* or *shorts* and "**long**(s)" means the same number of **longs** or **shorts**.

long(s) *max real*	**long**(s) **real**	the largest **long**(s) **real** value which can be represented
long(s) *small real*	**long**(s) **real**	the smallest **long**(s) **real** value which can be meaningfully added to or subtracted from *1*.
long(s) *bits width*	**int**	the number of bits in **long**(s) **bits**
long(s) *bytes width*	**int**	the number of **chars** in **long**(s) **bytes**
long(s) *int width*	**int**	the number of decimal digits required to represent *long*(s) *max int* – not including sign
long(s) *real width*	**int**	the number of decimal digits required to represent a mantissa, such that *long*(s) *small real* is not neglected in comparison with *1* – not including sign
long(s) *exp width*	**int**	the number of decimal digits required to represent a decimal exponent, such that *long*(s) *max real* can be correctly represented – not including sign
long(s) *pi*	**long**(s) **real**	π

6.7.2. Procedures

The procedures introduced in 6.2.2 [R 10.2.3.12, 10.5.1] also have their long and short versions. Note that a procedure with, for example, a **long long real** formal-parameter yields a value whose mode has exactly the same number of **longs** in it.

identifier of **proc**	mode
long sqrt	**proc(long real)long real**
long long sqrt	**proc(long long real)long long real**
and so on. We shall adopt the same abbreviation as before.	
long(s) *exp*	**proc(long(s) real)long(s) real**
long(s) *ln*	**proc(long(s) real)long(s) real**
long(s) *cos*	**proc(long(s) real)long(s) real**
long(s) *arccos*	**proc(long(s) real)long(s) real**
long(s) *sin*	**proc(long(s) real)long(s) real**
long(s) *arcsin*	**proc(long(s) real)long(s) real**
long(s) *tan*	**proc(long(s) real)long(s) real**
long(s) *arctan*	**proc(long(s) real)long(s) real**
long(s) *next random*	**proc(ref long(s) int)long(s) real**
long(s) *last random*	**ref long(s) int** . . . . initialized to **round**(*long*(s) *max int/2*)
long(s) *random*	**proc long(s) real** . . . . which uses *long*(s) *last random*: see 6.2.2
long(s) *bitspack*	**proc([ ]bool)long(s) bits**
long(s) *bytespack*	**proc(string)long(s) bytes**

6.7.3. Operators

Most of the operators introduced in 6.1.1, 6.1.2 and 6.3 have their **long**(s) and **short**(s) counterparts [R 10.2.3]. We shall not list them all here; instead we shall give you a rule for working them out yourself.

Each operator has one or two parameters and a result, each being of some mode. If one or more of these modes is:

int, real, compl bits or **bytes**

then new version of that operator can be obtained by inserting **long**(s) or **short**(s) in front of each of those modes. However, the modes **bool, char** and **string**, wherever they occur, must be left strictly alone.

For example, one of the versions of the operator "+" can be used to add a **real** to an **int** yielding a **real** (6.1.2). There therefore exists another version which adds a **long long real** to a **long long int** yielding a **long long real** (but not to add a **long long real** to a **long int** yielding a **real** – the number of **longs**

added must be the same throughout). Likewise, a **short real** can be added to a **short int** yielding a **short real.**

However, there are certain exceptions to this general rule, all of which are concerned with not allowing **long(s) ints** in places where they would clearly be ridiculous. Thus:

The **abs** of a **char** yields a (single) **int**
The **repr** of a (single) **int** yields a **char**
long(s) ints, **reals** and **compls** can be raised to a (single) **int** power, yielding correspondingly **long(s) ints** etc.
strings and **chars** can be replicated a (single) **int** number of times using **x**, * or **timesab**
The (single) **int** th **element** of a **long(s) bits** or **bytes** yields a **bool** or a **char**
The **level** of a (single) **int** yields a **sema** and vice versa (3.7.2)

In all of these, the phrase "(single) **int**" implies **int** where **long(s) int** or **short(s) int** might otherwise have been expected.

6.7.4. leng and shorten

There are no coercions provided in the language for converting, for example, **ints** into **long ints** or vice versa. Instead, you are provided with the monadic-operators **leng** and **shorten** (see 8.4.2 for a meaningful example of their use):

operator	priority	mode of *a*	mode of result	meaning
leng	10	**long(s) int**	**long long(s) int**	the **longer** value equivalent to *a*
		long(s) real	**long long(s) real**	
		long(s) compl	**long long(s) compl**	
		long(s) bits	**long long(s) bits**	making up with **falses** at the left
		long(s) bytes	**long long(s) bytes**	making up with *null characters* (6.2.1) on the right
shorten	10	**long long(s) int**	**long(s) int**	the **shorter** value equivalent to *a*, if it exists
		long long(s) real	**long(s) real**	
		long long(s) compl	**long(s) compl**	
		long long(s) bits	**long(s) bits**	truncating on the left
		long long(s) bytes	**long(s) bytes**	truncating on the right

In this table, as usual, "**long**(s)" means **long** or **short** repeated zero or more times (but the same number of times in each column). Also, of course, **leng** of a **short short int** yields a **short int**, **leng** of a **short int** yields an **int**, and so on. Note that if you try to **shorten**, for example, a **long int** which is greater than *max int* (6.2.1), then the result is undefined.

6.7.5. **up** and **down**

The operators **up**, **down** and **level**, as applied to **sema**s, were defined in 3.7.2.

Vertical readers, please turn to 7.7.

7. TRANSPUT

7.1. Formatless transput

"Transput" is the name given to all those operations which communicate with the environment. These include input, output and transfers to backing media such as magnetic tape and discs.

There is a large variety of transput facilities provided to suit the user's taste, ranging from the simplest formatless transput described in this section, through the facilities for accessing various types of device (7.2) and for dealing with exceptional situations (7.4.4) up to the formatted transput described in 7.6 and the binary transput in 7.7.

7.1.1. Formatless output

Formatless output is achieved by means of the procedure *print*, e.g.:

(E1) *print (x); print (i+3); print (p & i<j); print ("A");*
print ("ABC"); print (x **i** *y); print (***bin** *5); print (r)*

The modes of the actual-parameters in these examples are, respectively, **real**, **int**, **bool**, **char**, **string**, **compl**, **bits** and **bytes** (for a description of the mode **string** see 2.5.3 and 5.5.1.1, for **compl** see 2.4.4 and 5.4.3 and for **bits** and **bytes** see 2.7.1 and 5.7.1.1). In addition, multiple values (1.5.1, 2.5) and structures (1.4.1, 2.4) made up of any of these modes can be used. The actual-parameter of this procedure is a firm unit (the firmness arises from the particular way in which it is defined in the standard-prelude), which means that widening is not allowed, but dereferencing is. Note that it is not possible to output names (i.e. modes beginning with **ref**), or **formats** (7.6.2) or routines.

It is possible to output more than one item with one call on *print*:

(E2) *print ((x, i+3, p & i<j, "A", "ABC", x* **i** *y,* **bin** *5, r))*

which is equivalent to the series of separate *print*s in E1 above. Note the additional parentheses (these are required because the "data list" of items is really a row-display (see 3.5.1)).

The following layout procedures [R 10.3.1.6] may be called upon in between calls of *print*:

identifier of proc	mode	
newpage	**proc(ref file)void**	continue printing at the beginning of the next page
newline	**proc(ref file)void**	continue printing at the beginning of the next line
space	**proc(ref file)void**	skip one character (which results in a space character unless cunning use has been made of *backspace*)
backspace	**proc(ref file)void**	move back one character (but not beyond the start of the current line). A subsequent call on *print* will overwrite whatever character was previously there (but a call on *space* will not).

These layout procedures (and other **proc(ref file)voids** (see 7.2.5) written by the user) may also be called upon within a *print* call. Thus:

(E3) *print ((newpage, "HEADING", newline, "X = " , x))*

means the same as:

(E4)
```
newpage (stand out);
print ("HEADING");
newline (stand out);
print (("X = " , x))
```

The *stand out* parameter specifies the **file** (see 7.2.1) to be affected. The **file** *stand out* is automatically implied by *print*, and is therefore supplied as the parameter of these routines when they are called from within it.

When *print* is called, the mode of each item to be printed is identified, and appropriate action taken as follows:

ints, reals, compls:

If there is not room for the item on the current line (page), then *newline* (*newpage*) is called. Then the item is printed, preceded by a space (if not at the beginning of a line), allowing sufficient positions to cope with the largest permissible value of that mode.

Examples are:

```
+123456
   +456
     -1
     +0
+1.23456₁₀+11
-6.54321₁₀ -2
+1.23456₁₀+11 ⊥-6.54321₁₀ -2
```

chars, bools:

newline or *newpage* is called if necessary as above, and then the item is output with no preceding space. The (single) characters to be printed for **true** and **false** are to be decided by the implementer (see 7.5.3). In this book we use T and F.

strings, bytes, bits:

These are treated as sequences of **chars** or **bools** as appropriate, *newline* and *newpage* are called wherever required (and thus a **string** may get split over a line – but see 7.4.4 for how to control this).

If an item is a multiple (structure), then its component elements (fields) are output in turn according to the above rules. This is discussed more fully in 7.5.1 (7.4.1) under the heading of "straightening".

For those who prefer it, the procedure *write* may be used instead of *print.*

7.1.2. Formatless input

This is achieved by means of the procedure *read*, e.g.:

(E5) *read (x); read (i); read (p); read (c); read (s); read (z); read (t); read (r)*

The modes of the actual-parameters in these examples are, respectively:

ref real, ref int, ref bool, ref char, ref string, ref compl, ref bits, ref bytes

In addition, references to multiples and references to structures made up of any of the modes referred to can be used. The actual-parameter is in fact a firm unit, which means that

(E6) *read (xx)*

is allowed, where *xx* is of mode **ref ref real** and must be dereferenced once.

The quantity actually input, of course, is of mode **real**, **int**, etc. The actual-parameter yields the name of the place where it is to be put. Note that it is not possible to input names, or **formats**, or routines or **unions** (1.6, 2.6).

It is possible to input more than one item with one call on *read*:

(E7) *read ((x, i, p, c, s, z, t, r))*

which is entirely equivalent to the series of separate *reads* given above. Note the additional parentheses (these are required because the "data list" of items is really a row-display (see 3.5.1)).

The following layout procedures [R 10.3.1.6] may be called upon in between calls of *read*:

identifier of **proc**	mode	
newpage	**proc(ref file)void**	ignore the rest of the current page and start reading the next
newline	**proc(ref file)void**	ignore the rest of the current line, and start reading the next
space	**proc(ref file)void**	ignore the next character
backspace	**proc(ref file)void**	move back one character (but not beyond the start of the current line). A subsequent call on *read* will yield the last character again

These layout procedures (and other **proc(ref file)voids** (see 7.2.5) written by the user) may also be called upon within a call of *read.* Thus:

(E8) *read ((newpage, s, newline, x))*

means the same as:

(E9)
```
newpage (stand in);
read (s);
newline (stand in);
read (x)
```

The *stand in* parameter specifies the **file** (see 7.2.1) to be affected. The **file** *stand in* is automatically implied by *read*, and is therefore supplied as the parameter of these routines when they are called from within it.

When *read* is called, the mode of each item required is identified, and appropriate action taken as follows:

ints, reals, compls, bools:

The input stream is searched for the first character that is not space (*newline*

and *newpage* being called as necessary). When it is found, the required item is read in (note that when **real** is called for, an **int** will suffice). If no recognisable item is found, then the result is undefined (unless the user has called the *on char error* procedure (see 7.4.4.7). The following examples are acceptable:

```
+123456
+   456
   +456
-1
123456
12.3456
  .3456
-  12.34
1.23456₁₀+11
1.23456e-2
12₁₀12
12.34 ⊥ 1.23456₁₀12
T (for true)
F (for false)
```

" i " may be accepted in place of "⊥", and " e " or "\" in place of "$_{10}$".

chars:

The next-character (possibly space) is read from the input stream, *newline* or *newpage* being first called if necessary.

strings (i.e. **flex** [ ] **chars**):

Characters are read from the current position until either the end of the current line is reached, or (if the user has called *make term* (see 7.4.2)) one of the terminating characters is found (this character is not yielded as part of the **string**, but will be read by the next *read*). If you do not want to have the **string** stopped by the end of the line (but only by the *term*), you may call the *on line end* procedure (7.4.4.4) so as to call *newline* automatically. If the current position is already at the end of the line (or if the line is empty), *newline* is not called – you just get an empty **string**. On the other hand, *newpage* will be called if you are off the end of the page.

[*m* : *n*] **chars** (i.e. a multiple with fixed bounds), **bytes** and **bits**:

The exact number of **chars** needed (i.e. *n–m+1* or *bytes width* or *bits width*)

is read, *newline* and *newpage* being called as needed. In the case of **bits**, spaces are skipped (as with **bools**).

If an item refers to a multiple value (structure), then its component elements (fields) are sought in turn according to the above rules. This is discussed more fully in 7.5.1 (7.4.1) under the heading of "straightening". The number of elements expected in a multiple value (other than a **string**) is the number contained in the existing multiple referred to by the item.

Vertical readers, please turn to 8.1.

7.2. Files

7.2.1. Channels, books and files

Your particular-program communicates with its environment via facilities termed "**channels**" [R 10.3.1.2]. A **channel** may be anything from a keyboard to a wind tunnel, with all the usual peripherals (tape, cards, magnetic tape, discs) coming in between. In a large operating system, it will most likely turn out to be a file in its filestore. We distinguish between "character transput" (in which the external representation of the data is potentially readable) and "binary transput" (in which it is not, and which we shall not consider until 7.7).

Confining ourselves, then, to character transput (although some **channels** may be able to accommodate both varieties), it is convenient to imagine that at the other end of the **channel** is a "book" (or maybe several books). Some **channels** permit the program to read the book, and some to write in it. Some very accommodating ones will permit both, and may even allow you to browse through the pages in any order. A program connected to several paper tape readers would be reading several such books through one **channel** (a **channel** is thus a type of device, rather than an individual piece of hardware). This **channel** would presumably permit reading (*get possible*), forbid writing (¬*put possible*), would insist that the book be read in strict sequence (¬*reset possible* and ¬*set possible*) but might conceivably agree to provide data in binary (*bin possible*).

A book has pages, lines and characters, the maximum number of each of which may be limited by the **channel**, although the actual book may be smaller. If a line is able to accommodate n characters, then positions within the line are specified by one of $n+1$ character numbers (the extra one

corresponding to an overflow position at the end). If the line is empty (i.e. it contains zero characters), then its character number is always *1* and it is always in the overflowed state. Similarly, if a page (a book) is able to accomodate *m* lines (*m* pages), then positions within the page (the book) are specified by one of *m+1* line numbers (page numbers). If the page (the book) is empty, then its line number (its page number) is always *1*. The "current position" is a triple (*page number, line number, char number*) specifying the position of the character to be read or written to next (normally the position just after the character read or written last). New readers are advised to start at (*1, 1, 1*). If, by some mischance, you find your current position to be at the overflow position of a line, or of a page, or of the book, then you have overflowed the "physical" book. If the book has been written, but not right up the its end, then the "logical end of file" is the position (page, line and character number) just after the last character written. New writers are recommended to start on an empty book with its logical end of file at (*1, 1, 1*). If by some further mischance you contrive to get your current position beyond your logical end of file, then you have overflowed the "logical" book.

A book has a title, its "identification", which you may use to ensure that you get the right one. Sometimes, you may be allowed to change the identification of the book (*reidf possible*). For your convenience when referring to the book from within your program, we provide an identifier for it, and with this we associate a record containing useful information (as detailed in 7.4.2). This record is of a special mode called **file** (in actual fact a **file** is a particular form of structure (1.4), and there is no reason why several **file**s should not refer to one book, nor why one **file** should not be assigned to another). **file**s may be declared thus:

(E1) **file** *my input, my output;*

The process of causing a **file** to refer to a book (via some specified **channel**) is known as "opening" the **file** (see 7.2.3 below). Initially, every particular program is provided with one book to be read, one to be written, and one to browse in. These are already opened in the standard-prelude [R 10.5.1.c] (and are closed in the standard-postlude), and are referred to by **file** variables called:

stand in *stand out* *stand back*	the books being linked via **channel**s called	*stand in channel* *stand out channel* *stand back channel*

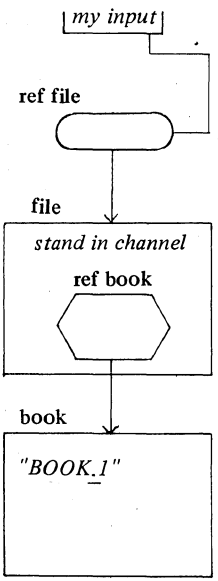
The properties of all books linked via these standard channels are given by

	stand in channel	*stand out channel*	*stand back channel*
set possible			true
reset possible			true
get possible	true		true
put possible		true	true
bin possible			true
reidf possible			
estab possible			
compressible			

You may open further **files** on these and other **channels**:

(E2) *open (my input, "BOOK_1", stand in channel);*
create (my output, special printer channel)

The results of the first of these can be represented thus:



The procedures *print* and *read* introduced in 7.1. automatically use the books referred to by *stand out* and *stand in.* Two further procedures *put* and *get* perform identical functions for other **files**. These must specify some **file**, and are such that:

*print (*XXXXX*)* is equivalent to *put (stand out,* XXXXX*)*
*read (*XXXXX*)* is equivalent to *get (stand in,* XXXXX*)*

7.2.2. Environment enquiries

You will see, therefore, that **channel**s and books have lots of useful properties. What we need are some more environment enquiries (see 6.2.1) to guide us [R 10.3.1.3]. Since some of these properties may be a function of the **channel** and some of the book and some (at the whim of the operating system) of both, or neither, these environment enquiries mostly take the form of procedures with **ref file** parameters. They are therefore only meaningful when a book has been opened on the file.

identifier of **proc** or constant	mode	value
set possible	**proc(ref file)bool**	**true** if random access is permitted
reset possible	**proc(ref file)bool**	**true** if the current position can be reset to *(1,1,1)* (e.g. rewind on magnetic tape)
get possible	**proc(ref file)bool**	**true** if input is possible
put possible	**proc(ref file)bool**	**true** if output is possible
bin possible	**proc(ref file)bool**	**true** if binary transput is possible
reidf possible	**proc(ref file)bool**	**true** if the **string** which identifies the book can be changed by *reidf* (7.2.3)
compressible	**proc(ref file)bool**	**true** if the line length can be varied on output
chan	**proc(ref file)channel**	the **channel** on which the **file** has been opened
estab possible	**proc(channel)bool**	**true** if new books may be established (7.2.3) on the **channel**
stand in channel	**channel**	the **channel** on which the **file** *stand in* is opened
stand out channel	**channel**	the **channel** on which the **file** *stand out* is opened
stand back channel	**channel**	the **channel** on which the **file** *stand back* is opened

Some further environment enquiries will be given in 7.5.3.

Observe that **channel** is actually a new mode. Implementations will doubtless ascribe to suitable identifiers in their library-preludes (1.1) extra **channel**s on the lines of *stand in channel* etc. (but you cannot create new **channel** values for yourself).

7.2.3. Procedures for opening and closing [R 10.3.1.4]

identifier of **proc**	mode	
open	**proc(ref file** *file*, **string** *idf*, **channel** *ch*)**int**	Attach *file* to an existing book with identification *idf* through **channel** *ch*. The book will already contain writing up to some logical end of file, and it will have some number of pages, lines and characters consistent with the maxima for *ch*.
establish	**proc(ref file** *file*, **string** *idf*, **channel** *ch*, **int** *mp*, *ml*, *mc*) **int**	Create a new, empty book with identification *idf* and with *mp* pages each of *ml* lines of *mc* characters. Attach *file* to this book through **channel** *ch* (with which *mp*, *ml* and *mc* must be consistent).
create	**proc(ref file** *file*, **channel** *ch*)**int**	Create a new, empty book with undefined identification, and with the maximum number of pages, lines and characters permitted by the **channel** *ch*. Attach *file* to this book through *ch*.
associate	**proc(ref file** *file*, **ref** [ ] [ ] [ ] **char** *sss*)**void**	The existing multiple value (of mode [ ] [ ] [ ] **char**), referred to by *sss* is attached to *file* in lieu of a book. By virtue of the rowing coercion (5.5.0 and 5.5.1.3), the actual-parameter supplied for *sss* may be of mode **ref** [ ] [ ] **char** (giving a "book" of only 1 page) or of mode **ref** [ ] **char** (giving a "book" of only 1 line).

Note that the **int** returned by *open*, *establish* and *create* is normally zero, but if the opening is not successful for some reason (e.g. the required book does not exist, or the operating system is unable to provide the required facility) some other integer may be returned, depending upon the implementation.

identifier of **proc**	mode	
scratch	**proc(ref file** *file*)**void**	detach the book (if any) attached to *file* and burn it
close	**proc(ref file** *file*)**void**	detach the book from *file* (but it may subsequently be opened again)
lock	**proc(ref file** *file*)**void**	detach the book from *file*. It may not be re-opened until it has been unlocked again by some action of the operating system
reidf	**proc(ref file** *file*, **string** *idf*)**void**	change the identification of the book to *idf* (*reidf* must be *possible*)

Note that, if two **files** are declared, and one is assigned to the other, then they are both attached to the same book. For the application of such assignations, see 7.4.2.

7.2.4. Position enquiries

[R 10.3.1.5]

identifier of **proc**	mode	
page number	**proc(ref file** *file)***int**	the current page number of the book
line number	**proc(ref file** *file)***int**	the current line number of the book
char number	**proc(ref file** *file)***int**	the current character number of the book. N.B. the page, line and character number between them define the character position about to be read from or written to.

7.2.5. Layout routines

[R 10.3.1.6]

identifier of **proc**	mode	
set	**proc(ref file** *file*, **int** *p, l, c)* **void**	Set the current position of the book referred to to *(p, l, c)*. Only meaningful if *set possible*. See 7.7.1 for applications.
set char number	**proc(ref file** *file*, **int** *c)***void**	Set the current position to character *c* within the current line.
reset	**proc(ref file** *file)***void**	Reset the current position to *(1, 1, 1)*. Only meaningful if *reset possible*. See 7.7.2 for further effects.
newpage	**proc(ref file** *file)***void**	see 7.1.1 and 7.1.2
newline	**proc(ref file** *file)***void**	see 7.1.1 and 7.1.2
space	**proc(ref file** *file)***void**	see 7.1.1 and 7.1.2
backspace	**proc(ref file** *file)***void**	see 7.1.1 and 7.1.2

Note that if the book is *compressible* (7.2.2), the effect of *newline* (*newpage*) during output is to terminate the current line (page) immediately, the length of the line (page) being determined by the number of characters (lines) already written. If the book is not *compressible*, the line (page) is filled out with spaces to the size specified when the book was *established* (7.2.3) (the sizes specified in *establish* are simply the maxima allowed in the *compressible* case).

It will be recalled that procedures of mode **proc(ref file)void** (or firmly coercible thereto) may appear as actual-parameters in calls of *get*, *read* (7.1.2), *put* and *print* (7.1.1). Of course, any procedure of this mode written by the user is acceptable in such positions. Of the procedures defined in the

standard-prelude, the following are the relevant ones:

backspace, space, newline, newpage, reset, scratch, close, lock.

Note that where such procedures are called from inside *read*, etc. they need no actual-parameter. In other places, the **file** must be specified.

Vertical readers, please turn to 8.2.

7.4. Structures and events

7.4.1. Straightening of structures

Given:

(E1) **struct** *(***int** *a,* **real** *b,* **compl** *c,* **char** *d, ...) s;*

then *print (s)* (or *put*, *get*, *read*, etc) is equivalent to

(E2) *print((a* **of** *s, b* **of** *s, c* **of** *s, d* **of** *s, ...))*

In other words, the fields of *s*, taken in order in which they were declared, are *print*ed (or *put*, or *got*, or *read*) in accordance with whatever rules are applicable to their modes. This is known as "straightening" [R 10.3.2.3]. If one of the fields is a further structure or a multiple value, then that field itself is also straightened, and so on. Note, however, that although the mode **compl** is a **struct**, it is specifically forbidden from being straightened into two **reals**.

Clearly, since the transput of names and routines and **formats** and the input of **unions** is forbidden (at least so far as the transput routines declared within the standard-prelude are concerned), it follows that these things cannot appear in **structs** that are to be transput.

For straightening of multiple values, see 7.5.1.

7.4.2. Files

A **file** is, in reality, a **struct** being declared in the standard-prelude [R 10.3.1.3.a] somewhat like this:

(E3)
```
mode file = struct(ref book ?book?,
                   secret ?conv?, ?term?,
                   proc(ref file)bool
                       ?logical file end?,
                       ?physical file end?,
                       ?page end?,
                       ?line end?,
                       ?format end?,
                       ?value error?,
                   proc(ref file, ref char)bool
                       ?char error?)
```

The queries around the field-selectors are intended to convey to you that these are not the true selectors of those fields. The true ones are secret, and so you have no way of making use of them. You can only alter them via procedures provided for the purpose or by assigning a complete new **file**, and it is therefore up to the implementor whether his **struct** is actually made up of the fields suggested in E3*.

