Prog Lang
BIB 212797

An

ALGOL 68 COMPANION

J.E.L.Peck

Department of Computer Science

University of British Columbia

Vancouver

Revised Preliminary Edition
March 1972

CONTENTS

Introduction

This book is not intended as a complete description of the language ALGOL 68. That description already exists in the form of the "Report on the Algorithmic Language ALGOL 68", hereinafter referred to as the "Report" and referenced by [R] (see the references). The Report is, of course, a reference document and it must, of necessity, strive for the utmost precision in meaning. Certain sections, therefore, may yield their proper intent only after what the reader may think is an excessive amount of close scrutiny. But then, like any legal statute, the Report should be read carefully, for the authors were determined that, when the reader eventually gropes his way to a meaning in a carefully worded passage, it should yield, beyond all possible doubt, the meaning which was intended, and not some other meaning which the reader may have had in mind. A student of law does not learn the law by first studying the statutes. Likewise, the best approach to a new programming language may not be through its defining document. The law student must be taught how to find his way among the statutes and the student of programming needs to be shown how to get the information he needs from the defining document of a programming language.

Our intention is therefore to introduce the reader, in easy stages, to the ideas and the terminology contained in the Report. Since it is assumed that the Report is always at hand (this book should not be read without it), we absolve ourselves of the necessity for describing every detail of the language. Our purpose will have been fulfilled, if the reader can, after studying this book, put it aside, and from that point onward use the Report alone.

This approach means that it will not be in the interests of the reader to try to explain ALGOL 68 in terms of the concepts used in, say ALGOL 60, or those used in any other programming language. ALGOL 68 has its own new terminology because many of the concepts are new, and though there are similarities with the concepts in other languages, usually the exact counterpart is not available. We shall therefore try to be meticulous about using only the terminology which is employed in the Report; in this way the transition from the Companion to the Report will be easier.

We adopt the same typographical devices as in the Report, whereby examples of the ALGOL 68 representation language are given in italic, e.g., ¤<u>begin</u> print("algol.68") <u>end</u>¤, and notions (i.e., metasyntactic variables, in the sense of ALGOL 60, or nonterminals in the sense of formal grammars) are in a type font which is larger than normal, e.g., •serial-clause•, and usually hyphenated. Experience shows that this practice does not unduly disturb the eye on first reading. It has the advantage that closer examination can reveal whether a word is used in the ordinary sense of the English language or whether it is used in a technical sense. For example, if the reader wishes to know the meaning of "formula", he will look it up in his favourite dictionary; however, to find out about "•formula•" he must look at the rule 8.4.1.a of the Report. This practice will enable us to use words with a precision which would otherwise be difficult to achieve. As with the Report, there are also other words, like "name" or "mode" which are not part of the syntax, but each is given a technical meaning. We shall use quotes, when introducing the reader to these words, to alert him to the fact that he is meeting a new word with a special meaning.

At the end of each chapter is a set of review questions, the answers to which are provided in the final pages. Many of these questions test the material as presented in this text, but others require a deeper study of some parts of the Report. We have tried to provide references to the Report wherever these may be needed.

Some of the earlier chapters of this text were read and corrected by Daniel Berry, Wendy Black, Hellmut Golde, Lambert Meertens, Tad Pinkerton, Helge Scheidig, Aad van Wijngaarden and many others who may forgive the lack of mention here. Their assistance is gratefully acknowledged. Naturally the author is responsible for any remaining imperfections in this preliminary edition. He hopes that readers will communicate with him, thereby helping to eliminate as many errors as possible from the final edition.

The preliminary edition

This preliminary editicn