Now, if you declare two **file**s, open a book on one, and then assign it to the other:

(E4)
```
file first, second;
open (first, "bookname", channel);
second := first;
```

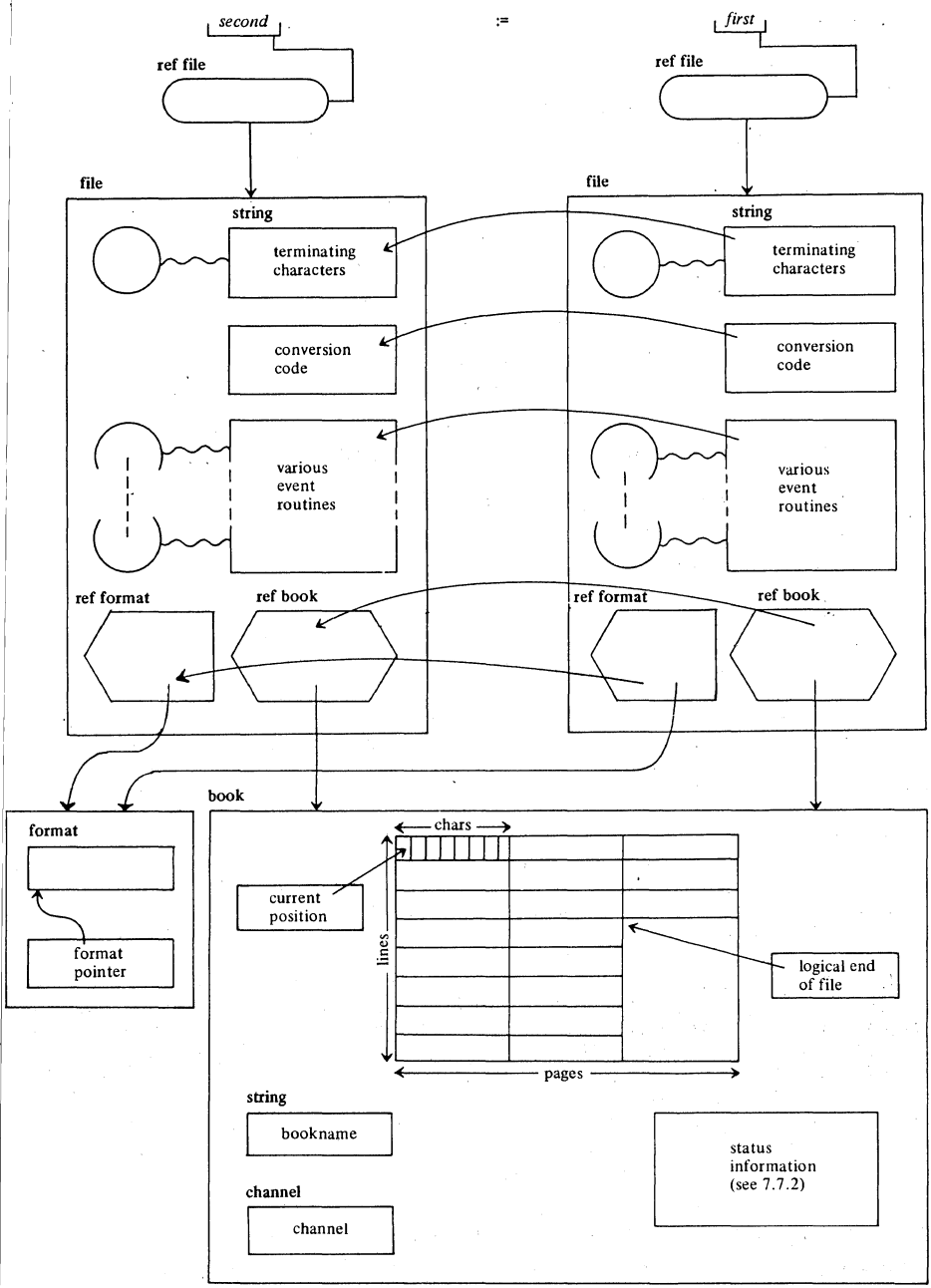
you arrive at the situation shown on the next page.

Both **file**s must inevitably refer to the same book, and there is no way in which you can change this. However, if you contrive to make one of the fields different in the two versions, then you may get different results when

* Pedantically speaking, this is not quite true. Your compiler ought not to complain upon seeing:

```
file f := (skip, skip, stand in channel, f, i, p, p, p, p, p,
           skip, "", skip, (ref file f) bool: false,
           (ref file f) bool: false, (ref file f) bool: false,
           (ref file f) bool: false, (ref file f) bool: false,
           (ref file f) bool: false, (ref file f, ref char a) bool: false)
```

but we would not like to take any bets on it.


second
:=
first
ref file
ref file
file
file
string
string
terminating characters
terminating characters
conversion code
conversion code
various event routines
various event routines
ref format
ref book
ref format
ref book
book
format
format pointer
chars
current position
lines
logical end of file
pages
string
bookname
channel
channel
status information (see 7.7.2)

you use them:

(E5)
make term (first, "A");
make term (second, "BC");
string *s;*
get (first, s);
comment will read in a **string** of characters from the current position up to the end of the current line, or up to an *"A"* (whichever occurs first) (see 7.1.1) **comment**
get (second, s);
comment will read in a **string** from the same book as before, starting from where the previous *get* left off (presumably starting with an *"A"* in this case) and reading up to the end of the line, or until either a *"B"* or a *"C"* is encountered **comment**

make term is the first of the procedures referred to above. Its use should be apparent from the above example. A complete list of these procedures is now given, and their use will be explained in the sections which follow

[R 10.3.1.3]

identifier of **proc**	mode	
make term	**proc(ref file** *f*, **string** *t)void**	assigns *t* to the *?term?* field of *f*
make conv	**proc(ref file** *f*, **secret** *c)void**	assigns *c* to the *?conv?* field of *f*
on logical file end	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?logical file end?* field of *f*
on physical file end	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?physical file end?* field of *f*
on page end	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?page end?* field of *f*
on line end	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?line end?* field of *f*
on format end	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?format end?* field of *f*
on value error	**proc(ref file** *f*, **proc(ref file)bool** *p)void**	assigns *p* to the *?value error?* field of *f*
on char error	**proc(ref file** *f*, **proc(ref file, ref char)bool** *p)void**	assigns *p* to the *?char error?* field of *f*

Procedures which alter the *?book?* field of a **file** are *open*, *close*, *establish*, *associate*, etc. (7.2.3). Note that *associate* causes the **file** to refer to a [ ] [ ] [ ] **char** instead of to a **book**.

7.4.3. Code conversion

All transput is really a matter of sending **chars** to or from a book. The function of the various transput procedures (*put*, *get*, etc.) is basically to convert the value on hand to or from **strings** of **chars**, and to transput the latter. You will doubtless have observed that internally we have been talking of "chars" – that is the internal objects of mode **char** which can be handled by an ALGOL 68 program. The things which we write in the book (i.e. the external representations) we have been talking of as "characters".

The relationship between these is determined by a conversion rule, and the conversion rule is kept in one of the secret fields of the **file**. A standard conversion rule is provided for each **channel**, and the intention is that the library-prelude of your implementation will provide additional ones to suit any special codes with which your installation may have to deal.

An environment enquiry [R 10.3.1.2.d] provides the standard rules, and the procedure *make conv* attaches them to the **file**.

identifier	mode	value
stand conv	**proc(channel** *chan)* **secret**	gives the standard conversion for *chan*

The mode **secret** is not really called **secret**, so you cannot do anything with it, except use it in *make conv*. When a **file** is *open*ed (or *establish*ed or *creat*ed) on a **channel**, it is set up with the appropriate *stand conv*. When a **file** is assigned, its *?conv?* goes with it.

(E6)
```
file first, second;
open (first, "bookname", stand in channel);
second := first;
make conv (second, special conv);
        ¢ supposing that special conv is available in the
          particular library-prelude ¢
get (first, s); ¢ reads a string according to the standard ¢
get (second, s); ¢ reads the next string from the same
                  book, according to special conv ¢
make conv (second, stand conv (stand in channel));
                ¢ restores the original rule ¢
get (second, s) ¢ now does the same as get (first, s)
                  would have done ¢
```

Each conversion rule specifies the transformation of each single **char** into a

single external character and vice versa (the mapping rule is not necessarily one to one in either direction).

7.4.4. Event routines

The remaining fields in **file** are event routines [R 10.3.1.3.cc]. They are provided to enable some user-defined action to take place when, for example, the end of a page is reached, without the user having to insert a test for this at the end of every transput call.

They all yield some **bool** value, and their default state, as left by *open*, *create*, *establish* or *associate*, is to yield the value **false**. If you write some routine of your own, and associate it with your **file** by means of one of the *on* procedures given above (7.4.2), then you may do what you like inside it, but there are three ways in which you may finish it:

1) yield **false**. In this case you are asking the transput routine which called you in to continue by taking its default action (which in some cases is left undefined by the Report, but which should then be some sensible system action).
2) yield **true**. In this case the calling routine will presume that you have corrected the situation to your satisfaction, and it will continue with next business.
3) jump right out of (i.e. terminate) your routine. In this case, the calling routine is terminated also. However, you must be sure that the label to which you jump is in the same reach as that in which your routine was declared, or in a surrounding range (**else** you will be in identification trouble (see 3.2.3)).

In cases 1) and 2), you may alter any of the values associated with the book (e.g. the current position – by *newline*, *reset*, etc. – or the **format**, or the contents) or with the **file** (e.g. by *close*ing and re-*open*ing it with a different book, or by providing a different terminating **string**, conversion rule or event routines).

Beware of associating an event routine with a **file** if its scope (4.2.3) is newer than that of the **file**. If necessary, you must declare a copy of the **file** with the scope of the proposed routine, so that the original **file** can continue to use its original routine outside this scope. A good example of this technique can be found in the Report at 10.3.1.3.cc.

We shall now consider the various routines and their uses. They all cause an event routine to be associated with the appropriate secret field of the **file**. The first four of them are concerned with exceeding the logical or physical limits of the book (7.2.1). To do this is in itself no crime. You are quite

entitled to have overflowed these limits by one character, one line or one page as the case may be. It is only when you are in such an overflowed position and you try to make matters worse by attempting transput (*put*, *get*, etc.) or layout (*space*, *newline*, etc.) that an event routine will be called [R 10.3.1.6.dd].

7.4.4.1. On logical file end

The associated event routine can be called by the input routines (*get*, *getf*, *get bin*, etc.) and by the layout routines *space*, *set char number*, *newline* and *newpage* when called in conjunction with input operations. It can also be called by *set*. Input or layout continues right up to and including the last character present in the book before the logical end of the file. If a further character is now demanded (or one of the layout routines is called), then the event is called. If this returns **false** (or if no such routine is provided) then the further elaboration is undefined (presumably the implementation halts the program with suitable diagnostics). If it returns **true**, then a further attempt is made to input the character or perform the layout.

The most likely action of the user's routine here is to recognise that his input data is ended, and to take steps to commence the next phase of his program.

7.4.4.2. On physical file end

The associated event routine can be called by the output routines (*put*, *putf*, *put bin*, etc.) and by the layout routines *space*, *set char number*, *newline* and *newpage* when called in conjunction with output operations. Output or layout continues until the book has overflowed (i.e. until the current position has gone beyond the last page that is physically available). If further output is now attempted (or one of those layout routines is called), then the event is called. The action taken is similar to the previous case – if **false** is returned the further elaboration is undefined, if **true** the output or layout is attempted again.

This event might be called, for example, if you had filled up a reel of magnetic tape, in which case it would be appropriate for your routine to *close* it and *open* another one.

7.4.4.3. On page end

The associated event routine can be called by all the transput routines (i.e. by both *put* and *get*, etc.) and by the layout routines *space*, *set char number*

and *newline*. Transput or layout continues until the current page has overflowed (i.e. until the current position has gone beyond the last line physically available in the page). If further transput is now attempted (or one of those layout routines is called), then the event is called. If this returns **false** (or if no such routine is provided), then the default action is to call *newpage* (which should remedy the situation). If it returns **true**, then the transput or layout is attempted again.

The user may of course call *newpage* himself and return **true**, but he then also has the opportunity, for example, of outputting some heading and page number on the new page.

7.4.4.4. On line end

This is exactly like *on page end*, except that the event happens when the current line has overflowed and the default action (except in formatted transput and when getting **strings**) is to call *newline* (which should remedy the situation).

The user may, again, call *newline* himself and return **true**, perhaps first outputting some line number.

It is to be noted that the default actions or user routines invoked by the above events may provoke other events. For example, the default call of *newpage* in *on page end* will normally remedy the situation but, in the exceptional case that this is the last page in the book, or before the logical end, the *physical* (or *logical*) *file end* event will then be invoked. In the extreme case in which the user when invited to mend the situation fails to do so, but erroneously claims (by returning **true**) that all is now well, the same event will be invoked again, and so on indefinitely.

7.4.4.5. On format end

The associated event routine is called by the formatted transput routines (*putf*, *getf*, etc.) when the **format** is exhausted. It may yield **false** whereupon the previous **format** associated with the book is repeated, otherwise it must provide a fresh **format** and yield **true**.

7.4.4.6. On value error

The associated event routine is called by the formatted transput routines (7.6.3) when the (internal) value on hand is incompatible with the current picture [R 10.3.4.1.1.hh, ii]. For example, on output the picture may provide too few digits, on input the value yielded may be too large to store

(e.g. > *max int*) or the expected literal in a choice may not be found, and in either case the mode of the picture may be wrong. If **true** is yielded, the offending value and picture are skipped; otherwise the result is undefined, except that with *putf* the value is first output with *put*.

This event is also invoked during formatless input (*get*) when the value yielded is too large to store.

7.4.4.7. On char error

The associated event routine is called by the input routines (*get* and *getf*) when the converted **char** read from the book does not agree with the sort of value expected (e.g. a letter is found when a number has been called for) or when the character in the book cannot be converted to any **char** (e.g. a parity error). With the call is provided the name of a **char** which it is proposed to substitute for the offender in order that input may continue. If **false** is yielded, then the implementation will take its own action (e.g. diagnostic message) after which the substitution may duly be made (although the user's routine may nevertheless have taken some note of the error). Alternatively, the user's routine may yield **true**, after possibly having assigned some alternative **char** to be used in place of the suggested one.

The suggestions that will be made in various circumstances are as follows:

Expected	suggestion
a number (unformatted)	*0*
a digit or supressed zero (formatted)	*0*
a sign (formatted)	+
a decimal point (formatted)	.
10 (formatted)	10
⊥ (for **compl**) (formatted or unformatted)	⊥
a **bool** value (formatted or unformatted	*F* (but see 7.5.3)
an expected **char** of an insertion (formatted)	that **char**

If the user, in this routine, wants to examine the offending character, he has only to backspace the **file** and use *get*. If he decides the offending character should be a candidate for the next input operation, he has only to leave the **file** backspaced.

In the case of an unconvertible character, the suggestion will be "␣" (i.e. a space).

Vertical readers, please turn to 8.4.

7.5. Rows and strings

7.5.1. Straightening of multiple values

Given:

(E1) [*1:n, 1:4*] **int** *j2;*

then *print(j2)* (or *put*, *get*, *read*, etc.) is equivalent to:

(E2) *print((j2[1,1], j2[1,2], j2[1,3], j2[1,4],*
j2[2,1], j2[2,2], j2[2,3], j2[2,4],
j2[3,1], j2[3,2], j2[3,3], j2[3,4],
. . . . .))

In other words, the rows of *j2*, and within them the elements of each row, are *print*ed (or *put*, or *got*, or *read*) in accordance with whatever rules are applicable to their mode. This is known as "straightening" [R 10.3.2.3]. If one of the elements is a structure, then it itself is straightened in accordance with 7.4.1 and if it is itself a multiple, then it is straightened as above, and so on. Note, however, that a **string** or [ ] **char** is never straightened into its constituent **chars**.

7.5.2. Conversion procedures

Basically, non-binary transput consists of considering values of various modes and converting these values to or from **strings**. It is the **strings** which are transput across the **channel** to or from the book.

print and *put* provide certain fixed rules for converting **ints**, **reals**, **compls**, **bools**, etc. (7.1.1 and 7.1.2). These are quick and easy to use, but they may not always provide the layout you want, in which case you must do-it-yourself (or you might consider formatted transput (7.6)). You can always output a **string**, and so you use one of the following do-it-yourself procedures [R 10.3.2.1] to make your own **string**. In the following table, **number** stands for **real** or **int** or any **long** or **short** version thereof.

identifier of **proc**	mode	
whole	**proc(number** *v*, **int** *w)***string**	Converts *v* into a **string** of **abs** *w* **chars**. Leading zeroes are replaced by spaces. If *w* is positive, a "+" or "−" is always included in the **string**. If *w* is negative, the **string** is unsigned for positive *v*. If *w* is zero, the shortest possible **string** into which *v* can be converted is provided (including a "−" if *v* is negative, but never a "+"). If *v* cannot be converted within **abs** *w* **chars**, a **string** of *w* asterisks is returned (but see 7.5.3).
fixed	**proc(number** *v*, **int** *w*, *a)***string**	Converts *v* into a **string** of **abs** *w* **chars**, including sign, if any, and decimal point and with *a* digits after the decimal point. The cases when *w* is negative or zero are treated as in *whole*. If *v* cannot be converted within **abs** *w* **chars** (even after reducing *a* so as to provide more positions before the decimal point), then *w* asterisks are returned.
float	**proc(number** *v*, **int** *w*, *a*, *e)***string**	Converts *v* into a **string** of **abs** *w* **chars**, including sign, decimal point and "$_{10}$", and with *a* digits after the decimal point and **abs** *e* digits (including any sign) of exponent. If *w* is negative, a leading "+" is replaced by "$\underline{+}$" and, if *e* is negative, positive exponents are not signed.
char in string	**proc(char** *c*, **ref int** *i*, **string** *a)***bool*	Returns **true** if *c* is contained in *s*, in which case the index of its first occurrence in *s* is assigned to *i*.

Examples. Given:

(E3) *i := 1023; x := 999.888;*

print(whole(i, 7));	¢ ␣␣+1023 ¢
print(whole(i, −7));	¢ ␣␣␣1023 ¢
print(whole(i, 0));	¢ 1023 ¢
print(whole(i, 3));	¢ *** ¢
print(fixed(−x, 8, 3));	¢ -999.888 ¢
print(fixed(−x, −8, 2));	¢ -1000.00 ¢
print(float(x, −12, 5, 3));	¢ ␣9.99888$_{10}$␣+2 ¢

7.5.3. Conversion environment enquiries

print and *put* always output the exact number of digits necessary to represent the largest possible magnitude of the value being output. The user might wish to know what this number of digits is, either when planning the layout of his page, or when using formatted transput (7.6). Appropriate environment enquiries are therefore provided [R 10.3.2.1.m, n, o, 10.2.1.r, s, t].

identifier of constant	mode	value
int width	**int**	the number of decimal digits required to represent *max int* (6.2.1) – not including sign
real width	**int**	the number of decimal digits required to represent a mantissa, such that *small real* (6.2.1) is not neglected in comparison with *1* – not including sign
exp width	**int**	the number of decimal digits required to represent a decimal exponent, such that *max real* (6.2.1) can be correctly represented – not including sign
error character	**char**	the **char** used to represent unconvertible values in *whole*, etc. (7.5.2) – in this book we use *"**"
flip	**char**	the **char** used to represent **true** during transput – in this book we use *"T"*
flop	**char**	the **char** used to represent **false** during transput – in this book we use *"F"*

See 6.7.1. for long(s) versions of these.

Note that the **char**s used for *error character*, *flip* and *flop* are to be chosen by the implementer – the choice of "*", "*T*" and "*F*" is merely the convention adopted for this book.

Vertical readers, please turn to 8.5.

7.6. Formatted transput

In formatted transput, the information about the values to be transput is presented separately from the information about how they are to be laid out. For example, given, in the book to be read, characters to yield the **string**:

*"+123␣456␣789/A47/999.888*6"*

we could read in the **int**s *123456789* and *47* and the **real** *999888000* by writing:

(E1)
```
read f( (
    $ + 3d x 3d x 3d "/", x 2d "/", 3d . 3d se "*" d $,
    i, j, x) )
```

Here, *i, j, x* is a data list of the names to which the values input are to be assigned. In this case they are two **ref int**s and a **ref real**, but they could have been of any of the modes acceptable to *read* (7.1.2) including references to multiples and structures which would require straightening (7.4.1 and 7.5.1). The only things not permitted here are **proc/ref file/void**s such as *newline*, *space*, etc. because these are concerned with the layout rather than the values.

The layout is controlled by the piece between the two "$"s, which is known as a 'format-text'. This is made up of various items known as 'frames', 'alignments' and 'literals', and the meaning of each item in this example is as follows:

item	name	effect
$		to introduce the format-text
+	sign frame	expect a "+" or a "−"
3d	digit frame	read *3* digits
x	alignment	skip one character
3d	digit frame	read *3* digits
x	alignment	skip one character
3d	digit frame	read *3* digits
"/"	literal	the next character must be a "/"
,		This is the end of the 1st "picture". The sign and the *9* digits that have been read are to be converted and assigned to the 1st value, which is *i*.
x	alignment	skip one character
2d	digit frame	read *2* digits
"/"	literal	the next character must be a "/"
,		This is the end of the 2nd picture. The *2* digits that have been read since the last picture are to be converted and assigned to the second value, which is *j*.
3d	digit frame	read *3* digits
.	point frame	expect a "."
3d	digit frame	read *3* digits

se	exponent frame	because of the *s* (for suppressed), no character is read for this frame, but the next digit frame will be interpreted as the start of a decimal exponent
"∗"	literal	the next character must be an ∗
d	digit frame	read *1* digit
$		End of the format-text and of the 3rd picture. The characters read (or implied) by its various frames are to be converted to **real** and assigned to the 3rd value, which is *x*.

Thus every character position of the input line is accounted for, and each must contain exactly what the format-text says it should. Formatted input is therefore very suitable for punched card input, where fixed layouts are customary, but less so for paper tape where the free layouts accepted by the unformatted procedures will often be more appropriate.

For output, however, the formatted procedures will always give more control over what is printed, chiefly by their ability to include fixed information (i.e. literals) anywhere amongst the values that are being printed. For example, to print the same line that we read in in E1, we could write:

```
(E2)    printf( (
            $ + 3d x 3d x 3d "|", "A" 2d "|", 3d . 3d se "∗" d $ ,
            123456789, 47, 999888000) )
```

You will see that the format-text here is almost exactly the same as before, the only difference being that here we specify the *"A"* that is to be printed, whereas on input we were prepared to pass over any character that might have been present. On the other hand, the alignment *x* is quite sufficient to ensure that a space will be output, unless some other piece of transput has tried to put some other character there. In this respect, the *x* behaves just like a call on *space* (7.1.1).

7.6.1. Format texts

A format-text consists of a list of 'pictures' separed by commas, the whole being enclosed between "$"s [R 10.3.4.1.1]. Each picture is obeyed in turn, and if it contains any frames it is matched up against the next value that is to be transput (otherwise its insertions are performed and the next picture is taken).

Within each picture there may be found 'insertions' (which can be further subdivided into 'literals' and 'alignments'), and 'frames'. Insertions, or sequences of insertions, may be put at the beginning or end of the picture, or in between any two frames.

7.6.1.1. Literals

A literal [R 10.3.4.1.1.i] consists simply of a **string** denotation (5.5.1.1). On output, when this point in the format-text is reached, the **string** denotation is printed. On input, it is "expected" [R 10.3.4.1.1.ll] ; i.e. the characters read from the book at this point must match the literal – if they do not, then the *char error* event is invoked (7.4.4.7).

The **string** denotation of a literal is actually preceded by a 'replicator'. These will be described more fully in 7.6.1.4. It will suffice for the moment that a replicator can consist of either "empty" or of an **int** denotation (5.1.1.1), and that we shall indicate the possible presence of one in what follows by an "R".

(E3) *printf(($ "START" 7"_." 3d $, i))*

There are two literals in the one and only picture in the format-text in this example. The first has an empty replicator, implying that the **string** *"START"* is to be printed only once. The replicator in the second shows that *"_."* is to be printed 7 times, so that the characters written to the book should look like this:

```
START.......987
```

Note that if two literals occur in succession, then the second one must have a non-empty replicator, for otherwise:

(E4) *"SMITH""JONES"*

would be ambiguous. In fact, E4 is a single literal which would be printed out (see 5.5.1.1) as:

```
SMITH"JONES
```

7.6.1.2. Alignments

Alignments [R 10.3.4.1.1.e] do not write any characters to the book. Their purpose is to move the current position (7.2.1) to some different page, line or char number, in a similar manner to the procedures *newpage*, *newline*, *space* and *backspace* (7.1.1 and 7.1.2) used in formatless transput. The following are the alignments permitted [R 10.3.4.1.1.ff] :

alignment	effect
R*x*	call *space* the number of times specified by the replicator R. I.e. skip over R characters.
R*q*	write (or expect) R space characters
R*y*	call *backspace* R times
R*l*	call *newline* R times
R*p*	call *newpage* R times
R*k*	call *set char number* (7.2.5) with R as its second parameter

(E5) *printf((*
$ l "ABCD" 4x 4a, 5k 4a $,
"IJKL", "EFGH"))

(in which *4a* is a character frame) will therefore cause to be written, on a new line in the book:

ABCDEFGHIJKL

7.6.1.3. Frames

frame type	syntax	effect on input	effect on output
digit frame	R*d* R*sd*	expect R digits	print R digits
sign frame	+ − R*z*+ R*z*−	expect "+" or "−" expect "." or "−" pass over up to R spaces (say *n*), and then expect "+", or "−" or "." as above, followed by R−*n* digits	print "+" or "−" print "." or "−" replace up to R leading zeroes (say *n*) by spaces, and then print "+", "−" or "." as above, followed by R−*n* digits
zero frame	R*z* R*sz*	expect R digits with leading zeroes replaced by spaces	print R digits with leading zeroes replaced by spaces
point frame	. *s.*	expect a decimal point	print a decimal point
exponent frame	*e* *se*	expect "$_{10}$" "\" or "*e*"	print "$_{10}$"
complex frame	*i* *si*	expect "⊥" or "*i*"	print "*i*"

frame type	syntax	effect on input	effect on output
radix frame	*2r* *4r* *8r* *16r*	convert the digits read, using the specified radix, into a **bits** value	convert the **bits** being output using the specified radix, and print in accordance with the rest of the frames
character frame	R*a* R*sa*	expect R characters	print R characters
boolean frame	*b*	expect *"T"* or *"F"* (but see 7.5.3)	print *"T"* or *"F"*
general frame	*g* *g(w)* *g(w, a)* *g(w, a, e)* here, *w*, *a* and *e* stand for units yielding **int**	accept characters as unformatted input (*read* or *get*)	print as in unformatted output (*print* or *put*) *print(whole(v, w))* *print(fixed(v, w, a))* *print(float(v, w, a, e))* where *v* is the value to be output
format frame	*f(format)* here *(format)* stands for any ENCLOSED-clause (3.2.4) yielding a **format**	transput proceeds using the **format** yielded by *format*. Upon exhaustion of this, the transput reverts to the next item of the original **format**	

In all the frames listed above, if an *s* is present, then on input the expected characters are not read from the book, but input proceeds as though they had been (and in the case where digits were expected, zeroes are yielded). On output, the characters concerned are not written to the book, but are simply "thrown away".

Note that sign, point, exponent and complex frames serve a dual purpose. They indicate that a certain character is to be expected or printed (unless suppressed), and they also indicate to the conversion routines the significance of the adjacent digit frames. Here are some examples:

(E6)

```
x := 999888000;
printf( ( $ +d.5de2d    $, x)); ¢   +9.99888₁₀08      ¢
printf( ( $ –5zde–zd    $, x)); ¢    999888₁₀  3      ¢
printf( ( $ .6de+2d     $, x)); ¢   .999888₁₀+09      ¢
printf( ( $ .5de+2d     $, x)); ¢   .99989₁₀+09       ¢
printf( ( $ +11zd.      $, x)); ¢   +   999888000.    ¢
printf( ( $ 11z+ds.     $, x)); ¢      +999888000     ¢
printf( ( $ 9d.3d       $, x)); ¢  999888000.000      ¢
printf( ( $ 6d3z.       $, x)); ¢  999888             ¢
printf( ( $ 6d3sds.     $, x)); ¢  999888             ¢
```

As you will see, any reasonable combination of frames is permissible. Rather than try to list all the permitted cases, we shall instead point out certain compatibility restrictions [R 10.3.4.2 – 10.3.4.7] which arise with certain modes of the value being transput. Failure to observe them will result in invocation of the *value error* event (7.4.4.6).

1) A radix frame (*r*) may only be useful if the value is **int**.
2) Either a point frame (.) or an exponent frame (*e*) must be present if the value is **real**.
3) A complex frame (*i*) must be present if the value is **compl**, with either a point frame (.) or an exponent frame (*e*) somewhere on each side of it.
4) On output, there is no objection to having point, exponent and complex frames present when the value is **int**, nor to having a complex frame when the value is **real**, since the **int** or **real** can be widened. These cases are not acceptable on input, however.
5) Character frames (*a*) may only be used if the value is **char** or **string**.
6) A boolean frame (*b*) may only be used if the value is **bool**.
7) The order in which the various frames, if present, must appear in **int**, **real** or **compl** pictures is as follows:
   sign frame (+, –, *z*+, *z*–)
   zero frames (*z*) and digit frames (*d*)
   point frame (.)
   zero frames (*z*) and digit frames (*d*)
   exponent frame (*e*)
   sign frame (+, –, *z*+, *z*–)
   zero frames (*z*) and digit frames (*d*)
   complex frame (*i*), in which case all the preceding frames may occur again.
8) Character frames (*a*) must not be mixed with other types within one picture, but there may be several of them.
9) A boolean frame (*b*), a general frame (*g*) or a format frame (*f*) must be the one and only frame of its picture.

Here are some more examples, illustrating **compls**, **bools** and **strings** (single **chars** are treated exactly like **strings**):

(E7)
```
printf( ( $ 2z–d.2d i –d.2de–d $, 37.2 i –43.4) );
        ¢  37.201–4.34₁₀1  ¢
printf( ( $ b,b,b $, true, false, true) );
        ¢  TFT  ¢
printf( ( $ 4a x 4a $, "ABCDEFGH" ) );
        ¢  ABCD EFGH  ¢
make term (stand in, "F");
readf( ( $ g $, s) ) ¢ reads "ABCDE" from a book containing
        ¢  ABCDEFGH  ¢
```

In addition to the various frames introduced above, there are two further types, known as "choices" [R 10.3.4.8], which can be used when the value to be transput is **int** or **bool**:

(E8)
```
i := 4; j := 5;
printf( (
    $ c("SUN", "MON", "TUES", "WEDNES", "THUS", "FRI",
        "SATUR") "DAY" $,
    i) );
    ¢ prints  WEDNESDAY  ¢
printf( (
    $ b("LESS_THAN", "GREATER") $,
    i<j) )
    ¢ prints  LESS THAN  ¢
```

On output, one of these literals is selected from the list according to the value of the **int** or the **bool**. (Note that a sequence of literals, complete with replicators (7.6.1.1), could be used in place of each of the single literals shown in the examples.) On input, one of the literals listed is expected, and a value is assigned to the **int** or **bool** accordingly. If two or more of the literals match, the earliest one in the list is taken. The search is never carried beyond the end of the current line. If no literal matches, the *value error* event is invoked (7.4.4.6).

7.6.1.4. Replicators and collections

Two types of replicator have been introduced already (7.6.1.1). These consist of "empty" and of an **int** denotation. There exists a third type, known as a "dynamic" replicator [R 10.3.4.1.1.dd], which consists of an *n*

followed by an ENCLOSED **int** clause (3.2.4):

(E9)
```
proc digits in = ( int i) int: entier(ln(i)/ln(10)+1);
j := 0;
for i to 4
   do
   j timesab 10 plusab i;
   printf ( ( $ l n(digits in (j))d $, j) )
   od
```
¢ prints:
```
1
12
123
1234 ¢
```

If the expression of a dynamic replicator yields a negative value, then zero is assumed.

Replicators may also be used to cause a 'collection' of pictures within a format to be repeated. This is particularly useful when a multiple value of flexible size is to be transput:

(E10)
```
flex [1 : 0] struct (int i, char a) ic1 := ((1,"A"), (2,"B"), (3,"C"), (4,"D"));
file f : = stand out; ¢ f has the same scope as ic1 ¢
putf (f, ($ p n(upb ic1)(4z+d, 2x a l), "TOTAL=" 3z+d $,
        ic1, (int i := 0; for j to upb ic1
             do i +:= i of ic1 [j] od; i) ) )
```

¢ which will print out, on a new page:

```
  +1  A
  +2  B
  +3  C
  +4  D
TOTAL=   +10 ¢
```

Such replicated collections can, of course, be nested to any depth. Moreover the grouping of the pictures so defined need not correspond to any natural grouping in the values being transput:

(E11)
```
[1:4] int j1 := (999,999,999,999), k1 := (888,888,888,888);
printf ( (
      $ 3d, 3(3"A" 2(3d x) 3x), 3d $.
      j1, k1) )
```

¢ which would print:

```
999AAA999 999     AAA999 888     AAA888 888     888¢
```

In this example, *j1* and *k1* are first straightened (7.5.1). The changeover from the values arising from *j1* to those arising from *k1* actually takes place half way through the second repetition of the outermost collection – this is slightly odd, but perfectly permissible.

7.6.2. formats

We shall now introduce a new mode, known as **format**. As with other modes, you may declare **formats**, assign them, refer to them, construct multiples and structures and **unions** out of them, invent **procs** that deliver them, and have operators operating upon them. About the only thing you cannot do with a **format** is to transput it.

The value of a **format** is the internal object yielded by some format-text (7.6.1). Moreover, the value of any **format** at any stage in the elaboration of a program, unless it is undefined, must be traceable back to a format-text somewhere in that program.

(E12)
```
format f; ¢ f is therefore of mode ref format ¢
f := $ + n(int width)d $;
printf ( (f, 999) )
¢ which will print something like:
+00000999 ¢
```

Because a format-text can contain expressions which in turn may contain identifiers of limited reach, it should be pointed out that the scope of a format-text is determined by exactly the same rule as is applied to routines (see 4.2.3), so that the result of the elaboration of the following is not defined:

```
format f;
   begin
   int i := 4;
   f := $ n(i)d $
   end
printf ( (f, j) )
```

7.6.3. The formatted transput procedures

Associated with each **file** is a **format** value and a **format** pointer (which

keeps track of which picture is to be used next). As with *print* and *read* (7.1.1, 7.1.2), the parameter supplied to *printf* or *readf* is a data list. each element of which yields either a **format** value, or a value to be transput (but not a **proc/ref file/void** this time). Usually, the first element provides a **format** (it is then associated with the **file** *stand in* or *stand out* as the case may be) whose sequence of pictures (as expanded by performing the replication of any collections) is then matched against the sequence of values obtained by straightening the remaining elements. This was done in all the examples shown so far, but it need not necessarily be the case. Example E2 can thus be rewritten as

```
(E13)   printf( ( $ + 3d x 3d x 3d "|" $, 123456789,
                  $ "A" 2d "|" $, 47,
                  $ 3d . 3d se "*" d $, 999888000) )
```

If a new **format** is provided before the old one has been exhausted, the old one is lost (note, however, that if the new **format** is provided to a copy of a **file** obtained by assignation, the original **file** still retains the old **format**, as should be clear from the diagram in 7.4.2). If part of a **format** still remains unused when the end of the data list is reached, it remains associated with the **file** and will be used for the next data list to be transput via that **file**.

```
(E14)   printf ($ p 6( 8( 6(+d.6de+d 2x) 1) 2l) $);
              ¢ a data list of one element which actually performs no
                transput at all ¢
        proc real compute = c computes some result c;
        to some large number
        do printf(compute) od
```

This will print the results of the computation, 6 numbers to a line. 48 lines to the page, with 2 extra blank lines inserted after every 8 lines. After printing the results of 288 computations the **format** will be exhausted. Then the *format end* event will be invoked (7.4.4.5). In this case the user has provided no event routine and so the default action is taken, viz the same **format** is started again (and printing is continued on a fresh page).

The procedures *printf* and *readf* use the books and **formats** referred to by *stand out* and *stand in* (there is also a procedure *writef* which is identical to *printf*). Two further procedures *putf* and *getf* perform identical functions for other **files**, and are such that:

printf (XXXXX) is equivalent to *putf (stand out, XXXXX)*
readf (XXXXX) is equivalent to *getf (stand in, XXXXX)*

7.6.4. Events

The various procedures which you can associate with each **file** in order to trap various events during transput were described in 7.4.4. We shall now summarise the situations in which each of them can be called during formatted transput.

During both input and output:

1) The various frames, alignments and literals encountered in the **format** may cause the current position to overflow the physical book (7.2.1). If a character is read from or written to the book in this situation, the *line end*, *page end* or *physical file end* event is called as appropriate.
2) If the end of the **format** associated with the book is reached before the straightened data list of values being transput has been exhausted, then the *format end* event is called.
3) If the mode of the value being transput is incompatible with the sequence of frames which occurs in the current picture (the possible causes of this were listed in 7.6.1.3), then the *value error* event is called.

During input:

4) If an attempt is made to read beyond the (logical) end of the book (7.2.1), then the *logical file end* event is called.
5) If the value yielded by conversion of various digits etc, in accordance with the picture, is too great for values of that mode (e.g. an **int** greater than *max int* (6.2.1)), then the *value error* event is called.
6) If the character read from the book in accordance with one of the frames *d, z, +, −, . , e, i* or *b* is not expected, then the *char error* event is called.
7) If a character that has been expected by a literal is not found, then the *char error* event is called.
8) If none of the literals of a choice is found, then the *value error* event is called.

During output:

9) If a value to be output cannot be converted into the number of digits specified in the picture, then the *value error* event is called.
10) If a negative value is to be output, and the picture does not contain a sign frame, then the *value error* event is called.
11) If the **int** to be output by a choice is zero or negative or greater than the number of literals in the list, then the *value error* event is called.

Vertical readers, please turn to 8.6.

7.7. Binary transput

Binary output may be used where the sole purpose of the material produced is that it should subsequently be read (during the same or some other program) by binary input. Normally, the medium used will be magnetic tape, disc or drum, but paper tape or cards can be used if your implementation permits.

During binary transput, the medium is still divided into pages, lines and chars, but it is not defined how many chars are occupied by each object that is transput, and your implementation may provide some strange shape for the physical book (e.g. a magnetic drum might be regarded as a continuum of chars, all on the one and only line of the one and only page: on magnetic tape, a line might correspond to some block length, and on a disc a page might demarcate a region which could be accessed without head movement). Nevertheless, it will always be true that the current position will be defined by the triple *(page number, line number, char number)*, and that it can always be inspected and manipulated by means of the facilities provided (7.2.4 and 7.2.5). In particular, the procedure *reset* can be used (7.2.5) if *reset* is *possible* (7.2.2) (as on tapes, discs and drums) and *set* can be used if *set* is *possible* (as on discs and drums). All the binary transput routines which are about to be described start from the current position. As line and page boundaries are passed, *line end* and *page end* events are called, but the default action is to call *newline* or *newpage*, so that the routines proceed automatically to cover as many chars, lines and pages as the values being transput may demand.

7.7.1. Binary transput procedures

In all of these procedures, the values are first straightened (7.4.1 and 7.5.1), and the straightened values are transput [R 10.3.6]. Thus all information as to how the original values were divided into structures and rows of multiple values is lost, as are the values of any bounds. Nevertheless, if the values output in this way are subsequently read back into a set of structures and multiples identical to that from which they originally came, then the new set will be an exact copy of the old.

(E1)
```
struct(int a, b)s1, s2, [1:4]int i1, i2;
s1 := (1,2); i1 := (3,4,5,6);
reset(stand back);
put bin(stand back, (s1, i1));
reset(stand back);
get bin(stand back, (s2, i2));
    comment we might just as well have said
                s2 := s1; i2 := i1
                However, if we want to mix things up: comment
reset(stand back);
get bin(stand back, (i2, s2))
    ¢ whereupon i2 has the value (1,2,3,4) and s2
      has the value (5,6) ¢
```

identifier of **proc**	mode	
put bin	**proc(ref file** *f*, [ ] **outtype** *x)*	*x* is straightened and the resultant values are output to the book referred to by *f*, starting at the current position (which is suitably advanced)
write bin	**proc(**[ ]**outtype** *x)*	equivalent to *put bin(stand back, x)*
get bin	**proc(ref file** *f*, [ ]**intype** *x)*	*x* is straightened to yield the names to which the values in the book referred to by *f*, starting at the current position (which is suitably advanced), are read
read bin	**proc(**[ ]**intype** *x)*	equivalent to *get bin(stand back, x)*

In this table, the modes **outtype** and **intype** (these are not their real names) indicate the modes acceptable to *print* (7.1.1) and *read* (7.1.2), with the exception of **proc(ref file)void**.

7.7.2. Some restrictions

The environment enquiries listed in 7.2.2 show what can and cannot be done with the various **channels** provided in an implementation. You can only do binary transput if *bin possible*. However, this is not to say that non-binary (i.e. character) transput is forbidden thereby. You are perfectly entitled to *put* and *get* to and from your disc, provided only that you are prepared to tolerate any strange page and line sizes that it may have.

There is an important distinction between **channels** with *set possible* (i.e. drums and discs) and the rest [R 10.3.1.4.j–m]. With *set possible*, you can

roam around your book writing characters here, reading them back there, and doing binary transput in between. It is your entire responsibility to keep track of the *(page number, line number, char number)* where everything has been put, and of course if you try to read back as characters that which has been written in binary, or vice versa, then the result will be quite undefined.

If, on the other hand, *set* is not *possible*, then things are different. Whenever you start off at *(1,1,1)* (after a *reset*, for example, assuming *reset possible*), you have the choice of reading or writing in characters or binary (assuming a suitable combination of *put*, *get* and *bin possible* on the channel). Thus you have 4 possibilities:

read binary – you must carry on reading in binary.
write binary – you must carry on writing in binary.
read characters – you may continue reading characters up to some point, and then change to writing characters. But you must not read beyond the logical end of the file (7.2.1).
write characters – each time you write, the logical end of the file is set to the new current position, which effectively prevents you from doing anything other than to write characters again.

Thus, with *put*, *get* and *reset possible*, but not *set possible*, the normal behaviour expected with magnetic tape is obtained. If you disobey any of these rules, then the result is undefined. Note that, when *set* is *possible*, it is still impossible to *set* beyond the logical end. Any attempt to do so will merely set the current position to the logical end exactly, and then call the *logical file end* event.

The possible properties of the three standard channels were given in 7.2.1. Note that only the minimum requirements are given there, and individual implementations might, for example, allow *bin possible* on *stand in channel.*

Vertical readers, please turn to 8.7.

8. EXAMPLES

8.1. Simple examples

In this chapter, which forms the tail of the columns in our othogonal plan, we shall show you, column by column, what you can do with the facilities described so far.

However, the language available to us at the end of this first column is rather sparse. We have shown you only the crudest form of conditional-statement. **strings** have only been hinted at. You cannot declare **procs**, nor **refs**, nor even constants. Any substantial and worthwhile example within these limitations could never do justice to the expressive power of ALGOL 68, and we must therefore invite you to read further before encountering a real example in 8.2.

If shorter examples of these simple facilities are what you would like to see, then we must refer you back to the early parts of Chapter 0 (the Very Informal Introduction) where you will find many such.

Vertical readers, please turn to 1.2.

8.2. Procedure examples

8.2.1. Easter

The Gregorian Calendar, insofar as it determines upon which day each year shall start, is universally accepted throughout the world. It also fixes the date of Easter as being the next Sunday after the Paschall Full Moon, which is intended to be the first full moon occurring on or after the Vernal Equinox (March 21st). The rules given for computing this are not so widely accepted. For example, the Jewish Passover and the Orthodox Easter are determined from different (and probably more accurate) calendars.

The defining document for these rules was written by one Clavius under a commission from Pope Gregory XIII [1]. Absolute accuracy was not a prime consideration. The Full Moon is considered to occur on the fourteenth day of the lunar month (which commences with the new moon). The rule was carefully devised so that the date predicted for its new moon always fell on, or one or two days after, the true mean new moon of the astronomers – but never before it. This was to ensure that never, under any circumstances, would the Christian Easter fall on the same day as the Jewish Passover

(notwithstanding which, this terrible circumstance does occasionally arise, as in 1903).

The following is a complete program for calculating the date of Easter according to the Gregorian rule. For further explanations, see [3] and [4].

```
begin
int year, date, moon, paschal, easter;
```

comment We shall reckon dates by the number of days since the start of the year.

Thus: **comment**

```
int march21st = 31 + 28 + 21;
```

comment The Gregorian calendar was introduced into various parts of the world at different dates. In Great Britain, the year was: **comment**

```
int gregory start = 1752 ¢ or whatever date you prefer ¢ ;
read ( year );
if    year < gregory start
then  print ( ( "The Gregorian calendar was not introduced until",
                gregory start, newline) )
else  int   century = year ÷ 100,
            leap = abs ( year mod 4 = 0 ∧ year mod 100 ≠ 0 ∨
            year mod 400 = 0 );
comment leap = 1 for a leap year, and 0 otherwise. comment
      print ( (newpage,
               year,
               if leap = 1 then " (Leap year)" else "" fi,
               newline) );
```

comment To calculate the day of the week corresponding to any date, we associate with each year a Dominical Letter, whose position in the alphabet gives the date of the first Sunday in January. **comment**

```
      int dominic = 7 – (year + year ÷ 4 – century + century ÷ 4
                         – 1 – leap) mod 7;
      print ( ("The Dominical Letter is ",
               case dominic in "A", "B", "C", "D", "E", "F", "G" esac,
               if      leap = 1
               then    case (dominic – 2) mod 7 + 1
                       in   "/A", "/B", "/C", "/D", "/E", "/F", "/G"
                       esac
               else    ""
               fi,
               newline) );
```

```
proc weekday = (int date) string:
    case (date – dominic) mod 7 + 1
    in   "Sunday", "Monday", "Tuesday", "Wednesday",
         "Thursday", "Friday", "Saturday"
    esac;
proc month = (ref int date) string:
```

comment This **proc** has a **ref int** parameter which it will alter to become the date within the month. **comment**

```
    if   date ⩽ 31                      then "January"
    elif (date := date–31) ⩽ 28 + leap  then "February"
    elif (date := date–28–leap) ⩽ 31    then "March"
    elif (date := date–31) ⩽ 30         then "April"
    elif (date := date–30) ⩽ 31         then "May"
    elif (date := date–31) ⩽ 30         then "June"
    elif (date := date–30) ⩽ 31         then "July"
    elif (date := date–31) ⩽ 31         then "August"
    elif (date := date–31) ⩽ 30         then "September"
    elif (date := date–30) ⩽ 31         then "October"
    elif (date := date–31) ⩽ 30         then "November"
    else date := date–30; "December"
    fi;
```

comment The moon revolves around the earth once every 29.530588 days. 235 such lunations last just $1\frac{1}{2}$ hours less than 19 Julian years. The calendar is therefore based on a "Metonic" cycle of 19 years, each year in a cycle being allotted a "Golden Number" in the range 1 to 19: **comment**

```
    int golden = year mod 19 + 1;
    print ( ("The Golden Number is", golden, newline) );
```

comment However, following this cycle indefinitely would introduce an error of approximately 0.43 days per century. There is therefore a correction which, for convenience, is only allowed to change at the end of a century: **comment**

```
    int lilius ¢ who is a not inherently meaningful identifier ¢
          = (century – century ÷ 4      ¢ for the leap years omitted
                                          at the start of some cen-
                                          turies ¢
            – (century–(century–17) ÷ 25) ÷ 3
                                        ¢ the 1½ hours error ¢
            –8 ) mod 30;
```

comment On the 1st of January of any year, the number of days since the last new moon is given by the "Epact": **comment**

```
        int epact = (11 × (golden–1)–lilius) mod 30;
        print (("The Epact is", epact, newline) );
```

comment If successive new moons were to occur every 30 days, then we should be able to associate with each date a unique epact, one less for each day modulo 30 (then that date would be a new moon in years with that epact). In fact, six times in the year (and once extra at the end of 19 years) we must have a lunation of only 29 days, whereupon the sequence of epacts slips back a day and some date will have two epacts listed against it. These dates have been carefully chosen (it is alleged) so as to minimise the deviation from the true moon. One of them occurs in February and so happenings in March occur exactly 59 days after those in January (or 60 in a leap year, since the intercalary day, if any, in February is automatically added to the lunation in which it occurs). Therefore, there is a new moon on: **comment**

```
        moon := 31–epact + 59 + leap;
        if    (paschal := moon + 13) <march21st + leap
        then
```

comment the fourteenth day of this moon falls before the Vernal Equinox and we want the next one. The next date with two epacts against it occurs in April, the critical epact being given by: **comment**

```
            int clavius = if golden > 11 then 26 else 25 fi;
            moon := moon + (epact ⩾ clavius | 30 | 29);
            paschal := moon + 13
        fi;
        print ( ("The Paschal Full Moon falls upon ",
                 weekday ( paschal ), space,
                 month ( date := paschal ), space) );
```

comment Note how we have to break off the *print* here and start another one, so that we can use the value of *date*, as calculated therein, in the next *print*. If it had all been done in one *print*, then we might have been using *date* and assigning to it at the same time (i.e. collaterally), and anything might have happened. **comment**

```
        print ( (date, newline) );
        print ( ("Easter day, being the next Sunday after the ",
                     "Paschal Full Moon, therefore falls upon ",
                 month ( date := easter := paschal + 7–(paschal – dominic)
                     mod 7) ) );
        print ( (date, newline) )
    fi
    end
```

The following are the references quoted in this section:
[1] Christophorus Clavius. Kalendarium Gregorianum Perpetuum. Cum Privilegio Summi Pontificis Et Aliorum Principum. Rome, Ex Officina Dominicae Basae. MDLXXXII. Cum Licentia Superiorum.
A companion volume was also prepared, and published in 1603:
[2] Christophorus Clavius. Romani Calendarii a Gregorio XIII. Pontifice Maximo restituti Explicatio.
[3] A. de Morgan. A Budget of Paradoxes. Longmans, Green & Co, 1872.
[4] Sir Harris Nicolas. The Chronology of History, containing Tables, Calculations & Statements, indispensable for ascertaining the dates of Historical Events, and of Public and Private Documents from the earliest periods to the present time. Longman, Brown, Green and Longman's, 1838.

Vertical readers, please turn to 1.3.

8.3. Examples of operators

8.3.1. Parallel plus

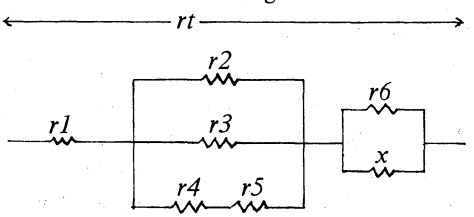
This example is intended to show how defining your own operators can lead to a considerable simplification of a program, at the same time making it more easy to follow.

It is well known that if two resistors A and B are placed in parallel, their combined resistance is given by:

$$\frac{1}{\frac{1}{A}+\frac{1}{B}}$$

We can now define the operator "parallel plus" (we shall represent it by **pap**) to perform this operation. **pap** is a well behaved operator, being both commutative and associative. For the sake of completeness we shall also define "parallel minus" (**pam**) and assigning versions (6.3) **paplusab** and **paminusab**.

Consider the following network:



It is required to find the value of *x* such that the resistance of the whole network shall be *rt*. Here is a program to do it:

```
begin
prio pap = 7, pam = 7, paplusab = 1, paminusab = 1;
op pap = (real a, b) real: a × b / (a + b);
op pam = (real a, b) real: a × b / (b–a);
op paplusab = (ref real a, real b) ref real: a := a × b / (a + b);
op paminusab = (ref real a, real b) ref real: a := a × b / (b–a);
real rt, r1, r2, r3, r4, r5, r6;
read ((rt, r1, r2, r3, r4, r5, r6));
print ((rt–(r1 + r2 pap r3 pap (r4+r5))) pam r6)
end
```

Electrical engineers will realise that all these operators ought also to be defined for **compl** operands (2.4.4) and for mixed **real** and **compl**.

Vertical readers, please turn to 1.4.

8.4. Two examples of library preludes

A library-prelude (see 1.1) is an expansion of the standard-prelude. It may contain further identity-, mode-, priority- and operation-declarations for use in particular applications. A library-prelude must be throughout consistent with the standard-prelude (i.e. indicators declared in it may not conflict with those declared in the standard-prelude).

In 8.4.1 we give a library-prelude for the basic operations on vectors in a Euclidean space E_3, the vectors being declared as

```
mode vec = struct (real xcoord, ycoord, zcoord);
```

In Section 8.5 we give a library-prelude for the basic operations on vectors in E_n (n arbitrary) which is more general, but also less simple than this one.

In 8.4.2 we give a library-prelude for the basic operations on rational numbers, declared as

```
mode rat = struct (int numerator, denominator);
```

The declarations in 8.4.1 and 8.4.2 (and also those in 8.5) are fully consistent with each other and may, therefore, be joined into one library-prelude without any precaution. They may be regarded as one of many possible expansions of the language. Their intention is to demonstrate the expressive power and efficient elegance of the language, and to suggest how

to do away in practice with dialects and specific languages for use in particular problems.

8.4.1. Operations on vectors in E_3

```
mode vec = struct ( real xcoord, ycoord, zcoord ) ;

(1)  prio parl    = 5 , perp = 5 , proj = 6, ⊥x = 7;
(2)  op x         = (vec u) real: xcoord of u ;
(3)  op y         = (vec u) real: ycoord of u ;
(4)  op z         = (vec u) real: zcoord of u ;
(5)  op +         = (vec u) vec: u ;
(6)  op –         = (vec u) vec: (–xu , –yu , –zu) ;
(7)  op +         = (vec u , v) vec: (xu + xv , yu + yv , zu + zv) ;
(8)  op –         = (vec u , v) vec: (xu – xv , yu – yv , zu – zv) ;
(9)  op x         = (real r , vec u) ¢ the product of a scalar and a vector ¢
                      vec: (r x xu , r x yu , r x zu) ;
(10) op x         = (vec u , real r) ¢ the product of a vector and a scalar ¢
                      vec: r x u ;
(11) op /         = (vec u , real r) ¢ the quotient of a vector and a scalar ¢
                      vec: (xu / r, yu / r, zu / r) ;
(12) op x         = (vec u , v) ¢ the innerproduct ¢ real:
                      xu x xv + yu x yv + zu x zv ;
(13) op ⊥x        = (vec u , v) ¢ the vectorproduct ¢ vec:
                      ( yu x zv – zu x yv ,
                        zu x xv – xu x zv ,
                        xu x yv – yu x xv ) ;
(14) op norm      = (vec u) real: sqrt ( u x u ) ;
(15) op e         = (vec u) ¢ the unit vector in the direction of u ¢ vec:
                      (eps u | skip | (1/norm u) x u) ;
(16) real eps     = c some small enough real number c ;
(17) op eps       = (real r) bool: abs r < eps ;
(18) op eps       = (vec u) bool: eps norm u ;
(19) op parl      = (vec u , v) ¢ parallel? ¢          bool: eps (eu ⊥x ev) ;
(20) op perp      = (vec u , v) ¢ perpendicular? ¢     bool: eps (eu x ev) ;
(21) op proj      = (vec u , v) ¢ the protection of u on a plane
                                 perpendicular to v ¢ vec:
                      (vec ew = e ( (v ⊥x u ) ⊥x v) ; (u x ew) x ew);
(22) proc angle   = (vec u , v) ¢ the angle between u and v ¢ real:
                      arccos (eu x ev) ;
```

```
(23) proc plane = (vec u , v , w ) ¢ in one plane ? ¢ bool:
                  eps ( eu x (ev ⊥x ew ) ) ;
```

8.4.1.1. Comments on the library-prelude 8.4.1

The given set of operations is confined to E_3-vectors over the field of real values; it is obvious that they can as easily be defined over the field of complex values and also over more particular fields (see, for example, the rationals as defined in 8.4.2). It is also obvious in what way more specific operations can be subjoined to those given in 8.4.1; the declarations (21), (22) and (23) are already of a somewhat specific nature.

The given set may be of use in a variety of applications in mathematics, physics, chemistry and astronomy, enabling the programmer to write transparent and straightforward algorithmic prose in his own professional jargon.

Further modes may be derived from the given **vec**, for instance:

```
mode event = struct ( real time , vec position ) ;
mode tens  = struct ( vec xlevel , ylevel , zlevel ) ;
```

The kinds of operators to be then declared for **event** and **tens** values, of course, depend entirely upon the specific applications. For an example of the use of the mode **event**, see 8.4.1.2.

The declarations (1) – (12) may speak for themselves.

In (13) we adopted for vector multiplication (sometimes termed the "outer product") the operator ⊥x. The formula u ⊥x v yields a **vec** perpendicular to both u and v and of length $|u||v|\sin(u, v)$.

The **norm** defined in (14) is the usual Euclidean norm. If the underlying field is that of the rational values (8.4.2) we may define:

```
op norm = ( vec u ) rat: max ( abs xu , abs yu , abs zu );
```

The environment enquiry *eps* in (16) serves as a criterion for "zeroness" and is used in (20), (21) and (23) to define parallelism, perpendicularity and "planeness". The value of *eps* may depend heavily upon the given input- and the required output-precision. If, for instance, **real** *eps* = *0.01*, then two vectors will be considered as "perpendicular" to each other as soon as their inner product is <0.01 (see also the example in 8.4.1.2).

In (21) the **vec** *ew* (which is local to the routine) is a unit vector perpendicular to v in the plane of v and u (we applied the vector product ⊥x twice); hence, *(u x ew) ¢ a* **real** ¢ x *ew* yields a **vec** in the plane perpendicular to v and in the plane of v and u.

The declarations (22) and (23) may speak for themselves.

8.4.1.2. An example of the use of vecs

The input starts with an integral number *n*, which fixes the number of **real** quadruples following. The first **real** in each quadruple is a point in time, the remaining three **reals** define a point in space (a **vec**). In the first line of the program below we define such a quadruple to be an **event**.

Let the input consist of some thousands of **events**, ordered in time; the time-coordinates are not necessarily equidistant. One may conceive the row of **events** to be the result of some smoothing process on a large set of measurements. The **events** may then describe, with sufficient accuracy to allow second order numerical differentiation, the orbit of a particle (be it a mass body or an electric charge) in a possibly complicated field of forces.

The program below surveys the motion and acceleration of the particle in its orbit. It includes a few features of the language which you may not have met yet (if you have been reading vertically). Nevertheless, we hope their meaning will be readily apparent to you (if not, please see 2.5 and 5.5.1.3 for multiple values and how to slice them, and 3.5.2 for the use of **for**).

```
(E1) begin mode event = struct (real time , vec pos ¢ ition ¢ ) ;
     proc deriv = ¢ yields the derivative of a vec as a function of time ¢
                  ( [ ] event e , int k ) event:
          ( event e1k = e[k-1], ek1 = e [k+1] ;
            real  dt  = time of ek1 - time of e1k ;
            ( time of e1k + dt/2 , (pos of ek1 - pos of e1k) / dt )
          ) ;
     int n ; read (n) ;
     [1:n] event orb; read (orb) ; ¢ see 7.1.2 ¢
     real init ¢ initial time ¢ = time of orb [1] ;
     print ( ("initial time =", init , newline) ) ;
     [2:n-1] event vel ¢ ocity ¢ ;
     for i from 2 to n-1 do vel [i] := deriv (orb , i) od ;
     vec newx = e pos of vel [2] ; ¢ unit vector tangent to the orbit at
                                     time init ¢
     vec newy = e ( pos of vel [3] proj newx ) ; ¢ newx-newy is the
                                                   tangent plane to the
                                                   orbit at time init ¢
     vec newz = newx ⊥x newy ; ¢ newz is the unit vector perpendicular
                                 to the unit vectors newx and newy ¢
```

```
proc new = ¢ yields the coordinates of an event relative to the initial
             time and the new axes newx-newy-newz ¢
             (event e) event:
             ( time of e – init , (pos of e x newx , pos of e x newy ,
               pos of e x newz) ) ;
proc pr = ¢ prints characteristic data of an event, preceded by a
          string and a newline ¢ ( string s , event e ) void:
    ( event newe = new(e) ;
      print (( newline , s , e pos of newe , norm pos of newe ,
      time of newe ))
    ) ;
for i from 3 to n–2
do  event veli = vel [i] , orbi = orb [i] ,
              acci ¢ acceleration ¢ = deriv ( vel , i ) ;
    vec v = pos of veli , a = pos of acci ;
    ¢ if certain situations occur, messages and data will be
      printed: ¢
    if   v perp newx   then pr ( "orbit in YZ" , orbi ) ;
                            pr ( "velocity = " , veli ) ;
    elif v perp newy   then pr ( "orbit in ZX" , orbi ) ;
                            pr ( "velocity = " , veli ) ;
    elif v perp newz   then pr ( "orbit in XY" , orbi ) ;
                            pr ( "velocity = " , veli )
                       else skip fi ;
    if   eps v         then pr ( "standstill" , orbi ) ;
                            pr ( "accelerat.=" , acci ) fi ;
    if   eps a         then pr ("zero force " , orbi ) ;
                            pr ( "velocity = " , veli ) fi ;
    if   a parl v      then pr ( "force//orbit" , orbi ) ;
                            pr ( "velocity = " , veli ) ;
                            pr ( "accelerat.=" , acci )
    elif a perp v      then pr ( "force⊥orbit" , orbi ) ;
                            pr ( "velocity = " , veli ) ;
                            pr ( "accelerat.=" , acci )
                       else skip fi
    ¢ if an appropriate device is available, one might conceive here
      the call of a procedure plotting the curve ¢
od
end
```

8.4.2. Operations on rational operands

```
    mode rat  = struct ( int numerator , denominator ) ;

(1) prio nn = 7, nd = 7, dd = 7;
(2) op n    = ( rat r ) int: numerator   of r ;
(3) op d    = ( rat r ) int: denominator of r ;
(4) rat o   = ( 0 , 1 ) ;
(5) rat l   = ( 1 , 1 ) ;
(6) proc errat = void: c some action interrupting or halting the elaboration
                       of the program signalizing that a result, required to
                       be rational, cannot be expressed as a value of the
                       mode rat c ;
(7) proc gcd = ( int n , d ) int:
                 if d = 0 then abs n else gcd ( d , n mod d ) fi ;
(8) proc long gcd = ( long int n , d ) long int:
                      if d = long 0 then abs n else long gcd ( d , n mod d )
                      fi ;
(9) op ↓ = ( int n , d ) rat:
             if   d = 0
             then errat ; skip
             else int k = gcd ( n , d ) ;
                  ( if sign n = sign d then abs n ÷ k
                    else – abs n ÷ k fi ,
                    abs d ÷ k )
             fi ;
(10) op ↓ = ( long int n , d ) rat:
             if   d = long 0
             then errat ; skip
             else long int k = long gcd ( n , d ) ;
                  long int nk = n ÷ k , dk = d ÷ k ;
                  if   abs nk ⩽ leng maxint
                       ∧ abs dk ⩽ leng maxint
                  then shorten nk ↓ shorten dk
                  else errat ; skip
                  fi
             fi ;
(11) op sign  = ( rat r ) int:  sign n r ;
(12) op whole = ( rat r ) bool: d r = 1 ;
(13) op entier = ( rat r ) int:  if n r ⩾ 0 then n r ÷ d r else n r ÷ d r – 1 fi ;
(14) op frac  = ( rat r ) rat:  r – entier r ;
```

```
(15) op round  = ( rat r ) int:   if frac r < 1 ↓ 2 then entier r else entier r + 1 fi ;
(16) op re     = ( rat r ) real:  n r / d r ;
(17) op nn     = ( rat p , q ) long int: leng n p × leng n q ;
(18) op nd     = ( rat p , q ) long int: leng n p × leng d q ;
(19) op dd     = ( rat p , q ) long int: leng d p × leng d q ;
(20) op <      = ( rat p , q ) bool: p nd q < q nd p ;
(21) op =      = ( rat p , q ) bool: p nd q = q nd p ;
(22) op >      = ( rat p , q ) bool: p nd q > q nd p ;
(23) op ≤      = ( rat p , q ) bool: ¬ ( p > q ) ;
(24) op ≠      = ( rat p , q ) bool: ¬ ( p = q ) ;
(25) op ≥      = ( rat p , q ) bool: ¬ ( p < q ) ;
(26) op +      = ( rat r ) rat: r ;
(27) op –      = ( rat r ) rat: –n r ↓ d r ;
(28) op +      = ( rat p , q ) rat:   ( p nd q + q nd p ) ↓ ( p dd q ) ;
(29) op +      = ( int n , rat r ) rat: ( leng n × leng d r + leng n r ) ↓
                                          leng d r ;
(30) op +      = ( rat r , int n ) rat: n + r ;
(31) op –      = ( rat p , q ) rat:   p + – q ;
(32) op –      = ( int n , rat r ) rat: n + – r ;
(33) op –      = ( rat r , int n ) rat: r + – n ;
(34) op ×      = ( rat p , q ) rat:   ( p nn q ) ↓ ( p dd q ) ;
(35) op ×      = ( int n , rat r ) rat: ( leng n × leng n r ) ↓ leng d r ;
(36) op ×      = ( rat r , int n ) rat: n × r ;
(37) op /      = ( rat p , q ) rat:   p × ( d q ↓ n q ) ;
(38) op /      = ( int n , rat r ) rat: n × ( d r ↓ n r ) ;
(39) op /      = ( rat r , int n ) rat: r × ( 1 ↓ n ) ;
(40) op ↑      = ( rat r , int n ) rat: ( n r ↑ n ) ↓ ( d r ↑ n ) ;
```

8.4.2.1. Comments on the library-prelude 8.4.2

Possibly more than may be necessary for vecs, the operations on **rats** should be "hand-coded" to take advantage of specific machine-features in double-precision integral arithmetic.

(1) The down-symbol ↓ is used to obtain a **rat** from two **ints** or from two **long ints**. It already has a priority of 8 (see 6.1.2).
(4–5) The rationals zero and one are ascribed to the identifiers *o* and *l*.
(7–8) The recursive declarations of *gcd* and of *long gcd* is not only the most natural algorithm, but most likely (in a good implementation) also the most efficient one.

(9–10) *777*↓ *1813* yields *(3,7)*
−777↓ *1813* yields *(−3,7)*
777↓ *−1813* yields *(−3,7)*
−777↓ *−1813* yields *(3,7)*
correspondingly for **long int** operands.

(8) and (10) All intermediate computations on the numerator and denominator will be performed in double-precision. If (being a vertical reader) you are not familiar with the mode **long int**, the denotation **long** *0* and the operators **leng** and **shorten**, please be assured that such double-precision is indeed achieved (or read 5.7.1.2 and 6.7). In (10), the fraction is then reduced (if possible) to the mode **rat**.

Notice that *1813 / 777* yields the **real** *2.3333333*
1813 ÷ 777 yields the **int** *2*
but *1813* ↓ *777* yields the **rat** *(7,3)*

(11) **sign** *(777* ↓ *1813) = 1* , **sign** *(−777* ↓ *1813) = −1* , **sign** *o = 0*
(12) **whole** *(777* ↓ *1813)* = **false** , **whole** *(37* ↓ *1)* = **true**
(13) **entier** *(1813* ↓ *777) = 2* , **entier** *(−1813* ↓ *777) = −3*
(14) **frac** *(1813* ↓ *777) = (1,3)*
(15) **round** *(1813* ↓ *777) = 2* , **round** *(2321* ↓ *777) = 3*
(16) **re** *(1813* ↓ *777) = 2.3333333*

(17–19) The operators **nn**, **nd**, **dd** are declared mainly to facilitate the notation of the remaining routines.

(20–40) These declarations may speak for themselves.

8.4.2.2. Some remarks on the use of **rat**ionals

The given set of operations on **rat**ional numbers may be of some importance in problems in which the (rational) coefficients of power series should be determined exactly. Such problems may arise when operations like addition, subtraction, multiplication, division and substitution of power series are relevant. A (truncated) power series (i.e. a polynominal) may, for instance be declared as a:

mode powser = flex [*1 : 0*] **rat** *;*

The power series for the exponential function then occurs as:

powser *exp = (1 , 1, (1,2) , (1,6) , (1,24) , (1,120) , (1,720) ,*
(1,5040) , (1,40320) , (1,362880) , **c** *etc* **c)** *;*

and the Bernouilli numbers (the Bernouilli polynomial) as:

```
powser bern = ( 1, (-1,2), (1,6), o , (-1,30), o , (1,42), o , (-1,30),
                o , (5,66), o , (-691,2730), o , (7,6), o ,
                (-3167,510), o , c etc c ) ;
```

For such **powsers** one may then define operators:

```
op + = (powser p , q) powser:   c routine for addition c ;
op - = (powser p , q) powser:   c routine for subtraction c ;
op x = (powser p , q) powser:   c routine for multiplication c ;
op / = (powser p , q) powser;   c routine for division c ;
```

(see also 8.5)

```
op ↓ = (powser p , q) powser:   c routine for the substitution of
                                  p in q c ;
```

The complex function defined by such **powsers** may be declared as:

```
proc fun = ( powser p , compl z ) compl:
             ( int m = lwb p, n = upb p ; compl value := re p [n] ;
               for i from n-1 by -1 to m
               do value := value x z + re p [i] od ;
               value);
```

Vertical readers, please turn to 1.5.

8.5. A library prelude for vector and matrix operations in E_n

For many applications in a variety of scientific disciplines you may want to write in your particular-program mode- and identity-declarations such as:

```
mode vector = [1:n] real ,
     matrix = [1:n,1:n] real ;
real   p , q , r ;
vector u , v , w ;
matrix a , b , c ;
```

and to apply operators yielding the sum, the difference, the (inner)product, the norm, etc. of such vectors and matrices, in order to be able to write straightforward expressions close to the well established mathematical notation, such as:

```
u := r × v ;
u := v + w ;
r := v × w ;
p := norm u ;
a := r × b ;
c := a + b ;
c := a × b ;
u := a × v ;
u := v × a ;
p := det a ;
v := inv a × u
```

Then you may, of course, also want to write composite formulae such as:

```
u := a × ( v / norm v ) ;
u := inv ( a × ( b−c ) ) × ( w−v )
```

You may even want to use vectors of unequal length and non-square matrices:

```
[1:n] real x , y ;      [1:m] real z ;
[1:m, 1:k] real m ;  [1:k, 1:n] real n ;
```

and formulae such as:

```
z := m × n × ( x + y )
```

The library-declarations listed in 8.5.1, 8.5.2 and 8.5.3 supply a basic and general set of such linear operations in E_n. The whole set is fully compatible with the library-declarations 8.4.1 and 8.4.2 and presupposes even the priority-declarations 8.4.1 (1) and the declarations for *eps* and **eps** in 8.4.1 (15) and (16). As was the case in the library-prelude for **vecs** (8.4.1) we confine ourselves to E_n-vectors and matrices over the field of real values; it may again be obvious that and how the whole set can be expanded over the field of complex values and even over more particular fields such as the rational values (8.4.2).

All our operation-declarations have formal-parameters of mode [ ] **real** or [,] **real**, even though this may lead to great inefficiency in some implementations due to the copying of multiple values when these operators are used in formulas (see 4.5). We make no apology for this. The operators reflect the user's requirement much more naturally in the form here given, and their yields (which are again multiple values) are in a form immediately suitable for use as operands in further formulas. As for the so-called inefficiency, this is simply a problem of implementation. Methods exist which avoid the copying entirely, except in those peculiar and rare cases where it is really needed.

(However, should you be faced with an old-fashioned inefficient implementation, then you will need to declare all these formal-parameters as **ref** [ ] **reals** or **ref** [,] **reals**. Moreover, the values returned will also have to be of these modes. Next you will have to worry about the scope (3.2.2) of these returned names, and you will then find that the variable-declarations which create them will have to have **heap** where we have written **loc** (2.7.3). Alternatively, shoot your implementor.)

In calculating the innerproduct of two vectors (line (10) of 8.5.1), we make use of double-precision arithmetic. If (being a vertical reader) you are not yet familiar with the **long** modes, please see the remarks about them in 8.4.2.1.

The intention of these declarations is (rather than to make a definite proposal for a particular library-prelude) to show how in a quite natural, even dogmatic, manner one can define a set of powerful operators which are, in their application, very close to the generally accepted conventions of mathematical notation.

A priori information about the multiples to which the operators are to be applied and considerations of required precision may influence the ultimate form of the routines ascribed to these operators. In particular the algorithms in 8.5.3 may, from several technical points of view, depend heavily upon a priori information about the condition of the matrices there we are faced with problems of numerical analysis rather than of programming.

In 8.5.1 we declare the operations on vectors, adopting a notation which is as close as possible to the notation of 8.4.1. In fact, if you redeclare in your particular-program:

mode vec = [*1*:*3*] **real**;

then the result yielded by the operators +, –, x, **norm**, **e**, **eps**, **parl** and **perp** will be the same by 8.5.1 as it otherwise would have been by 8.4.1.

In 8.5.2 we declare the operations on vectors and matrices, applying where possible the operators declared in 8.5.1. Consider, for example, in 8.5.2 (10) the statement:

for *k* **from** *p2* **to** *q2* **do** *ab*[, *k*] := *a* x *b*[, *k*] **od**

the occurrence of "x" identifies the operator declared in 8.5.2 (8) where in its turn the occurrence of "x" in the statement:

for *i* **from** *m1* **to** *n1* **do** *au*[*i*] := *a*[*i*,] x *u* **od**

identifies the operator declared in 8.5.1 (10) where in its turn the occurrence of "x" in the statement:

for *i* **from** *m* **to** *n* **do** *inpr* +:= **leng** *u*[*i*] × **leng** *v*[*i*] **od**

finally identifies the operator declared in the standard-prelude [R 10.2.3.4.l].

Observe that the operators +:=, −:=, ×:= and /:= in 8.5.1 and 8.5.2 may be considerably more efficient than +, −, × and /, because no intermediate results have to be stored anywhere; for which reason they are anyhow less space-consuming!

The library-declarations in 8.5.3 may be considered as examples of the use of the operators declared in 8.5.1 and 8.5.2. The procedure *crout* is essentially the procedure *det* from R 11.7 in an adopted notation with a minor improvement. The operator **invert** yields a routine which is the ALGOL 68 version of a procedure by T.J. Dekker [*].

It may be very instructive to study carefully the use of the slicing feature in the routines 8.5.3 (6) and (8).

8.5.1. Operations on vectors in E_n

```
(1)  op +    =( [ ] real u, v ) [ ] real:
               ¢ the sum of two vectors ¢
                 if   int m = lwb u, n = upb u ;
                      m = lwb v ∧ n = upb v
                 then loc [m:n] real s ;
                      for i from m to n do s[i] := u[i] + v[i] od ;
                      s
                 else error ; skip
                 fi ;

(2)  op −    =( [ ] real u, v ) [ ] real:
               ¢ the difference of two vectors ¢
                 if   int m = lwb u, n = upb u ;
                      m = lwb v ∧ n = upb v
                 then loc [m:n] real s ;
                      for i from m to n do s[i] := u[i] + v[i] od ;
                      s
                 else error ; skip
                 fi ;
```

[*] T.J. Dekker: ALGOL 60 procedures in numerical algebra, Part I
(Mathematical Centre Tracts 22,
Mathematisch Centrum, Boerhaavestraat 49, Amsterdam).

```
(3)  op x      = ( real r, [ ] real u ) [ ] real:
                   ¢ the product of a scalar and a vector ¢
                   ( int m = lwb u, n = upb u ;
                     loc [m:n] real ru ;
                     for i from m to n do ru[i] := r x u[i] od ;
                     ru ) ;

(4)  op x      = ( [ ] real u, real r ) [ ] real:
                   ¢ the product of a vector and a scalar ¢
                     r x u ;

(5)  op /      = ( [ ] real u, real r ) [ ] real:
                   ¢ the quotient of a vector and a scalar ¢
                     u x (1/r) ;

(6)  op +:=    = ( ref [ ] real u, [ ] real v ) ref [ ] real:
                     if   int m = lwb u, n = upb u ;
                          m = lwb v ∧ n = upb v
                     then for i from m to n do u[i] +:= v[i] od ;
                          u
                     else error ; skip
                     fi ;

(7)  op -:=    = ( ref [ ] real u, [ ] real v ) ref [ ] real:
                     if   int m = lwb u, n = upb u ;
                          m = lwb v ∧ n = upb v
                     then for i from m to n do u[i] -:= v[i] od ;
                          u
                     else error ; skip
                     fi ;

(8)  op x:=    = ( ref [ ] real u, real r ) ref [ ] real:
                     ( for i from lwb u to upb u do u[i] x:= r od ; u ) ;

(9)  op /:=    = ( ref [ ] real u, real r ) ref [ ] real:
                     ( for i from lwb u to upb u do u[i] /:= r od ; u ) ;
```

```
(10) op x       = ( [ ] real u, v ) real:
                  ¢ the innerproduct of two vectors ¢
                    if   int m = lwb u, n = upb u ;
                         m = lwb v ∧ n = upb v
                    then long real inpr := long 0 ;
                         for i from m to n
                         do inpr +:= leng u[i] × leng v[i] od ;
                         shorten inpr
                    else error ; skip
                    fi ;

(11) op norm    = ( [ ] real u ) real:
                  ¢ the euclidean norm of a vector ¢
                    sqrt ( u × u ) ;

(12) op e       = ( [ ] real u ) [ ] real:
                  ¢ a unit-vector in the direction of the given vector ¢
                    u / norm u ;

(13) proc norm  = ( ref [ ] real u ) ref [ ] real:
                  ¢ a reference to a vector divided by its norm
                    i.e., u := u / norm u ¢
                    ( real normu = norm u ;
                      for i from lwb u to upb u do u[i] /:= normu od ; u ) ;

(14) op =       = ( [ ] real u, v ) bool:
                    eps norm ( u−v ) ;      ¢ for eps applied to a real,
                                              see 8.4.1 (17) ¢

(15) op ≠       = ( [ ] real u, v ) bool:
                    ¬( u = v ) ;

(16) op parl    = ( [ ] real u, v ) bool:
                    e u = e v ;             ¢ for the priority of parl,
                                              see 8.4.1 (1) ¢

(17) op perp    = ( [ ] real u, v ) bool:
                    eps ( u × v ) ;
```

```
(18) proc error = void: c some action interrupting or halting the elaboration
                      of the program signalizing that two vectors or
                      matrices are of incompatible size c ;
```

8.5.2. Operations on matrices and vectors

```
(1)  op +  =( [,] real a, b )[,] real:
             ¢ the sum of two matrices ¢
               if   int m1 = 1 lwb a, m2 = 2 lwb a,
                      n1 = 1 upb a, n2 = 2 upb a ;
                    m2 = 2 lwb b ∧ n2 = 2 upb b
               then loc [m1:n1, m2:n2] real s ;
                    for j from m2 to n2
                    do s[ ,j ] := a[ ,j ] + b[ ,j ] od ; s
               else error ; skip
               fi ;

(2)  op −  =( [,] real a, b )[,] real:
             ¢ the difference of two matrices ¢
               if   int m1 = 1 lwb a, m2 = 2 lwb a,
                      n1 = 1 upb a, n2 = 2 upb a ;
                    m2 = 2 lwb b ∧ n2 = 2 upb b
               then loc [m1:n1, m2:n2] real d ;
                    for j from m2 to n2
                    do d[ ,j ] := a[ ,j ] − b[ ,j ] od ; d
               else error ; skip
               fi ;

(3)  op ×  =( real r, [,] real a )[,] real:
             ¢ the product of a scalar and a matrix ¢
              ( int m1 = 1 lwb a, m2 = 2 lwb a, n1 = 1 upb a, n2 = 2 upb a ;
                loc [m1:n1, m2:n2] real ra ;
                for j from m2 to n2 do ra[ ,j] := r × a[ ,j ] od ; ra ) ;

(4)  op ×  =( [,] real a, real r )[,] real:
             ¢ the product of a matrix and a scalar ¢
               r × a ;
```

```
(5)  op +:=   = ( ref [ , ] real a, [ , ] real b ) ref [ , ] real:
                if   int m1 = 1 lwb a, m2 = 2 lwb a,
                        n1 = 1 upb a, n2 = 2 upb a ;
                     m2 = 2 lwb b ∧ n2 = 2 upb b
                then for j from m2 to n2 do a[ , j ] +:= b[ , j ] od ; a
                else error ; skip
                fi ;

(6)  op −:=   = ( ref [ , ] real a, [ , ] real b ) ref [ , ] real:
                if   int m1 = 1 lwb a, m2 = 2 lwb a,
                        n1 = 1 upb a, n2 = 2 upb a ;
                     m2 = 2 lwb b ∧ n2 = 2 upb b
                then for j from m2 to n2 do a[ , j ] −:= b[ , j ] od ; a
                else error ; skip
                fi ;

(7)  op x:=   = ( ref [ , ] real a, real r ) ref [ , ] real:
                ( for j from 2 lwb a to 2 upb a do a[ , j ] x:= r od ; a ) ;

(8)  op x     = ( [ , ] real a, [ ] real u ) [ ] real:
                ¢ the product of a matrix and a column-vector ¢
                if   int m1 = 1 lwb a, m2 = 2 lwb a,
                        n1 = 1 upb a, n2 = 2 upb a ;
                     m2 = lwb u ∧ n2 = upb u
                then loc [m1:n1] real au ;
                     for i from m1 to n1 do au[i] := a[i, ] × u od ; au
                else error ; skip
                fi ;

(9)  op x     = ( [ ] real v, [ , ] real a ) [ ] real:
                ¢ the product of a row-vector and a matrix ¢
                if   int m1 = 1 lwb a, m2 = 2 lwb a,
                        n1 = 1 upb a, n2 = 2 upb a ;
                     lwb v = m1 ∧ upb v = n1
                then loc [m2:n2] real va ;
                     for j from m2 to n2 do va [j] := v × a[ , j ] od ; va
                else error ; skip
                fi ;
```

```
(10) op ×       =( [,] real a, b ) [,] real:
                  ¢ the product of two matrices ¢
                   ( int m1 = 1 lwb a, n1 = 1 upb a, p2 = 2 lwb b, q2 = 2 upb b ;
                     loc [m1:n1, p2:q2] real ab ;
                     for k from p2 to q2 do ab[ , k ] := a × b[ , k ] od ; ab ) ;

(11) proc icol  =( ref [,] real a, int j1, j2 ) ref [,] real:
                  ¢ interchanges a[ , j1] and a[ , j2] ¢
                   ( [1 lwb a:1 upb a] real u := a[ , j1] ;
                     a[ , j1] := a[ , j2] ; a[ , j2] := u ; a ) ;

(12) proc irow  =( ref [,] real a, int i1, i2 ) ref [,] real:
                  ¢ interchanges a[i1 , ] and a[i2 , ] ¢
                   ( [2 lwb a:2 upb a] real v := a[i1 , ] ;
                     a[i1 , ] := a[i2 , ] ; a[i2 , ] := v ; a ) ;
```

8.5.3. Operations on square matrices

```
(1)  op zero    =( ref [] real u ) ref [] real:
                   ( for i from lwb u to upb u do u[i] := 0 od ; u ) ;

(2)  op zero    =( ref [,] real a ) ref [,] real:
                   ( for i from 1 lwb a to 1 upb a do zero a[i , ] od ; a ) ;

(3)  op unit    =( ref [,] real a ) ref [,] real:
                   if    int m1 = 1 lwb a, n1 = 1 upb a ;
                         m1 = 2 lwb a ∧ n1 = 2 upb a
                   then  zero a ; for k from m1 to n1 do a[k, k] := 1 od ;
                         a
                   else  error ; skip
                   fi ;

(4)  proc iroco =( ref [,] real a, int i, j ) ref [,] real:
                   ¢ interchanges a[i , ] and a[ , j ] ¢
                     if    int m1 = 1 lwb a, n1 = 1 upb a ;
                           m1 = 2 lwb a ∧ n1 = 2 upb a
                     then  loc [1:n] real u := a[i , ] ;
                           a[i , ] := a[ , j ] ; a[ , j ] := u ;
                           a
                     else  error ; skip
                     fi ;
```

```
(5)  op trnsp   = ( ref [ , ] real a ) ref [ , ] real:
                  ¢ transposes the matrix a ¢
                    if    int m1 = 1 lwb a, n1 = 1 upb a ;
                          m1 = 2 lwb a ∧ n1 = 2 upb a
                    then for k from m1 to n1 − 1
                          do iroco ( a[k:n1, k:n1], k, k ) od ;
                          a
                    else  error ; skip
                    fi ;

(6)  proc crout = ( ref [ , ] real a, ref [ ] int p ) real:
                  ¢ By the method of Crout with row interchanges the
                    square matrix a is replaced by its triangular decompo-
                    sition a := 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. The procedure
                    crout yields the value of the determinant of a ¢
                if  int m1 = 1 lwb a, n1 = 1 upb a ;
                    m1 = 2 lwb a ∧ n1 = 2 upb a
                    ∧ m1 = lwb p ∧ n1 = upb p
                then [m1:n1] real norma;
                    for i from m1 to n1 do norma [i] := norm a[i, ] od ;
                    real   determinant := 1, r, pivot ;
                    for k from m1 to n1
                    do  int k1 = k−1 ; ref int pk = p[k] ; real max := −1 ;
                        ref [ , ] real l = a[ , m1:k1],    u = a[m1:k1, ]   ;
                        ref [ ] real ak = a[k , ]     ,   ka = a[ , k]     ,
                                     lk = l[k , ]     ,   ku = u[ , k]     ;
                        for i from k to n1
                        do      if   ( r := abs ( ka [ i ] −:= l[i , ] × ku ) /
                                       norma[i] ) > max
                                then max := r; pk := i
                                fi
                        od ;
                        if  pk ≠ k
                        then norma[pk] := norma[k] ;
                             irow ( a, pk, k ) ;
                             determinant := − determinant
                        fi ;
```

```
                    pivot := ka[k] ;
                    for j from k + 1 to n1
                    do ak[ j ] –:= ( lk × u[ , j ] ) /:= pivot od ;
                    determinant ×:= pivot
                od ;
                determinant
        else    0
        fi ;
```

(7) op det = ([,] real *a*) real:
¢ *the determinant of a square matrix* ¢
(int *m1* = *1* lwb *a*, *n1* = *1* upb *a*;
[*m1*:*n1*, *m1*:*n1*] real *lu* := *a* ;
crout (*lu*, loc[*m1*:*n1*] int)) ;

(8) op invert = (ref [,] real *a*) ref [,] real:
¢ *a reference to the inverted matrix a whose triangularly decomposed form l* × *u and pivotal indices* [*m1*:*n1*] int *p are obtained by a call of the procedure crout* (6)*; the inverse matrix supersedes the given matrix a* ¢

```
if      int m1 = 1 lwb a, n1 = 1 upb a; [m1:n1] int p ;
        crout ( a, p ) ≠ 0
then    [m1:n1] real rr, cc ;
        for k from n1 by –1 to m1
        do int k1 = k + 1 ;
           ref [ , ] real as ≒ a[k1 : n1, k1 : n1] ;
           ref [ ] real ar = a[k, k1 : n1], ac = a[k1 : n1, k] ;
           ref real akk = a[k, k] ;
           int m = n1 – k ;
           for i from m1 to m
           do rr[i] := – (ar × as[ , i] ) ;
              cc[i] := – (as[i , ] × ac ) / akk
           od ;
           ar := rr[ : m] ;
           akk := (1 – ar × ac ) / akk ;
           ac := cc[ : m]
```

```
                od ;
                for k from n1 by -1 to m1
                do int pk = p[k] ;
                   if pk ≠ k then icol ( a, k, pk ) fi
                od ;
                a
             else error ; skip
             fi ;
```

```
(9)  op inv  = ( [ , ] real a ) [ , ] real:
               ¢ an inverted copy of the given matrix a, which itself
                 remains unchanged ¢
               ( int m1 = 1 lwb a, n1 = 1 upb a,
                 m2 = 2 lwb a, n2 = 2 upb a ;
                 loc [m1:n1, m2:n2] real copya := a ;
                 invert copya ) ;
```

Vertical readers, please turn to 1.6.

8.6. Examples of transput

8.6.1. The happy family

This example is intended to show some of the things that can be done with formatted transput. The techniques shown are not necessarily the best ways of producing the particular outputs of this program, but they exemplify methods which may well be valid in more realistic situations.

The example concerns the history of the Fitzwilliam family, and the relationships between its members (or at least those relationships which they were disposed to publicise). We have eschewed the use of generators (in case, being a vertical reader, you have not yet come upon 5.7.2), but we did find it necessary to use identity-relations (5.7.4), and these are explained to you at their first occurrence.

```
    begin
comment This example concerns people: comment
    mode person = struct ( string surname, given ¢ name ¢,
                           ref person father, mother, wife ¢ or husband ¢,
                           flex [1:0] ref person children,
                           bool dead, male );
    bool male = true, female = false, alive = false, dead = true;
    ref person nobody = nil;
```

comment Sometimes it will be convenient to have a **person**'s given name and surname together: **comment**

```
proc names = (ref person pers) struct (string given, surname):
    ( given of pers, surname of pers );
```

comment All our formal-parameters will be of mode **ref person** rather than **person**, to save making unnecessary copies of **persons** (which are rather large) at run time. **comment**

comment Here is a procedure that will be used to add a little random spice to the messages that we shall produce. It yields a random integer in the range specified by its parameter. **comment**

```
proc randint = ( int range ) int :
    1 + entier ( random × range ) ;
read(last random); ¢ to start it off ¢
```

comment This program is going to print texts of variable length. We therefore have to take a newline whenever a line is full (after 80 characters, say), but before doing this we must go back to the last space and transfer the whole of the word which was about to be split onto the next line. Therefore, we shall output into a [ ] **char** instead of directly to the book. **comment**

```
file file; [1:80] char buffer;
for i to upb buffer do buffer [i] := "." od;
associate(file, buffer);
```

comment Whenever the buffer becomes full, its contents (except for the split word) must be printed in the real book. **comment**

```
proc empty buffer = (ref file f ) bool:
    ( int j := upb buffer;
      if   char number ( f ) > j
      then while buffer [ j ] ≠ "." do j −:= 1 od
      fi ;
      print ( ( buffer [ :j], newline ) ) ;
      reset ( f );
      put (f , buffer [j + 1: ] ) ;
      for i from upb buffer − j + 1 to upb buffer
      do buffer [i] := "." od;
      true);
on line end (file, empty buffer);
```

comment The [ ] **char** associated with *file* is like a book containing one page containing one line. As soon as we call *newline(file),* therefore, we shall find

that the page has overflowed (the current position will actually be at *(1,2,1)*). **comment**

```
    on page end(file, empty buffer);

    struct(int day, [1:3] char month, int year) date;
```

comment We shall frequently have occasion to print dates. Here is a **format** to do it. **comment**

```
    format datef = $ g(0)x, 3ax, 2d $;
    proc generate = ( ref person infant, father, mother,
                      string given name, bool male) void:
        if male of father ∧ ¬male of mother ∧ ¬dead of mother
        then op plusab = ( ref flex [ ] ref person names, ref person pers ) void:
                         names :=
                           ( int upb = upb names;
                             [1:upb + 1] ref person new names;
                             new names [1:upb] := names;
                             new names [upb + 1] := pers; new names);
             infant := ( surname of mother,
                         given name,
                         father,
                         mother,
                         nil,
                         ( ), ¢ not yet ! ¢
                         alive,
                         male);
             children of father plusab infant;
             children of mother plusab infant;
             if wife of father :=: mother
```

comment That was an identity-relation. If you have not yet read 5.7.4, please accept our assurance that " :=: " is a sort of operator which yields **true** if the two names which are its operands in fact are the same name. In this case, the operands were of mode **ref person**, and **if** the **persons** refed to turn out to be the same **person comment**

```
             then putf(file,
                  ( $ 2l "Birth."
                      l 4x g $,          surname of infant,
                    $ ". On " f(datef) $, date,
                    $ " to " g $,        given of mother,
                    $ ", wife of " g $,  given of father,
```

```
                $ ", a " c ( "darling",
                             "bouncing",
                             "beautiful",
                             "tiny" ) $, randint(4),
                $ x b("son", "daughter")
                  "—" $,                 male,
                $ g " . " $,             given name ))
comment       else  no comment comment
              fi
```

comment The above call of *putf* is intended to produce messages such as:

```
Birth.
    Fitzwilliam. On 3 MAR 28 to Eleanor, wife of
Ebenezer, a beautiful son - Japhet. comment
```

```
        else  stop ¢ the birth was quite impossible ¢
        fi;   ¢ end of generate ¢
```

comment The following procedure is intended to print the name of some **person**, together with details of his parents. However, if there is some doubt about the marital state of the parents, then we shall draw a discreet veil over the matter by using a different **format. comment**

```
    proc details = ( ref person pers) void:
        if    mother of pers :=: ref person(wife of father of pers)
        then  bool sex = male of pers;
              putf(file,
              ( $ g ", " $,            given of pers,
                $ c("only", "youngest", "younger",
                    "eldest", "elder", "" ) x $,
                        ( int j := 0, k;
                          ref flex [ ] ref person children =
                              children of father of pers;
                          int upb = upb children;
                          for i to upb ¢ each brother/sister of pers ¢
                          do ref person child = children [i] ;
                             (male of child = sex | j +:= 1);
                             (given of child = given of pers | k := j)
                          od;
                          ( j = 1 | 1 ¢ only ¢
                          |: k = j | 2 + abs (j = 2) ¢ youngest or younger ¢
                          |: k = 1 | 4 + abs (j = 2) ¢ eldest or elder ¢
                          | 6) ),
```

```
            $ b("son", "daughter")
              " of " $,                sex,
            $ g " and ", g x, g $,     given of father of pers,
                                       names(mother of pers) ))
        else putf(file, ($ g x, g $,   names(pers) ))
        fi; ¢ end of details ¢
    proc marry = ( ref person bride, groom) void:
        if  male of groom ∧ ¬dead of groom
            ∧ ¬male of bride ∧ ¬dead of bride
            ∧ (wife of groom :=: nobody | true
                | dead of wife of groom)
            ∧ (wife of bride ¢ sic ¢ :=: nobody | true
                | dead of wife of bride)
        then wife of groom := bride;
            wife of bride := groom;
```

comment We are now going to produce a message such as:

```
Marriage.
    Fitzwilliam/Jones. On 1 APR 24, Eleanor, only
daughter of Emrys and Myfanwy Jones to Ebenezer,
elder son of Aloysius and Anastasia Fitzwilliam.
```

comment

```
            putf(file,
            ( $ 2l "Marriage."
                l 4x g "/", g ". On " $, surname of groom, surname of bride,
              $ f(datef) ", " $,          date) );
            details(bride);
            put(file, " to ");
            details(groom);
            put(file, ".");
            surname of bride := surname of groom
        else stop ¢ the marriage is impossible, or illegal, or both ¢
        fi; ¢ end of marry ¢
    proc kill = (ref person bloke) void:
        if  ¬dead of bloke
        then dead of bloke := true;
            bool sex = male of bloke;
            bool wa ¢ wife alive ¢ =
                (wife of bloke :=: nobody | false
                | ¬dead of wife of bloke);
```

```
string ¢ name of ¢ wife = (wa | given of wife of bloke | "" );
```

comment The following call of *putf* is intended to produce messages such as:

```
Death.
    On 21 DEC 68, Ebenezer, elder son of Aloysius
and Anastasia Fitzwilliam, mourned by his devoted
wife Eleanor comment
```

```
putf(file,
($ 2l "Death."
   l 4x " On " f(datef) ", " $,   date));
details(bloke);
if    wa
then putf(file,
     ($ ", mourned by "
        b("his", "her") x,
        c("everloving", "devoted",
          "thankful") x,
        b("wife", "husband"),
        x g $,
      sex, randint(3), sex, wife))
fi;
```

comment If *bloke* has surviving descendants, the dirge continues in the following vein:

```
and his children Shem, Ham and Japhet and his
grandchildren Ananias, Azarias and Misael and his
great-grandchild Tom. comment
```

```
bool mp ¢ mourners printed ¢ := wa;
```

comment The following **proc** calls itself recursively for each generation. **comment**

```
proc print children of = ( [ ] ref person parents,
                           int generation) void:
    begin int i := 0, j := 0;
        [1:( int i := 0;
             for j to upb parents
             do i +:= upb children of parents [j] od;
             i )] ref person children, living children;
        for k to upb parents
```

```
                do    for l to upb children of parents [k]
                      do ref person child =
                              (children of parents [k] ) [l];
                         children [i +:= 1] :=
                              ( ¬dead of child
                              | living children [j +:= 1] := child
                              | child )
                      od
                od;
                if j ≠ 0
                then ¢ there are living children to be printed ¢
                      putf(file,
                      ( $ f(mp | $ " and" $ |: wa | $ "," $
                              | $ " mourned by" $),
                         x b("his", "her" ) x,
                         n(generation−1) "great-"
                         f(generation ≠ 0 | $ "grand" $ | $ $),
                         "child" f(j ≠ 1 | $ "ren" $ | $ $) x,
                         n( j ) (g, f ( (j −:= 1) + 1
                                   | $ $, $ " and " $
                                   | $ ", " $) ) $,
                         sex,
                         ( [1:j] string names;
                           for i to j do names [i] :=
                                given of living children [i] od ;
                           names) ));
                         mp := true
                fi;
                if    upb children ≠ 0
                then print children of (children, generation + 1)
                fi
          end   ¢ of print children of ¢;
          print children of (bloke, 0);
          put(file, ".")
     else stop ¢ the bloke was dead already ¢
     fi;  ¢ end of kill ¢
```

comment Now we are ready to start our tale. Since we do not wish to go right back to Adam, we shall start by declaring the story so far: **comment**

```
person aloysius :=
     ("Fitzwilliam", "Aloysius", skip, skip, skip, ( ), dead, male);
```

```
    person anastasia :=
            ("Fitzwilliam", "Anastasia", skip, skip, aloysius, ( ), dead, female);
    person ebenezer :=
            ("Fitzwilliam", "Ebenezer", aloysius, anastasia, nil, ( ), alive, male);
    person alaric :=
            ("Fitzwilliam", "Alaric", aloysius, anastasia, nil, ( ), alive, male);
comment We were unable to include anastasia as aloysius' wife when initializing
him, because her declaration had not been elaborated at that time (cf. 3.2.E7).
We can rectify this, and the similar case of their children, now comment
    wife of aloysius := anastasia;
    children of aloysius := children of anastasia := (ebenezer, alaric);
comment We shall declare the next family differently, so avoiding this
problem: comment
    person emrys, myfanwy, frederick, eleanor;
    emrys := ("Jones", "Emrys", skip, skip, myfanwy, (frederick, eleanor),
                    dead, male);
    myfanwy := ("Jones", "Myfanwy", skip, skip, emrys, children of emrys,
                    alive, female);
    frederick := ("Jones", "Frederick", emrys, myfanwy, nil, ( ),
                    alive, male);
    eleanor := ("Jones", "Eleanor", emrys, myfanwy, nil, ( ),
                    alive, female);
    person shem, ham, japhet, ananias, azarias, misael, tom;
comment These are the unborn generations, and are therefore undefined.
comment
    date := (1, "APR", 24); marry(eleanor, ebenezer);
    date := (1, "JAN", 25); generate(shem, ebenezer, eleanor,
                                        "Shem", male);
comment We don't waste much time in this program. comment
    date := (31, "MAR", 26); generate(ham, ebenezer, eleanor,
                                        "Ham", male);
    date := (3, "MAR", 28); generate(japhet, ebenezer, eleanor,
                                        "Japhet", male);
comment This will produce the example given in the proc generate. comment
    date := (14, "JUL", 48);
comment Now we need to declare some eligible young ladies. comment
    person a, b, josie, rosie;
    josie := ("Smith", "Josephine", a, b, nil, ( ), alive, female);
    rosie := ("Smith", "Rose", a, b, nil, ( ), alive, female);
    marry(josie, shem);
```

date := (23, "JAN", 49); generate(ananias, shem, josie,
"Ananias", male);
comment Well, perhaps it was premature. **comment**
date := (14, "DEC", 50); generate(azarias, shem, josie,
"Azarias", male);
date := (29, "FEB", 52); kill(josie);
comment Alas ! But . . . **comment**
date := (28, "DEC", 52); marry(rosie, shem);
comment There are some interesting ecclesiastical problems in that one.
comment
date := (14, "JAN", 54); generate(misael, shem, rosie,
"Misael", male);
comment Here is a not-so-eligible young lady: **comment**
person *x* := (**skip**, **skip**, **skip**, **skip**, **nil**, **skip**, *alive, female);*
date := (20, "DEC", 68); generate(tom, azarias, x, "Tom", male);
comment And so the permissive society has arrived. Nothing will be printed.
comment
date := (21, "DEC", 68); kill(ebenezer);
comment Poor chap! This will produce the example given in the **proc** *kill.*
comment
newline(file); newline(file) ¢ *to ensure that the final contents of buffer get printed* ¢
end

Vertical readers, please turn to 1.7.

8.7. Examples of everything

8.7.1. Analytic differentiation

This example is intended to illustrate how generators can be used to achieve more efficiently results which would formerly have necessitated the use of some list-processing language.

The following program will accept a series of expressions punched in a convenient notation, differentiate each one, and print the result in the same notation.

begin
mode formula = **struct**(**operand** *left*, **int** *operator*, **operand** *right);*
int *plus = 1, minus = 2, times = 3, over = 4, to = 5;*
mode operand = **union**(**ref formula**, **var**, **const**);
mode var = **ref struct**(**string** *s*, **var** *next);*

comment We propose to ensure below that all **var** values refer to structures containing distinct **strings**. Hence equality of two such **strings** implies identity of their **vars**. **comment**

```
    mode const = struct(real r);
```

comment The mode **const** is different from the mode **real** in order that the operator + to be declared between **operands** should not be confused with the already existing operator + between **reals**. I.e. **const**, declared thus, is not firmly related to **operand** (4.3.3), and neither is **var**. **comment**

```
    op con = (real a)const: (const b; r of b := a; b);
    op = = (const a, b)bool: (r of a = r of b);
    const zero = con 0.0, one = con 1.0;
    op + = (operand a, b)operand:
        begin
        case a in
        (const c):
            case b in
            ( const d): con (r of c + r of d)
                ¢ This is the one case where we can use the
                standard version of + (between two reals)
                to do some real arithmetic. ¢
        out (c = zero | b | goto formula) ¢ 0 + b = b ¢
        esac
    out case b in
        (const d): (d = zero | a | goto formula) ¢ a + 0 = a ¢
        out goto formula
        esac
    esac
    exit
formula: ¢ No simplification was possible. We now have no alternative
        but to generate a new piece of tree on the heap ¢
    heap formula := (a, plus, b)
    end ¢ of + ¢;
op − = (operand a, b)operand:
    begin
    case b in
    (const d):
        case a in
        (const c):
            (real x := r of c − r of d;
             (x > 0 | con x | goto formula) )
```

```
                out   (d = zero | a | goto formula) ¢ a – 0 = a ¢
                esac,
          (var t):
                case a in
                (var s): (s :=: t | zero | goto formula) ¢ a – a = 0 ¢
                out   goto formula
                esac
          out   goto formula
                ¢ We do not attempt to produce the value –b here when a is zero,
                  because our system has no provision for a monadic minus. ¢
          esac
          exit
formula:  heap  formula := (a, minus, b)
          end ¢ of – ¢;
    op x  = (operand a, b)operand:
          begin
          case a in
          (const c):
                case b in
                (const d): con (r of c x r of d)
                out   (c = zero | zero ¢ 0 x b = 0 ¢
                      |: c = one | b ¢ 1 x b = b ¢
                      | goto formula)
                esac
          out   case b in
                (const d):
                      (d = zero | zero ¢ a x 0 = 0 ¢
                      |: d = one | a ¢ a x 1 = a ¢
                      | goto formula)
                out   goto formula
                esac
          esac
          exit
formula:  heap formula := (a, times, b)
          end ¢ of x ¢;
    op /  = (operand a, b)operand:
          begin
          case a in
          (const c):
                case b in
```

```
                (const d): con (r of c / r of d)
                out (c = zero | zero | goto formula) ¢ 0 / b = 0 ¢
                esac
        out     case b in
                (const d):
                        (d = zero | goto help ¢ a/0 is undefined ¢
                        |: d = one | a ¢ a/1 = a ¢
                        | goto formula)
                out     goto formula
                esac
        esac
        exit
formula: heap formula := (a, over, b)
        end ¢ of / ¢;
  op ↑  = (operand a, b)operand:
        begin
        case b in
        (const d):
                case a in
                (const c): con exp(ln(r of c) × r of d)
                out     (d = zero | one ¢ a ↑ 0 = 1 ¢
                        |: d = one | a ¢ a ↑ 1 = a ¢
                        | goto formula)
                esac
        out     goto formula
        esac
        exit
formula: heap formula := (a, to, b)
        end ¢ of ↑ ¢;
```

comment We shall now arrange to read in expressions consisting of **strings** (for variables), constants, the operators +, −, ×, /, and ↑, and pairs of parentheses, each expression being terminated by a semicolon. Spaces and newlines will be ignored in plausible places. Each expression read in is to be stored, on the heap, as an **operand. comment**

```
    make term (stand in, "+−×/↑( )␣;");
```

comment These will be regarded as terminating a variable. **comment**

```
    on logical file end (stand in, (ref file f)bool: goto stop);
```

comment When we have read all the expressions, we shall have finished the program. **comment**

```
    var string list := nil;
comment We shall use this to record all the variables met so far. Initially,
it points to an empty chain. comment
    proc get var or const = union(var, const, void):
        begin string s, real x, file fool := stand in,
            var pointer := string list;
        on char error(fool,
            (ref file f, ref char c)void:
                (backspace(f); goto notreal) );

comment We had to invent fool, for local use, to avoid scope troubles.
We are now going to try to read a real. If the next thing on the input stream
(ignoring any spaces and newlines) is not a constant, in some readable format
(see 7.1.2), the char error event will be called, and we shall presume it was a
variable. comment

        get (fool, x);
        con x exit ¢ con x is now united to union(var, const, void) ¢
    notreal: read(s); ¢ up to one of the term chars, or end of line. ¢
        if    s = "" then empty
comment We met a term char straight away. comment
        elif  while (var(pointer) :≠: nil
                  | s of pointer ≠ s | false)
              do pointer := next of pointer od;
comment We see if we have had this string before. comment
              var(pointer) :≠: nil then pointer
comment We have, so we yield the old var containing it. comment
        else  string list := heap struct(string s, var next)
                  := (s, string list)
comment It is a newcomer. We have made a copy of it on the heap and
inserted it at the start of the chain. comment
        fi
comment The value of this conditional-clause is either a var or is empty.
It is now united to union (var, const, void) comment
        end ¢ of get var or const ¢;

comment The next procedure is going to read an operand. It reads the
first variable or constant itself, with the following operator, and then
calls itself recursively to deal with the rest. Its int formal-parameter is
used to convey the priority of the operator currently being processed.
comment
```

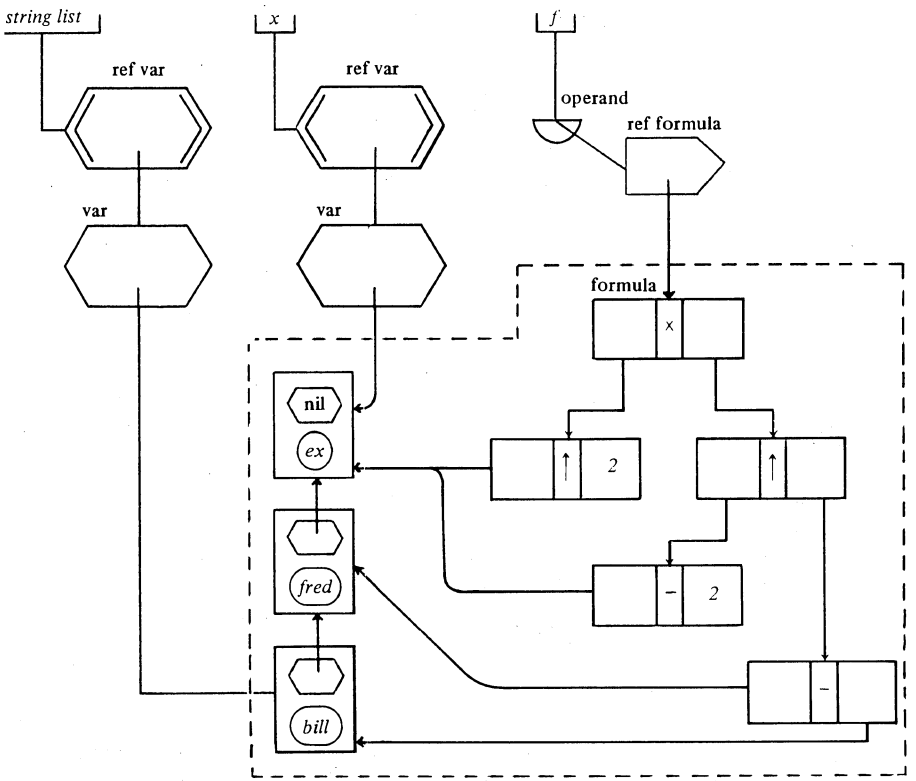
```
proc read operand = (int priority)operand:
     begin operand operand, char c, int operator;
     case get var or const in
     (const x): operand := x ,
     (var s): operand := s ,
     (void):
          if read(c); c ≠ " ( " then goto help
          else operand := read operand(1)
          fi
comment If we meet an opening parenthesis, we are to start again at
priority one. comment
     esac;
loop:     while read(c);
comment get the operator after the first operand. comment
          if ¬(char in string (c, operator, " .;)+–x/↑")
          then goto help
          fi;
          operator = 1
     do skip od ¢ ignore spaces ¢;
     operator –:= 2;
                    ¢  ; = 0
                       ) = 1
                       + = 2
                       – = 3
                       x = 4
                       / = 5
                       ↑ = 6 ¢
     if   operator ≤ priority
     then (operator > 1 ∨ operator < priority | backspace(stand in));
          operand

comment If the next operator is of lower or equal priority to the one
currently on hand, we ignore it for the time being, and yield what we
have got so far. This is in fact the only exit from this procedure. Note
that ; and ) are bound to take this path. comment
     else operand := case operator – 1 in
          operand + read operand(2),
          operand – read operand(3),
          operand x read operand(4),
          operand / read operand(5),
```

operand ↑ *read operand(6)* **esac**;
goto *loop*
fi
end ¢ *of read operand* ¢;
comment Suppose we were now to write:
var *x* := *(read operand(0)* | *(***var** *s)*: *s* | *help);*
operand *f* = *read operand(0)*
and suppose that the book being read contained, at its current position:

```
ex;
ex↑2 ×(ex-2)↑ (fred-bill);
```

then, after elaboration of these phrases, we should have the following situation:



Note that everything inside the dotted line is on the heap.

Now we shall write our procedure to differentiate an operand with respect to a variable. **comment**

```
    proc diff = (operand d, ¢ wrt ¢ var x)operand:
        case d in
        (const): zero ,
        (var s): (s :=: x | one | zero) ,
```

comment The use of an identity-relation is quick and safe here because we ensured, during *get var or const*, that if *s* and *x* refer to structures containing identical **strings**, then they do in fact refer to the same structure **comment**

```
        (ref formula form):
            begin ref operand left = left of form,
                right = right of form;
            case operator of form in
                ¢ + ¢ diff(left, x) + diff(right, x),
                ¢ – ¢ diff(left, x) – diff(right, x),
                ¢ x ¢ diff(left, x) x right + diff(right, x) x left,
                ¢ / ¢ (diff(left, x) – diff(right, x) x d) / right,
                ¢ ↑ ¢ begin
                    proc checkforx = (operand f, var x)bool:
                        (f |
                        (ref formula form):
                            checkforx(left of form, x) ∨
                                checkforx(right of form, x),
                        (var s): s :=: x
                        | false);
```

comment That was a conformity-clause, in case you didn't notice. This **proc** yields **true** if *f* is a function of *x*. **comment**

```
                    if checkforx(right, x) then goto help
```

comment The present program does not purport to cope with this case. **comment**

```
                    else right x left ↑ (right – one) x diff(left, x)
                    fi
                    end
            esac
            end
        esac;
```

comment Now we ought to have a procedure to print our results **comment**

```
    proc print operand = (operand operand, int priority)void:
        case operand in
```

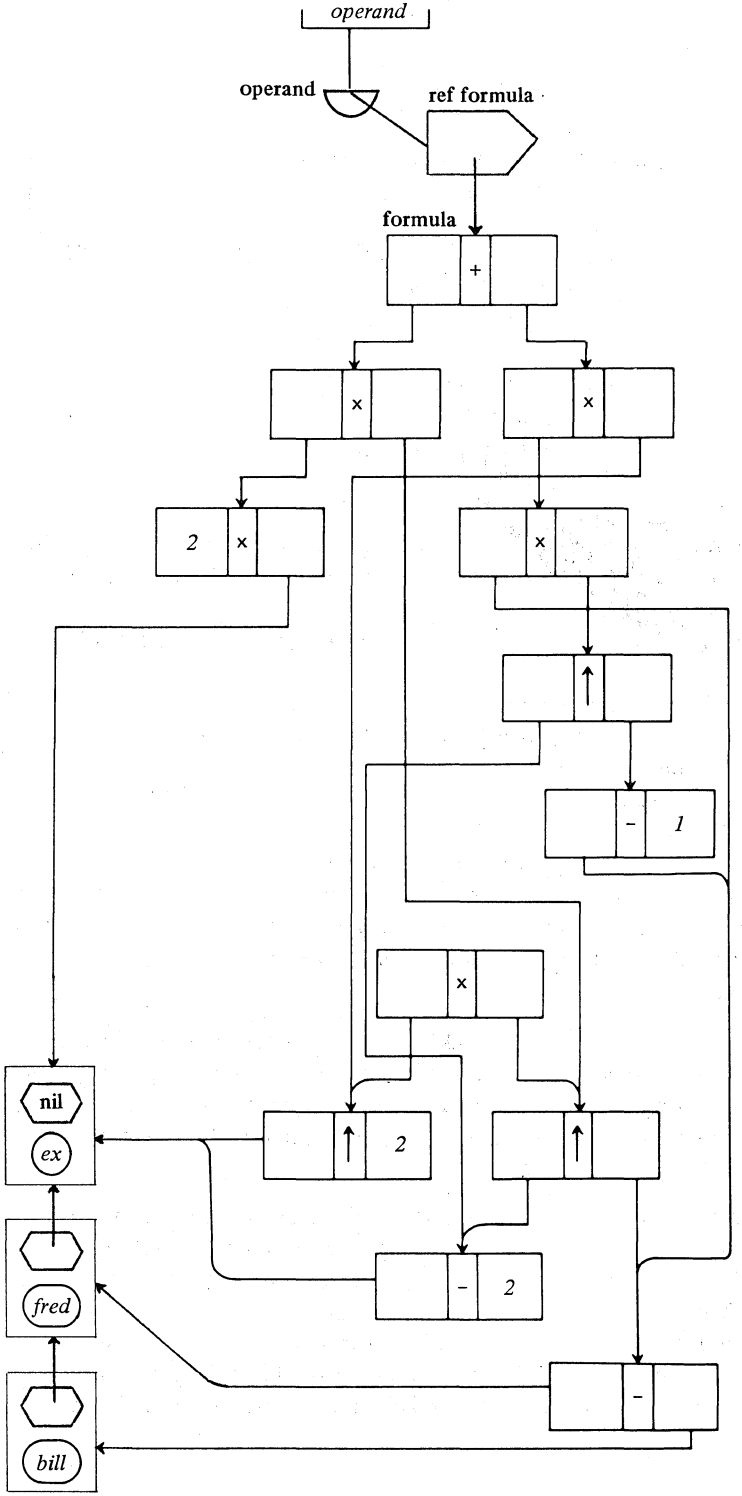
```
        (const x): printf(($2zd.3d$, r of x)),
        (var s): print(s of s),
        (ref formula f):
            if    int i; (i := entier ((operator of f + 1) / 2) ) < priority
            then  print("("); print operand(f, 0); print(")")
            else  print operand(left of f, i); print("+−x/↑"[operator of f]);
                  print operand(right of f, i)
            fi
    esac;
comment Now we come to the rest of the body of our program (you will
have noticed that it actually started when we called make term a little
way back). comment
    do  ¢ ad nauseam, or at least until the logical file end event happens ¢
        newline(stand out);
        case read operand(0) in
            (var x): print operand(diff(read operand(0), x), 0)
        out goto help
        esac
    od;
help:   print("This is not a legitimate case for this program")
comment In the best circles, the program should here print out some more
informative diagnostic message, but to include such in our present example
would be tedious rather than instructive. comment
    end
```

Suppose, now, that this program were to be offered as input the piece of text we discussed earlier. Then, during the elaboration of the *print operand*, its formal-parameter *operand* would be as in the scheme on the next page.

The bottom part of the picture is, of course, that which you saw in the previous diagram.

Note that all the items shown here are on the heap, and that at this instant there is only one of them that is pointed to, and which is therefore available for garbage collection. As soon as *print operand* has been elaborated, they will all become garbage.

If you follow through the operation of the program on this example, you will see that the number of items generated would have been much greater had it not been for the facilities for dealing with singular cases (*1 × x*, *a + 0*,


operand
operand
ref formula
formula
+
×
×
2
×
×
↑
1
×
nil
ex
2
↑
↑
fred
2
bill

etc.) in the operation-declarations. When the elaboration of *print operand* is over, it will have printed the following:

```
2.000 x ex x (ex-  2.000) ↑ (fred-bill) + (fred-bill)
x (ex-  2.000) ↑ (fred-bill-  1.000) x ex ↑ 2
```

For a slightly different treatment of problems of this nature, you might now like to study example R 11.10 in the Report.

APPENDICES

APPENDIX 1. Alternative Representations

As explained in 0.2 and 1.3.3, the Report provides representations for all the symbols needed to construct an ALGOL 68 program, and in this Introduction we have adhered to these (as indeed will most implementations). In many cases, however, the Report provides two or more representations for the one symbol. Usually, one is a bold word (1.3.2) and the other a distinct graphic mark. These bold words are described as "reserved", which means that you are forbidden to redeclare them as mode-indications or operators of your own (imagine the ambiguities that would arise if you could declare **begin** to be an operator). Here then is a list of all the reserved bold words, together with their graphic alternatives, if any.

begin end	( )	3.2.4.1
if then elif else fi	(| |: |)	3.2.4.2
case in ouse out esac	(| |: |)	3.2.4.3, 3.6
for from by to while do od		3.5.2
par		3.7.2
exit		3.1.4
at	@	5.5.1.3
is	:=:	5.7.4
isnt	:≠: :/=:	5.7.4
of		5.4.2
goto go to		4.7.1
skip	~	5.1.4.2
comment co	¢ #	1.3.2
pragmat pr		1.3.2
true false empty		5.1.1.1, 5.6.1
nil	°	5.2.3
long short ref loc heap struct flex proc union		2
mode prio op		2.3, 4.3.1, 4.3.2
int real bool char compl bits bytes string		2
sema file channel format		3.7.2, 7.2.1, 7.6.2
void		1.2.3
[ ]	( )	5.5.1.3
10	\	5.5.1.1
(i.e. a blank)	_	5.5.1.1

The bold words defined in the standard-prelude as operators, however, are

not reserved (you can redeclare them if you like). Moreover, where several symbols are declared as operators with the same function, they really ought not to be regarded as alternative versions of the same symbol for, if you redeclare the meaning of **not**, for example, the meaning of ¬ and of ~ remains unchanged, unless you redeclare them too. Bearing this in mind, we give now a list of all those functions for which several operators are defined:

¬	~	**not**			6.1.1
entier	⌊				6.1.1
i	⊥	+x	+*		6.1.2
↑	**	**up**			6.1.2
shl	**up**	↑			6.1.2
shr	**down**	↓			6.1.2
÷	%	**over**			6.1.2
mod	÷x	%x	÷*	%*	6.1.2
x	*				6.1.2
elem	□				6.1.2
<		**lt**			6.1.2
≤	<=	**le**			6.1.2
≥	>=	**ge**			6.1.2
>		**gt**			6.1.2
=		**eq**			6.1.2
≠	/=	**ne**			6.1.2
∧	&	**and**			6.1.2
∨		**or**			6.1.2
timesab	x:=	*:=			6.3
overab	÷:=	%:=			6.3
divab	/:=				6.3
modab	÷x:=	%x:=	÷*:=	%*:=	6.3
plusab	+:=				6.3
plusto	+=:				6.3
minusab	−:=				6.3
lwb	⌊				6.3
upb	⌈				6.5

APPENDIX 2. Sample Declarations

The following is a summary of the sample declarations introduced in Chapter 0, with some others from R 1.1.2.

```
int i, j, k, m, n;
real a, b, x, y;
real e = c a real value close to the base of natural logarithms,
        i.e. 2.718281828...c;
bool p, q, overflow;
char c;
ref real xx, yy;
compl w, z;
format f;
bytes r;
bits t;
mode vec = struct (real xcoord, ycoord, zcoord);
vec v1, v2, v3;
mode rational = struct (int numerator, denominator);
rational r1, r2, r3;
string s;
union (int, real) uir;
proc void task1, task2;
[1 : n] real x1, y1;
flex [1 : n] real a1;
[1 : m, 1 : n] real x2;
[1 : n, 1 : n] real y2;
[1 : n] int i1;
[1 : m, 1 : n] int i2;
[1 : n] compl z1;
proc x or y = ref real: (random <. 5 | x | y);
proc ncos = (int i) real: cos (2 x pi x i/n);
proc nsin = (int i) real: sin (2 x pi x i/n);
proc finish = void: goto stop;
mode book = struct (string text, ref book next);
book draft;
op i = (int a) compl: (0, a);
op i = (real a) compl: (0, a);
princeton: grenoble: st pierre de chartreuse: kootwijk:
warsaw: zandvoort: amsterdam: tirrenia: north berwick: munich:
```

APPENDIX 3. Glossary

The Report defines a vast number of new technical terms. In this Informal Introduction we have used those of them which we think could or should come into general use within the computing community. We have also invented one or two of our own (we hope they will be acceptable to you – they are marked with an * in the lists below), and occasionally the meaning of one of our terms differs slightly from its meaning in the Report (as marked with a † below).

We define below the meaning of the principal terms. For the others, you may follow the references given to their "defining occurrences" in our text. Usually, there are two such references – one to the basic concept in Chapter 1, and one to its practical realisation in Chapters 2 through 7.

1. Internal objects and modes

internal object	1.1.1	An object which is stored and manipulated inside the computer; i.e. an instance of a value.
* instance (of a value)	1.1.1	*a:=2; b:=2;* There are now two instances of the value "*2*". If we assign to *b* a "*3*", then we may say "an instance of *2* has been superseded by an instance of *3*", but not that "the value of *b*, which was *2*, is now *3*".
mode	1.1.1 1.2.3	The property of a value (and therefore of an instance) which defines the class to which it belongs, i.e. the amount of storage space it requires, its compatibility with other values with which it may be confronted, etc. A mode can also be a property of an external object if that object yields a value of that mode.
value	1.1.1	The ultimate object processed by the operations of the language; e.g. a number, a character. a structure, etc.

In our text, we use formal-declarers (e.g. **int**, **ref real**, **[ ] ref compl**) to specify modes, and also to indicate values of those modes. There is no

ambiguity. If we use such a declarer as a noun, it indicates (an instance of) a value. If we use it as an adjective, it is a mode – "The mode of an **int** is **int**".

primitive modes	1.2.3 2.1.1	The built-in modes in terms of which all other modes may be constructed.
int	2.1.1	
real	2.1.1	
bool	2.1.1	
char	2.1.1	
bits	2.7.1	
bytes	2.7.1	
void	1.2.3	
ref	1.2.3	Prefixes used to construct declarers (e.g. **ref real**, **union** *(***int**, **real***)*, **proc** *(***real***)* **int**) or to specify all the modes of the appropriate class (e.g. **proc** modes, 'row of' modes, etc.), or to indicate values of those classes of modes.
'row of'	1.4.0 2.5.1	
rowed		
[ ]		
struct	1.2.3 2.4.1	
union	1.2.3 2.6.1	
proc	1.2.3 4.2.1	
long	1.2.3 2.7.2	
short	1.2.3 2.7.2	
string	2.5.3	Derived modes built into the language.
sema	3.7.2	
file	7.2.1	
channel	7.2.1	
format	7.6.2	
primitive value		A value of mode **int**, **real**, **bool**, **char**, **bits** or **bytes**.
name	1.1.1	A value whose mode is **ref** some other mode, and which refers to a value of that other mode.
subname	1.4.1.2	If a name N refers to a multiple (a structure) V then the subnames of N refer to the subvalues (the fields) of V.
fixed name	2.5.2.1	The bounds of a multiple assigned to a fixed name must match the existing bounds.
flexible name	1.5.3 2.5.2.1	Assignation of a multiple to a flexible name may change the existing bounds.
transient name	5.5.1.3	A subname of a flexible name.

* structure structured value **struct**	1.4.0 2.4.1	A value consisting of several fields, each being a value of some other mode.
field	1.4.0	
multiple value * multiple	1.4.0 1.5.1	A value consisting of a sequence of values, its "elements", of some (same) mode, together with a descriptor.
element	1.4.0 1.5.1	
subvalue	1.5.2 5.5.1.3	A subset of the elements of a multiple, as specified by a different descriptor.
descriptor	1.5.1	See under external objects.
routine	1.1.4 4.2.2	The internal equivalent of a routine-text – a value of a **proc** mode.
† constant	1.1.1 1.2.2.1	A value which has been ascribed to an external object. Therefore, no name refers to it, and it cannot be changed. See also under external objects.
† variable	1.1.2.1 1.2.2.4	An instance of a value to which a name refers (so that it can be changed), together with that name.
subscript	1.5.1	
boundpair	1.5.1	

2. External objects

Most of the terms defined below are in fact what the Report would class as "paranotions" (see R 1.1.4.2). For this reason, at their defining occurrences in our text they are enclosed between single quotes (e.g. 'serial-clause') and they are hyphenated, whereas our defining occurrences of other technical terms are in double quotes.

construct (or external object)	1.1.1	A part of a program text, as classified below.
program	1.1	The program text provided by the user, together with the standard- and library-preludes and the standard-postlude.
particular-program	1.1 3.1	The program text provided by the user, on its own.

standard-prelude	1.1 4.1 6	The declarations already built into the language.
library-prelude	1.1	Additional built in declarations, peculiar to the particular implementation.
* standard-postlude	1.1	The administration of the completion of the program, following the label *stop*.
phrase	1.1.3	A declaration or a clause.
declaration	1.1.3 2	
* collateral-declaration	1.1.3 2.1.2	A list of declarations, separated by commas.
identifier-declaration	1.1.2 2.2	
identity-declaration	1.2.2 2.2.1	Ascribes a value to an identifier
variable-declaration	1.1.2.1 2.1.2	
routine-identity-declaration	1.2.3.1 4.2.2.1	
routine-variable-declaration	4.2.2.1	
mode-declaration	1.3.3.1	Causes a mode-indication to specify a mode.
mode-indication	1.3.3.1	A bold word that has been declared to specify a mode.
priority-declaration	1.3.3.3 4.3.1	
operation-declaration	1.3.3.2 4.3.2	Ascribes a routine to an operator.
declarer		An external object which specifies some mode.
actual-declarer	2.1.2 2.2.2 2.5.2.2	
formal-declarer	1.2.2 2.2.1	
† descriptor	1.5.1	At run time, an internal object has to be kept for each multiple value and subvalue to record the values of its bounds. This is called, in the Report, a "descriptor". We also apply this term to that external object which conveys the same information, viz. the list of boundpairs enclosed between "[" and "]" which appears in the actual-declarer of a 'row of' mode.

boundpair	1.5.1	
bound	1.5.1	
parameter		
formal-parameter	1.2.2 2.2.1	
actual-parameter	1.2.2 2.2.1	
clause		
ENCLOSED-clause	3.2.4	
closed-clause	1.1.3 3.2.4.1	A serial-clause enclosed between **begin** and **end** or "(" and ")".
collateral-clause	3.7.1	
row-display	3.5.1	
vacuum	3.5.1	
structure-display	3.4	
and-also-symbol	1.1.3	
parallel-clause	3.7.2	A void-collateral-clause preceded by **par**.
conditional-clause	3.2.4.2	**if XXXX then XXXX else XXXX fi**
case-clause	3.2.4.3	**case XXXX in XXXX, XXXX out XXXX esac**
conformity-clause	1.6.2 3.6	
specification	3.6	
loop-clause	3.5.2	
serial-clause	1.1.3 3.1 3.1.5	
completer	3.1.4	Constituents of serial-clauses.
label	3.1.2	
go-on-symbol	1.1.3 3.1	A semicolon.
enquiry-clause	3.2.4.2	
range	1.1.3 3.2.1	A piece of program text (usually a serial-clause) which demarcates the scope of the variables which are locally generated during its elaboration.
reach	3.2.1	A range, with the exclusion of all ranges contained within it.
unit	1.1.3 5.1	
coercend	5.1.0.1	
* quaternary	5.1.0.1	The same thing as a unit.
assignation	1.1.2.2 5.1.4.1	:=

destination	1.1.2.2	The LHS of an assignation
source	1.1.2.2	The RHS of an assignation
identity-relation	1.7.2 5.7.4	:=: or :≠:
routine-text	1.1.4 4.2.2.1	
tertiary	5.1.0.1	
formula	1.1.4 5.1.0.1 5.1.3	
operator	1.1.4 6.1	
monadic-operator	1.3.3.2 5.1.3	With one following operand.
dyadic-operator	1.3.3.2 5.1.3	Between two operands.
operand	1.1.4 5.1.3	A secondary or another formula.
secondary	5.1.0.1	
selection	5.4.2	of
field-selector	1.4.1 2.4.1	
generator	1.2.2.3 5.7.2	The means of making storage space available for variables.
loc generator	1.2.2.3 5.7.2.1	
heap generator	5.7.2.2	
primary	5.1.0.1	
denotation	5.1.1.1	Denotations are provided for **ints**, **reals**, **bools**, **chars**, **strings**, **bits**, and the **long**(s) versions (if any) of these.
cast	1.2.2.5 5.1.1.3	
format-text	7.6.1	The specification of the layout of the characters produced or expected during transput.
picture	7.6.1	To be matched against a single value.
insertion	7.6.1	
literal	7.6.1.1	
alignment	7.6.1.2	
frame	7.6.1.3	
replicator	7.6.1.4	
dynamic replicator	7.6.1.4	
† collection	7.6.1.4	A collection of pictures, to be replicated.
identifier	1.1.2 5.1.1.2	

	defining-identifier	3.2.3	
	applied-identifier	3.2.3 5.1.1.2	
	call	5.2.1	Of a procedure with parameters.
	slice	1.5.4 5.5.1.3	
	† indexer	1.5.2 5.5.1.3	
	trimscript	1.5.2.1 5.5.1.3	
	trimmer	1.5.2.1 5.5.1.3	
	subscript	1.5.2.1 5.5.1.3	
	revised-lower-bound	5.5.1.3	
	expression	3.1 5.1.0.1	A unit which yields a value.
	statement	3.1 5.1.0.1	A unit which yields **void**.
†	constant	1.2.2.1	A coercend (usually an identifier) which yields a value which is not a name. See also under internal objects.
	variable	1.2.2.4	A coercend (usually an identifier) which yields a name. See also under internal objects.
*	procedure	4.2.1	A coercend (usually an identifier or a routine-text) which yields a value of a **proc** mode.
*	LHS	1.1.2.2	The left hand side of an assignation or identity-declaration.
*	RHS	1.1.2.2	The right hand side.
	symbol	1.3.1	The smallest external object, out of which all the others are constructed, e.g. *a*, +, **begin**, etc.
*	bold word	1.3.2	A symbol, made up of underlined or bold faced characters (or otherwise), invented for use as a mode-indication or an operator.
	indicator	1.1.1	an identifier, a mode-indication or an operator.
	comment	1.3.2	May be inserted between any two symbols (except within a denotation or an identifier).
	pragmat	1.3.2	

3. Technical terms

contraction	1.1.3 2.1.2	Omission of redundant declarers, as in **real** *a, b.*
sublanguage	Appendix 4	A language (not ALGOL 68) all of whose particular-programs are also particular-programs of ALGOL 68 and have the same meaning [see R 2.2.2.c].
superlanguage	Appendix 4	
* to stop	1.3.2	To construct bold words out of sequences of letters and digits by underlining, prefixing with a point, etc.
firmly related modes	4.3.3	
* instance	1.1.1	Values have instances
occurrence	1.1.5	Constructs have occurrences.
defining occurrence	1.1.5 3.2.3	
applied occurrence	1.1.5 3.2.3	
scope	1.1.3 3.2.2	The scope of a value is the range (possibly the whole program) in which it is available for use.
reach (of a defining-indicator)	3.2.3	The part of a program text from which the given defining-indicator may be identified.
to yield	1.1.1	A construct yields a value.
to refer to	1.1.1	A name refers to a value.
to conform to	1.6.2 3.6	The yield of a **union** conforms to the mode actually in residence.
to identify	1.1.5 3.2.3	An applied occurrence identifies a defining occurrence.
to specify	1.4.1	A declarer specifies a mode.
to select	5.4.2	A field-selector selects a field from a structure.
to develop	1.3.3.1	To derive the mode specified by a mode-indication.
elaboration	1.1.1	The process of inspecting a construct and causing the corresponding actions (as specified by the semantics of the Report) to take place.

actions	1.1.1	The elementary operations (how elementary is not defined) which, when performed in the appropriate sequence, constitute the elaboration of a construct.
collateral elaboration	1.1.2.2 3.7.1	An elaboration in which the actions required to elaborate certain phrases are merged in time, in a manner left undefined.
* to supersede	1.1.2.2	To replace an instance of a value by another instance of a value.
to call	1.1.4 4.2.2	To initiate the elaboration of a procedure.
to parametrize	1.2.3.2.1	To substitute actual-parameters for formal ones.
to complete	3.1.4	To finish the elaboration of a serial-clause by yielding a value, or **void** (from its final unit, or from an **exit**).
to terminate	3.1.4	To finish the elaboration of a serial-clause abruptly, as when a jump is made out of it, or when some other elaboration collateral with it is terminated.
to halt	3.7.2	To suspend the elaboration of a serial-clause temporarily, as in the operator **down**.
to resume	3.7.2	To resume the elaboration of a clause that had been halted, as in the operator **up**.
to ascribe	1.1.1	A value is ascribed to an indicator.
to assign	1.1.2.2	A value is assigned to (the location addressed by) a name.
coercion	1.1.6 5.1.0	The changing of the mode of a coercend to that required by its context, with a corresponding modification to the actions performed upon elaboration of that coercend.
dereferencing	1.1.6 5.1.0.3	
widening	5.1.0.4	

deproceduring	4.2.2.2 5.2.0.2	
rowing	5.5.0	
uniting	5.6.0	
voiding	5.7.0.1	
balancing	5.2.0.1	
context	5.1.0.2	The context of a coercend is its relationship to the clause in which it occurs. With each such context is associated a strength.
strong	5.1.0.2	
firm	5.1.0.2	
weak	5.1.0.2	
meek	5.1.0.2	
soft	5.1.0.2	
STOWED	1.4.0	Structured or rowed.
stack (the)	1.2.2.3	That part of the storage of the computer where internal objects created by **loc** generators are kept.
heap (the)	5.7.2.2	That part of the storage of the computer where internal objects that cannot be held on the stack are kept.
flat descriptor	1.5.1 2.5.2.2	A descriptor in which at least one upper-bound is less than its matching lower-bound.
acceptable	1.6.1.1	A value is acceptable to a **union** mode if its mode can be united to that **union**.
garbage collection	5.7.2.2	The process of recovering storage space on the heap from internal objects that are no longer accessible to the program.
undefined	1.1.2.2	If the result of some elaboration is said to be undefined, then the Report does not oblige an ALGOL 68 implementation to produce any specific result. In practice, implementations may produce

		diagnostic message, or go completely haywire.
environment enquiry	6.2.1	A constant made available by the standard or library prelude to convey information about some property of a particular implementation.
transput	7.1	Input and output and transfers to backing media.
formatless transput	7.1	
formatted transput	7.6	
binary transput	7.7	
book	7.2.1	The input/output medium in use, together with its contents.
current position	7.2.1	The current page, line and character number of the book.
logical end of file	7.2.1	The last used page, line and character number of the book.
physical end of file	7.2.1	The last existing page, line and character number of the book.
channel	7.2.1	The facility through which transput to (from) the book takes place.
data list	7.1.1 7.1.2	A row-display of values used as a parameter of a transput procedure.
to open	7.2.1 7.2.3	To attach a book to a **file** through a **channel**.
to close	7.2.3	To disconnect a book from a **file**.
to straighten	7.4.1 7.5.1	To cause the elements of a multiple value or the fields of a structure to be presented in sequence as a stream of primitive (also **compl** or **string**) values.

APPENDIX 4. The Sublanguage

The language realised by a particular implementation may differ from ALGOL 68 as defined by the Report. The differences may be of two sorts:

> **Sublanguages.** If we omit some features of the language, or impose extra restrictions, then we have a "sublanguage" [R 2.2.2.c]. A particular-program written in a sublanguage should run without further ado on an implementation of the full language.
>
> **Superlanguages.** If we add new features, or define the results of programs whose results are at present left undefined, then we have a "superlanguage". A particular-program written in canonical ALGOL 68 is therefore automatically correct in any superlanguage.

Although many sublanguages of ALGOL 68 are possible, there is one particular sublanguage that has been accorded official recognition by IFIP Working Group 2.1*, and which is usually referred to as "ALGOL 68S".

ALGOL 68S is intended for use primarily in numerical and related areas. In spite of the various features of full ALGOL 68 that have been left out of it, it is still a viable language in its own right, within its field of application.

The omitted features have been chosen principally with a view to simplifying the compilation process, enabling the sublanguage to be implemented on mini computers with as little store as 16K words of 16 bits. Another design aim was to ensure that programs could be parsed and object code generated in one pass through the source text. Although this, of course, aids compilation on small machines, it has the further advantage, even on large machines with complex operating systems, that overlaying of the compiler is avoided. Since the operating-system overhead associated with bringing the compiler and/or its overlays into store may account for the bulk of the compilation cost for sufficiently small programs, it is expected that implementations of this sublanguage will be used for teaching purposes, since students' programs are typically small, but large in number.

* P.G. Hibbard, A. Sublanguage of ALGOL 68, SIGPLAN Notices **12** (5) (1977).

Restrictions

1. Modes

There are fewer modes than in the full language:

There are no **unions**.

There is no **flex** (but there are special provisions for **string**).

Structures may not contain multiples (e.g. no **struct(int** *no,* [ ] **real** *elems)*) and multiples of multiples (such as [ ] [ ] **int**) are forbidden.

In an actual-declarer, you may not write [*3*] **int** meaning [*1:3*] **int** (see 2.5.2.2.E7*), and quirks like **mode a** = [*1:(***a** *b = (4, 6, 8); b*[*i*]*)*] **int** (observe how **a** is used in the actual-bounds before its declaration is complete) are forbidden.

You must not expect standard-prelude operators to work (even with restricted precision) on **long** or **short** modes (2.7.2) beyond those implied by *int lengths*, *int shorths*, etc. (6.7.1).

If your implementation forces you to use "*(*" and "*)*" in place of "[" and "]", you are not allowed to omit the **loc** in variable-declarations such as **loc** *(1:n)* **real** *x1*.

2. Omitted constructs

No **heap** generators (5.7.2.2) (this immediately rules out all applications of a "list processing" nature, but it simplifies the run-time system considerably by removing the need for a garbage collector).

No parallel-clauses (3.7.2) (i.e. no **up**, **down** or **sema**).

No void-collateral-clauses (3.7.1).

No vacuums (3.5.1.E4) (you don't really need them, because of the absence of **flex**).

Jumps must be explicit (i.e. **go to** *l* or **goto** *l*, but not just *l* – see 4.7.1.E2).

No conformity-clauses (3.6) (because there are no **unions**).

No **empty** (5.6.1) (no **unions** again).

No $_{10}$ (or its alternative \). This is no hardship because you just use *1230e–1* in place of 1230_{10}*–1* (5.1.1.1). (See also Appendix 5).

No procedured jumps (4.7.2).

3. Textual order

The following restrictions arise because of the requirement for one-pass compilation:

All declarations of indicators (i.e. of identifiers, operators and mode-indications) must precede the first applied occurrences (3.2.3) which

identify them. This is normally good programming practice anyway (moreover, see 3.2.3.E7) and the only case where you might regret the restriction is that of mutually recursive pairs of procedures (you would have to declare one of them as a **proc** variable in the sublanguage) or of mode-indications (but, the sublanguage not being suitable for list processing anyway, the need for these is less likely to arise).

Priority-declarations of operators (4.3.1) must precede their corresponding operation-declarations (4.3.2) and, once a priority has been given for an operator, it may not be changed again within an inner range (which prevents you, among other things, from altering the priority of the standard-prelude operators). Also, a bold word declared as an operator may not subsequently be used as a mode-indication in an inner range, and vice-versa.

The "firmly related" restriction on declaring a given operator to apply separately to not-too-different modes, as given in 4.3.3, is made more severe. The new "meekly related" condition applies whenever there exists a common mode to which the two modes in question can be meekly coerced (e.g. **proc real** and **ref real** are meekly related) *.

Structure- and row-displays (3.4 and 3.5.1) may not be used in the strong position of a firm (or weaker) balance (5.2.0.1). I.e. although *z +:= (p | x | w)* would be accepted in the sublanguage (because of the balancing, *x* gets widened to **compl** even though the conditional-clause is in a firm context), *z +:= (p | (+1, −1) | w)* would not. Likewise, structure- and row-displays may not occur as the first (effectively the only) unit of a closed-clause (as in *((+1, −1))*). These restrictions are no great hindrance to the programmer (a cast (5.1.1.3) can always resolve the matter) but, for the compiler writer, they mean that as soon as he starts to compile such a display, he knows what its mode is meant to be.

loc generators (5.7.2.1) may not precede the first declaration in their range (the compiler must be able to know, at the time it encounters the **loc**, whether the range in question is going to be a local one or not (5.7.2.1), they may not stand as bounds in actual-declarers (but who would want to write **[***1*:**loc int] real** anyway), and they may not be operands (as in **loc int** + *1* – again not in the least useful).

A jump may not cause the elaboration of a declaration to be bypassed (as in *((p |* **goto** *l); real x;* **c** *statements involving x* **c**; *l*: **c** *statements not involving x* **c***)* which is legal, though hardly good style, in the full language).

* Moreover, the new operator may not be meekly related even to another operator declared in an outer reach.

4. Strings

string is actually a different mode from [ ] **char** in the sublanguage. However, many operations (e.g. slicing) work with both modes and a strong coercion from **string** to [ ] **char** is provided, so that you will hardly notice the difference. The following are the few cases which might arise:

The operators <, ≤, =, ≠, ≥, >, +, x, **plusab** and **plusto** work as usual for **string**, but not for [ ] **char**.

Slices work for both modes, but in the case of **string** the slice cannot yield a name (thus *s*[*2*] := "*A*" is excluded, but not **char** *c* := *s*[*2*]).

The lower-bound of a **string** is always *1* (and hence revised-lower-bounds (5.5.1.3) are never needed).

Contexts (even strong ones) expecting a **string** cannot accept a [ ] **char** (as in **proc** *p* = *(***string** *s)* **void**: **skip**; [*1*:*n*, *1*:*m*] **char** *rrc*; *p(rrc*[*i*,] *)*). The converse (**string** where [ ] **char** is expected) is all right, because of the extra strong coercion (as in *rrc*[*i*,] := *s*).

The rowing coercion can never yield a name (see 5.5.1.3.E20).

5. Transput

Many of the less used and more exotic transput features are omitted, as follows:

There are no **formats**.

Omitted procedures are *stand conv* (7.4.3), *make conv*, *make term* (7.4.2), *on format end* (7.4.4.5), *on value error* (7.4.4.6), *on char error* (7.4.4.7), *reidf*, *lock*, *scratch*, *create* (7.2.3), *backspace*, *set char number* (7.2.5) and *char in string* (7.5.2).

All the environment enquiries given in 6.2.1, 6.7.1, 7.2.2 and 7.5.3 are omitted, with the exception of *max int*, *max real*, *small real* (and their *long*(s) versions), *chan*, *stand in channel*, *stand out channel* and *stand back channel.*

Conclusion

It will thus be seen that most of the restrictions in ALGOL 68S will hardly be noticed by the average programmer (although they all help the compiler writer considerably). It will be noticed that, in the lists above, those restrictions likely to be of practical importance have been given first. These include the lack of **unions**, **flex**, parallel-clauses, **heap** generators, the requirement for defining before applying and **formats**. Even without all these, you still have a very powerful language.

APPENDIX 5. The Standard Hardware Representation

Of the large number of symbols that could be used in an ALGOL 68 program (see Appendix 1), most computer installations will be able to use but a few due to the limitations of their character codes. Indeed, the symbols used in this book (especially those such as ×, ∨ ∧, ¬ and $_{10}$ and the bold words) have been chosen, from amongst those permitted by the Report, for their clarity and conciseness rather than for their ready availability on real computers.

It is thus the responsibility of each implementor to choose which symbols he will represent (where there are alternatives he is not obliged to provide more than one) and how he will represent them. In order to give guidance to implementors and to facilitate the portability of programs between different implementations, IFIP Working Group 2.1 has approved a standard representation* to which it is expected that many implementations will adhere.

The standard seeks to represent all ALGOL 68 programs using only 60 "worthy characters", which have been chosen because of their ready availability in the majority of modern character codes, such as ISO (including its American version ASCII) and EBCDIC (although even these are not as well standardized as is popularly imagined, there being problems with specialized Natonal Characters in ISO and various positions for (or even a complete absence of) [and] in EBCDIC). The 60 worthy characters are:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
space "#$%'( )*+,-./:;<=>@[ ]_|
```

This is not to say that all these will appear on your keyboard. A local convention may tell you to punch "←" for "_" and "a" for "A". The important thing is that there should be a one-to-one correspondence between the worthy characters and the codes recognized by your machine so that, at the worst, transferring a program to another machine should require only a one-to-one transliteration of codes. To these 60 worthy characters may be added, optionally, a second alphabet ("a" to "z") but, except in the case of

* Wilfred J. Hansen and Hendrik Boom, The Report on the Standard Hardware Representation for ALGOL 68, ALGOL Bulletin, AB40, also in SIGPLAN Notices **12** (5) (1977).

UPPER stropping and **strings** (see below), "a" means the same as "A" and "z" means the same as "Z".

The manner in which these worthy characters are used to represent ALGOL 68 symbols such as ":=", ":=:" and "+:=" is quite obvious. For the symbols not immediately representable (e.g. ≠, ×, ≤, ÷) Appendix 1 gives representable alternatives (e.g. /=, *, <=, %) and for $_{10}$ or \ you can always write "*e*" (5.1.1.1). The letters A to Z can obviously be used to represent identifiers, which leaves us with just the "stropping convention" (1.3.2) to be used for the bold words – i.e. for mode-indications such as **real** and **int**, for operators such as **abs** and **entier**, and for delimiter words such as **begin** and **end**. For this, three distinct conventions, "POINT", "UPPER" and "RES", are prescribed.

1. POINT stropping

Each bold word is preceded by a point and followed by a "disjunctor". A "disjunctor" is anything other than a letter or a digit or an underscore (so that, if the bold word is to be followed by an identifier (which starts, of course, with a letter) you had better insert a space in between to act as the disjunctor – so that .REAL X means **real** *x* but .REALX means **realx**). Even if a second alphabet of letters ("a" to "z") is provided, no distinction is made between "a" and "A", so that .real x is no different from .REAL X.

Example:

```
.BEGIN .REF .LONG .REAL X := .LOC .LONG .REAL :=
    .LONG 3.141592654; X .MINUSAB LONG PI;
    PRINT(X)
    .END
```

If you want to see which spaces were actually essential as disjunctors, observe that the following is exactly (but confusingly) equivalent:

```
.BEGIN.REF.LONG.REAL X:=.LOC.LONG.REAL:=
.LONG 3.141592654;X.MINUSAB LONGPI;PRINT(X).END
```

The POINT stropping regime is introduced by the pragmat (1.3.2) **pr** *point* **pr** (which will usually appear as .PR POINT .PR) but, since point stropping continues to be valid even in the other two regimes about to be introduced, the chief effect of this pragmat is to turn those other regimes off.

2. UPPER stropping

Assuming the extra alphabet of lower-case letters ("a"to"z") to have been provided, all words written in upper case are deemed to be bold words in this regime, and so identifiers must be rewritten using the new lower-case facility. Example:

```
BEGIN REF LONG REAL x := LOC LONG REAL :=
    LONG 3.141592654; x MINUSAB long pi;
    print(x)
END
```

Disjunctors are now needed between two adjacent bold words, of course, but no longer between a bold word and an identifier. The shortest way of writing the above example is therefore:

```
BEGIN REF LONG REALx:=LOC LONG REAL:=LONG 3.141592654;
xMINUSABlongpi;print(x)END
```

You may still introduce bold words with a point, so that in this regime REAL, REALand .real but not real) are all bold words. Also, digits appearing in upper-case words are presumed to be "upper-case digits":

```
MODE ROW23 = [1:23] INT; LOC ROW23 row23;
```

This regime is introduced by the pragmat **pr** *upper* **pr** (which will usually appear as .PR UPPER .PR since, presumably, UPPER stropping was not in force before the pragmat).

3. RES stropping

RES stands for "reserved word", and in this regime the following 61 words are presumed to be bold whether they are preceded by a point or not (see also the first list in Appendix 1).

> **at, begin, bits, bool, by, bytes, case, channel, char, co, comment, compl, do, elif, else, empty, end, esac, exit, false, fi, file, flex, for, format, from, go, goto, heap, if, in, int, is, isnt, loc, long, mode, nil, od, of, op, ouse, out, par, pr, pragmat, prio, proc, real, ref, sema, short, skip, string, struct, then, to, true, union, void, while.**

Note that this list includes all the delimiter words and all the standard mode-indications – but none of the standard-prelude operators which must still, tberefore, be stropped with a point. Since, for example, LONG is

automatically a bold word in this regime, we have a problem when representing the identifier *long*, or even *long pi* if we are not prepared to represent it as *longpi*. Whereas in the POINT regime we needed the point as an "emboldening" symbol to turn *long* into **long**, we now need an "intimidating" symbol to turn **long** into *long*. For this purpose we use the underscore "_", and it may be placed either after the word to be intimidated, or between two such words. Thus, although END, OF and FILE are all bold words in this regime, END_OF_FILE is a single identifier, since all its words are either preceded or followed by an underscore. For our identifier *long*, then, we write LONG_ and for *long pi* either LONGPI or LONG_PI. Our example now appears as follows:

```
BEGIN REF LONG REAL X := LOC LONG REAL :=
    LONG 3.141592654;X .MINUSAB LONG_PI;
    PRINT(X)
END
```

and the minimum number of disjunctors is shown by

```
BEGIN REF LONG REAL X:=LOC LONG REAL:=LONG 3.141592654
X.MINUSAB LONGPI;PRINT(X)END
```

Of course POINT stropping may still be used (.REAL means the same as REAL and of course .REAL_ is illegal – remember that underscore is not a disjunctor).

This regime is introduced by the pragmat **pr** *res* **pr** (usually appearing as .PR RES .PR).

strings

Within a character- or string-denotation the worthy characters may be used freely to represent themselves with the exception of quote (") and apostrophe ('). These must appear in pairs. Thus """""" is a character-denotation for a single quote-symbol (as already explained in 5.1.1.1) and "''''" is similarly a character-denotation for a single apostrophe-symbol. This is a new feature, and its purpose is to enable a single apostrophe to be used, in some implementations, to escape into some other notation (e.g., in some implementation, 'BS' might represent the otherwise unrepresentable character "backspace" (a character not recognized by the Report) and might appear in string-denotations such as "='BS'/".

If the additional alphabet of lower-case letters is provided then, within character- and string-denotations they are distinct from the upper-case letters

(remember that outside such denotations they are not distinguished, except in UPPER stropping). If some non-worthy characters are available in your implementation (e.g. ~, {, }, etc.), they may be used in character- and string-denotations, but whether such programs could be transferred to another implementation is another matter.

An extra feature, entitled the "string break", is provided to reduce confusions arising when string-denotations occupy more than one line. Remember that two quotes ("") together stand for one quote-symbol. If, however, they are separated by a space, or a change to a new line, then they are ignored altogether:

```
PRINT("This is a very long string-denotation which "
      "occupies more than one line of our program "
      "text, even though we want it to appear as one "
      "line in our output")
```

Possible confusion

Since a point can sometimes mean a point, and sometimes the start of a bold word, is there any situation in which you cannot tell which is meant? It turns out that there is only one situation in legal ALGOL 68 where a genuine point-symbol can be followed by a bold word, and that is in the following format-text:

$ *2zd.* **comment** *format for 3 digit* **real** *number with decimal point but no decimal digits* **comment $**

Suppose we are in UPPER stropping and we write:

```
$ 2ZD.COMMENT FORMAT FOR . . . COMMENT $
```

COMMENT could be a representation of **comment** and so could .COMMENT (POINT stropping is valid in all regimes), so is the point a point or not? The answer is that it is not, because of the general rule [R 9.4.2.2.b] that, in any case of doubt, any sequence of marks is to be regarded as a single symbol wherever such an interpretation is possible. Thus .COMMENT always means **comment**, whether in UPPER or not. Clearly, the format-text in question should have been written as

```
$ 2ZD. COMMENT FORMAT FOR . . . COMMENT $
```

(which is much clearer to the human reader anyway).

APPENDIX 6. Syntax Charts

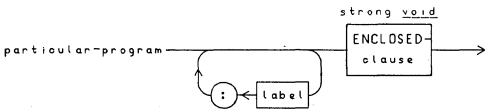
The following charts show exactly which sequences of symbols from a legal ALGOL 68 particular-program and which do not. To see what you may legally write, start where it says "particular-program" in the first chart below, and follow the line. Where the line diverges, you have a choice. You may either write an "ENCLOSED-clause", or you may write a "label" followed by a ":". If you write a label, then you get back where you started so, following the same lines again, you may now write an "ENCLOSED-clause" or you may go for another label. Eventually, you must write an ENCLOSED-clause in order to reach the outgoing arrow on the right, which signifies that your particular-program is complete.

In order to write any construct enclosed in a rectangle (such as an "ENCLOSED-clause"), you must find the start of that construct (usually on another chart) and follow the line from there, writing such constructs as you meet on the way, until you escape via an outgoing arrow. Then you have completed the construct in question and may continue following lines in the original chart. If you encounter a circle (or an oval), simply write the symbol inside it. So, to write an ENCLOSED-clause, find the start on the ENCLOSED-clauses chart. Immediately you are faced with a choice. Suppose you follow the route marked "closed-clause". Now you must write either "**begin**" or "*(*", and after that a "serial-clause" (which is on yet another chart). When your serial-clause is complete, you write "**end**" or "*)*", whereupon you reach the outgoing arrow and your ENCLOSED-clause is complete. Although the chart does not show it (it would have been just too complicated), if you write "**begin**" (rather than "*(*") before the serial-clause, then you must write "**end**" (rather than "*)*") after it, and vice-versa.

Every construct written inside a rectangle will thus be found as an entry point somewhere in one of the charts. The only exceptions are some very simple ones such as "label", "defining-identifier", "field-selector", "mode-indication", "operator", "character" and "digit". The first three of these are the same as "applied-identifier" (on the units chart). For mode-indications and operators see 1.3.2 and 4.3.

Above some of the rectangles there appears an indication of the mode that the construct inside is expected to yield, and the strength of its context (5.1.0.2) or whether it may be balanced (5.2.0.1). The mode written underneath an outgoing arrow tells you the mode of the construct you have just written. "MOID" stands for any mode including **void**, and "MODE" for any

PARTICULAR-PROGRAM

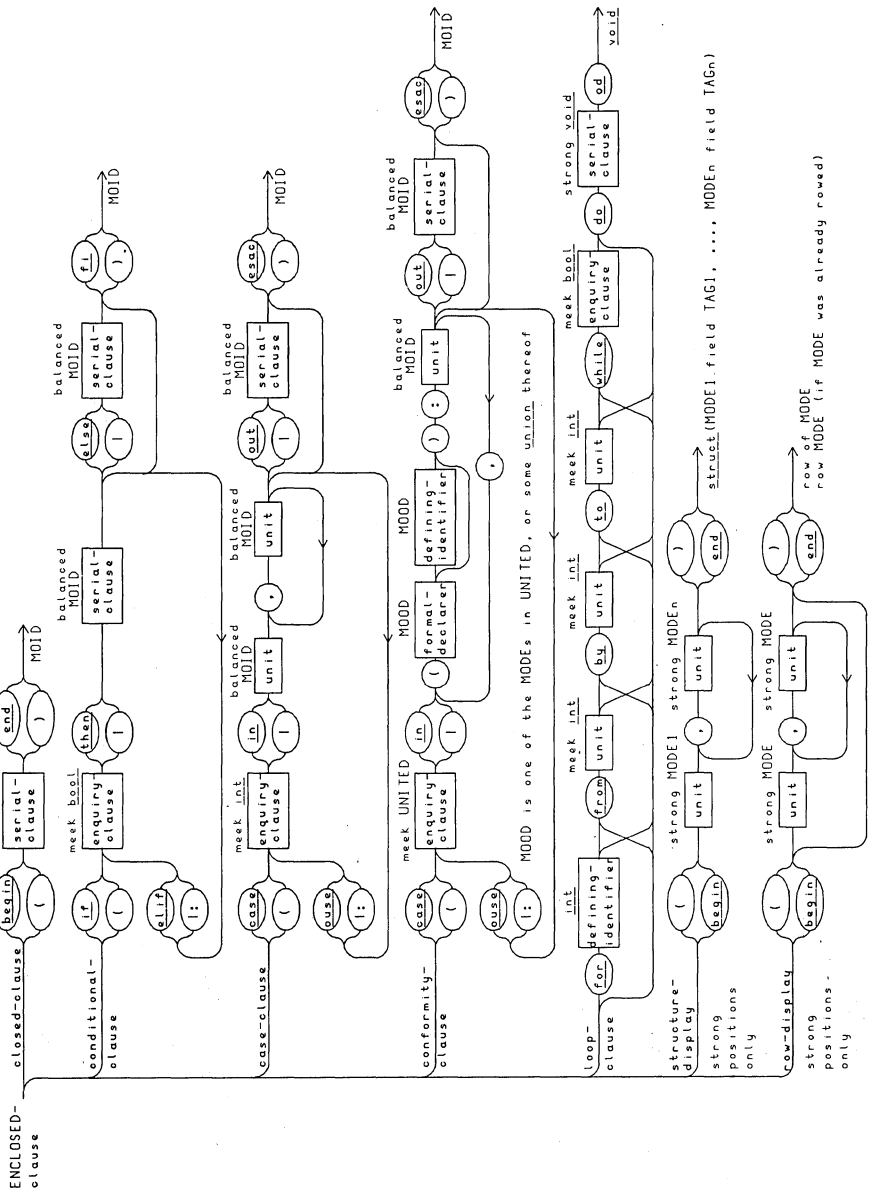


mode other than **void**. On any one pass through a particular chart, the MODEs etc. encountered must, however, always stand for the same mode.

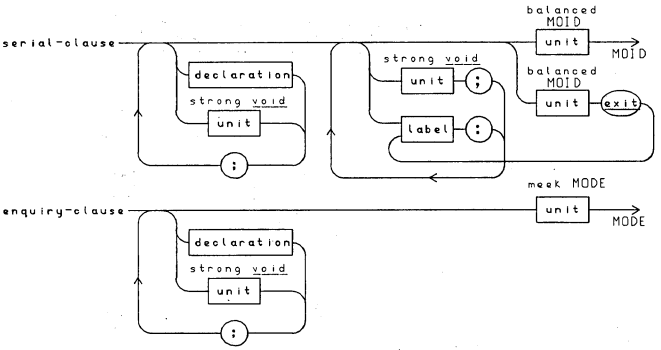
Generally speaking in ALGOL 68, comments and pragmats (1.3.2) may appear in between any two symbols, but there are some exceptions – notably in identifiers, denotations and format-texts. In these charts, you may insert a comment or a pragmat anywhere where you are following a continuous line, but if your route between two symbols is entirely over dotted lines, then you may not write comments or pragmats although blanks and newlines are still permitted (but see 5.5.1.1 for the dangers of doing this in string-denotations and see Appendix 5 for a commonly used solution to the problem).

In "collection-lists" in the format-texts chart, an indication is given of the modes in the data list of *getf* and *putf* which are compatible with the various patterns. For example, the chart shows that for a real-pattern the mode in the data list on output may be **int** or **real**, but than on input it may only be **ref real**. See 7.6.1.3 for further details on this point.

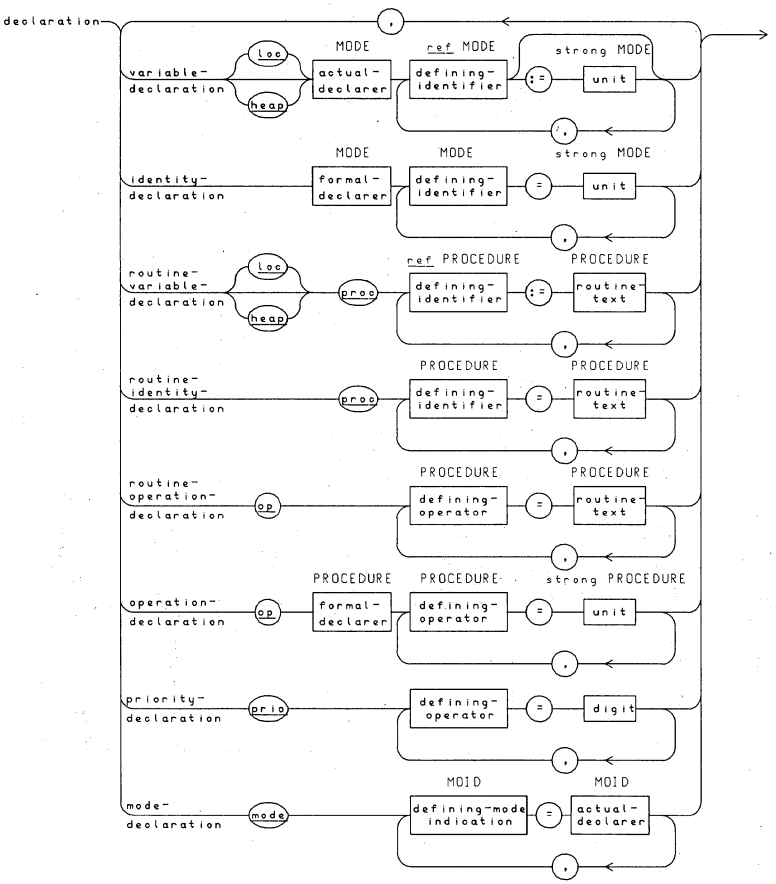
ENCLOSED-CLAUSES



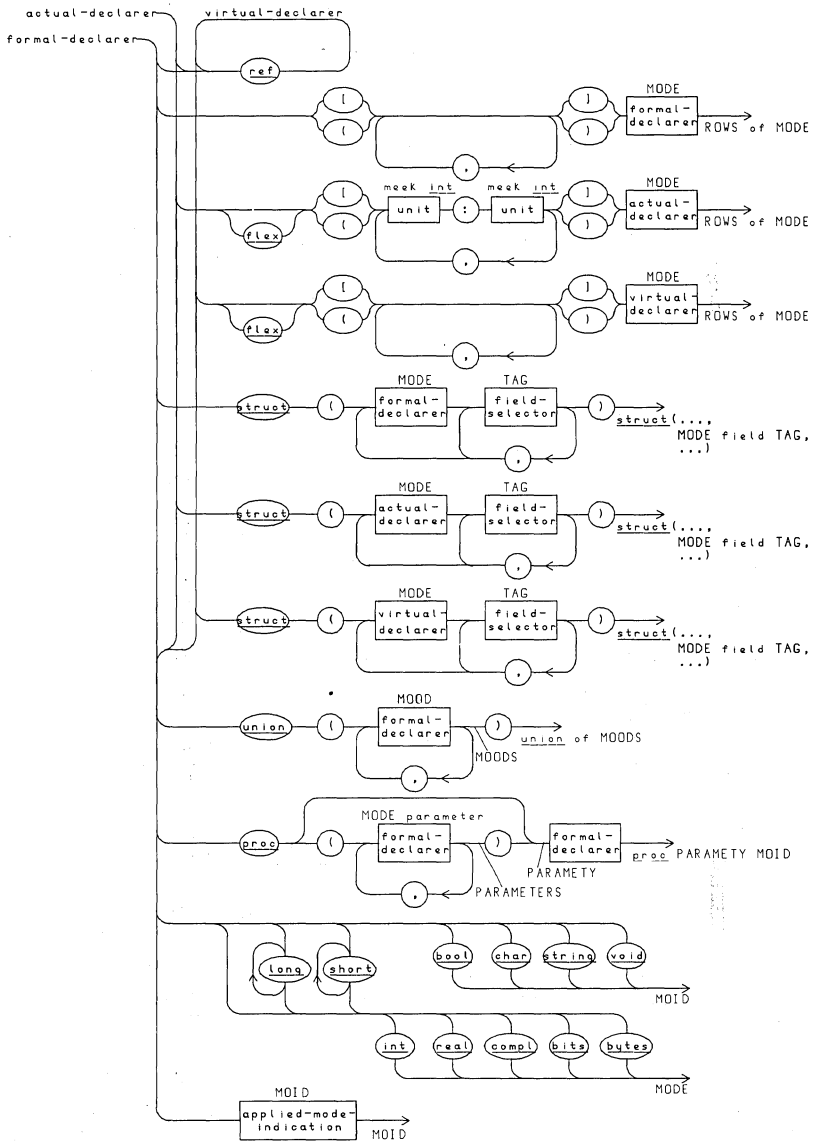
SERIAL-CLAUSES



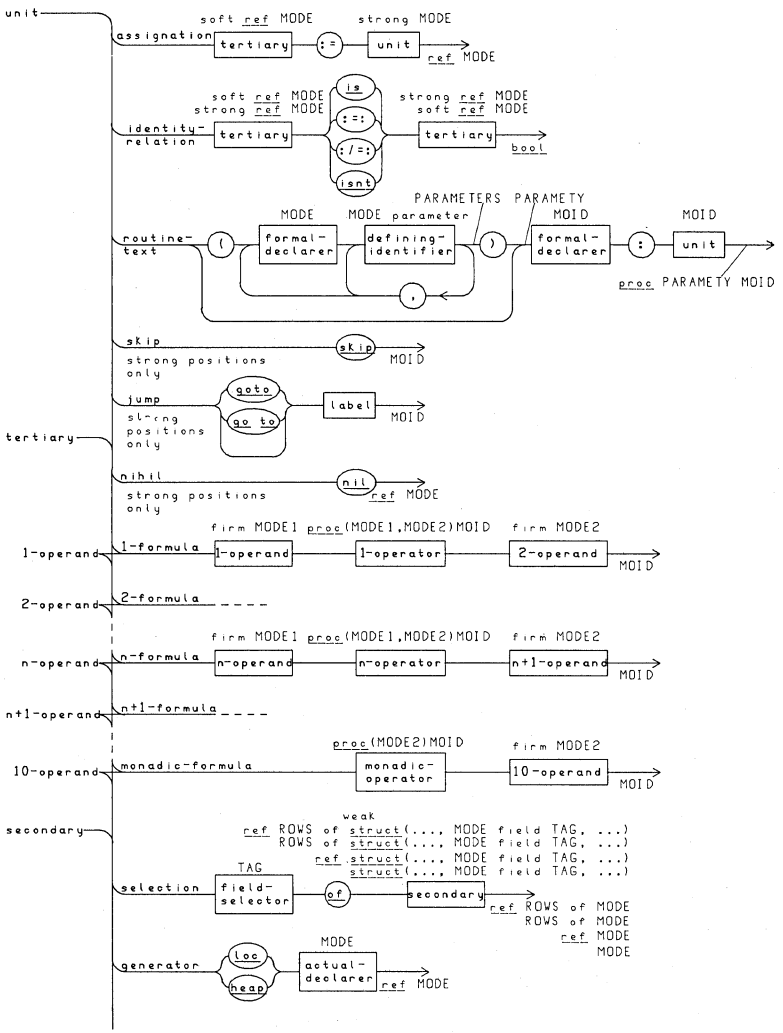
DECLARATIONS

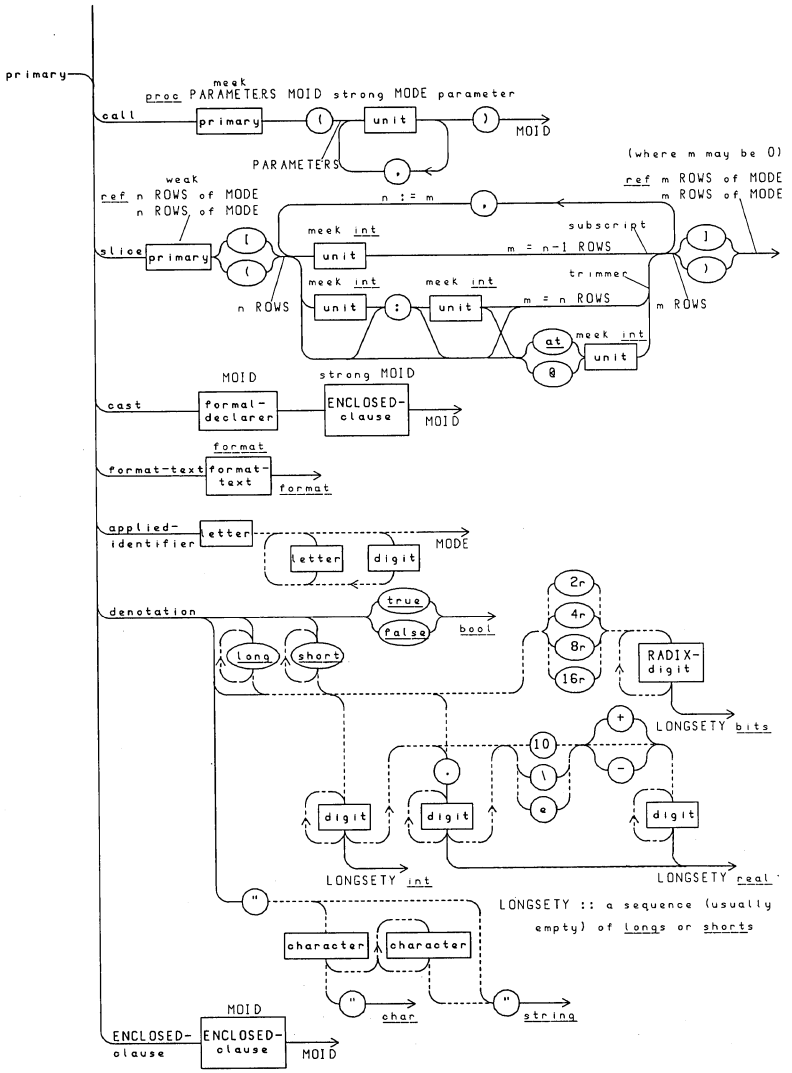


DECLARERS

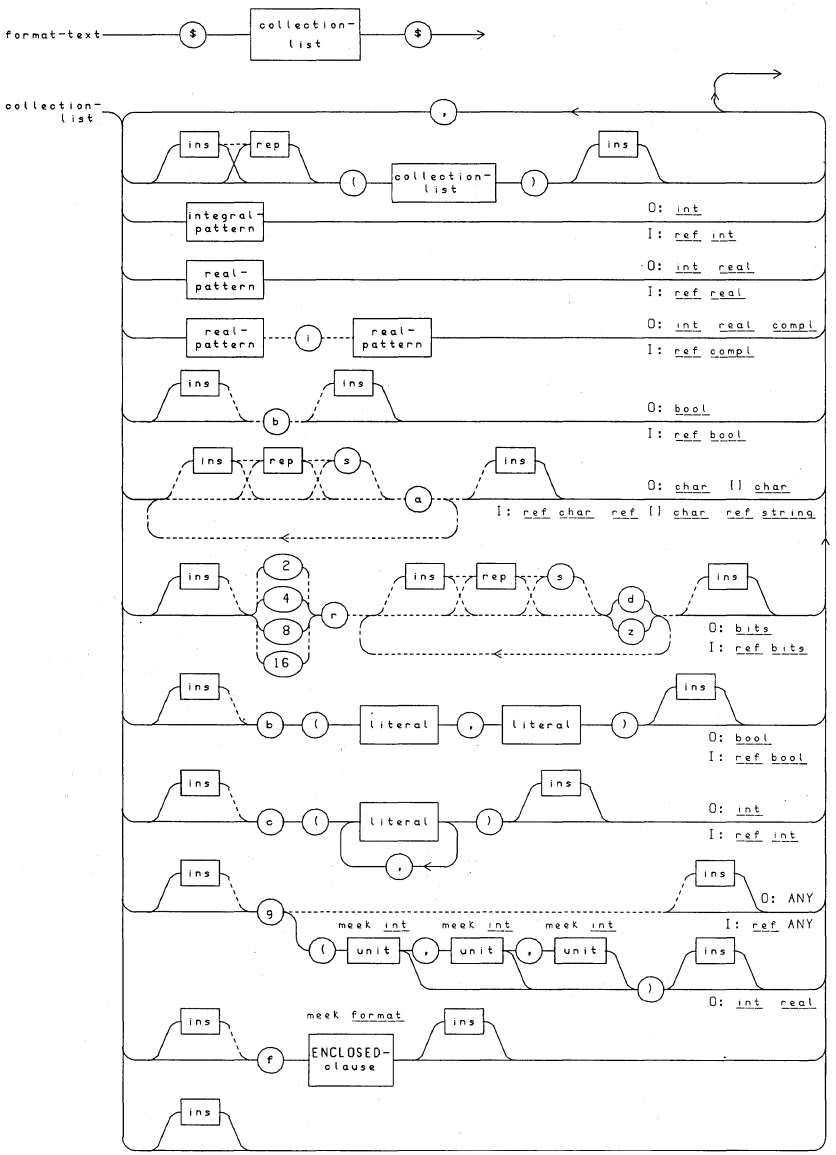

actual-declarer
formal-declarer
virtual-declarer
ref
[
(
,
)
]
MODE
formal-declarer
ROWS of MODE
flex
meek int
unit
:
actual-declarer
virtual-declarer
struct
TAG
field-selector
struct(..., MODE field TAG, ...)
union
MOODS
union of MOODS
proc
MODE parameter
PARAMETERS
PARAMETY
proc PARAMETY MOID
long
short
bool
char
string
void
MOID
int
real
compl
bits
bytes
MODE
applied-mode-indication

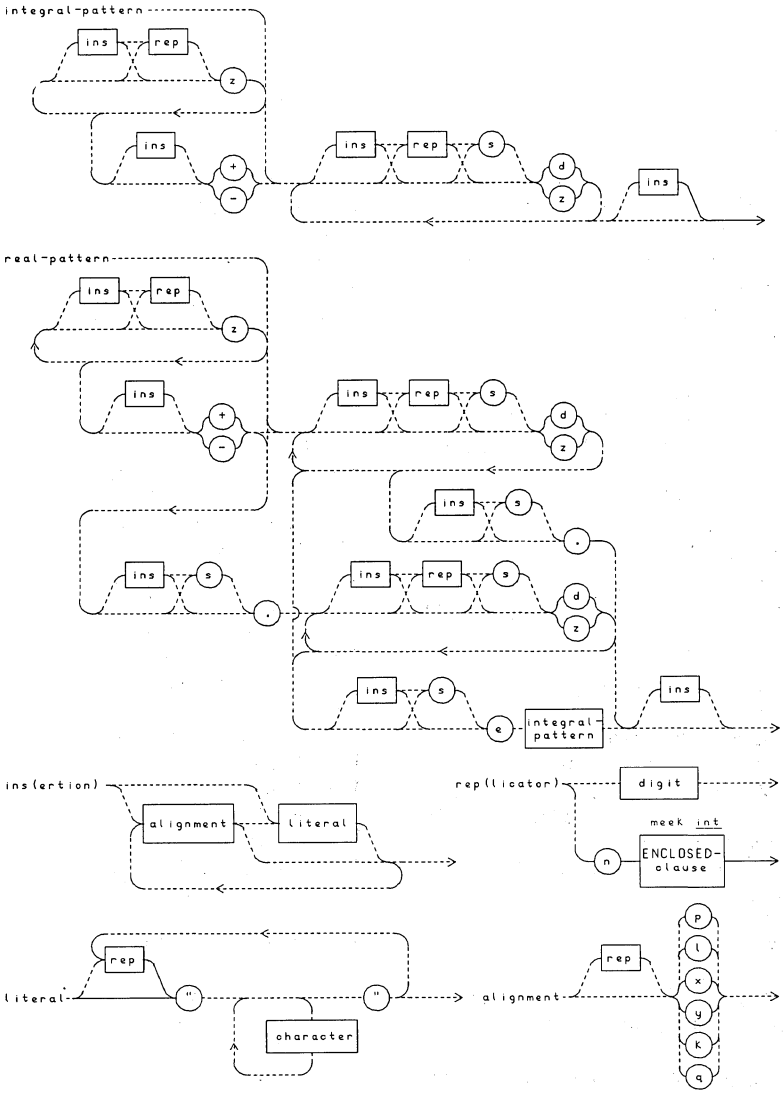
UNITS


unit
assignation
soft ref MODE
strong MODE
tertiary
:=
unit
ref MODE
identity-relation
soft ref MODE
strong ref MODE
is
:=:
:/=:
isnt
strong ref MODE
soft ref MODE
tertiary
bool
routine-text
PARAMETERS
PARAMETY
MODE
MODE parameter
MOID
MOID
(
formal-declarer
defining-identifier
)
formal-declarer
:
unit
,
proc PARAMETY MOID
skip
strong positions only
skip
MOID
jump
strong positions only
goto
go to
label
MOID
tertiary
nihil
strong positions only
nil
ref MODE
firm MODE1 proc(MODE1,MODE2)MOID
firm MODE2
1-operand
1-formula
1-operand
1-operator
2-operand
MOID
2-operand
2-formula
firm MODE1 proc(MODE1,MODE2)MOID
firm MODE2
n-operand
n-formula
n-operand
n-operator
n+1-operand
MOID
n+1-operand
n+1-formula
proc(MODE2)MOID
firm MODE2
10-operand
monadic-formula
monadic-operator
10-operand
MOID
secondary
weak
ref ROWS of struct(..., MODE field TAG, ...)
ROWS of struct(..., MODE field TAG, ...)
ref struct(..., MODE field TAG, ...)
struct(..., MODE field TAG, ...)
selection
TAG
field-selector
of
secondary
ref ROWS of MODE
ROWS of MODE
ref MODE
MODE
generator
MODE
loc
heap
actual-declarer
ref MODE


primary
meek
proc PARAMETERS MOID strong MODE parameter
call
primary
(
unit
)
MOID
PARAMETERS
,
(where m may be 0)
weak
ref n ROWS of MODE
n ROWS of MODE
n := m
ref m ROWS of MODE
m ROWS of MODE
meek int
subscript
slice
primary
[
(
unit
m = n-1 ROWS
]
)
trimmer
n ROWS
meek int
unit
:
meek int
unit
m = n ROWS
m ROWS
at
@
meek int
unit
MOID
strong MOID
cast
formal-declarer
ENCLOSED-clause
MOID
format
format-text
format-text
format
applied-identifier
letter
letter
digit
MODE
denotation
true
false
bool
long
short
2r
4r
8r
16r
RADIX-digit
+
-
LONGSETY bits
10
\
e
.
digit
digit
digit
LONGSETY int
LONGSETY real
"
character
character
"
char
"
string
LONGSETY :: a sequence (usually empty) of longs or shorts
MOID
ENCLOSED-clause
ENCLOSED-clause
MOID

FORMAT-TEXTS




integral-pattern
ins
rep
z
ins
+
-
ins
rep
s
d
z
ins
real-pattern
ins
rep
z
ins
+
-
ins
rep
s
d
z
ins
s
.
ins
s
.
ins
rep
s
d
z
ins
s
e
integral-pattern
ins
ins(ertion)
alignment
literal
rep(licator)
digit
meek int
n
ENCLOSED-clause
literal
rep
"
"
character
alignment
rep
p
l
x
y
k
q

INDEX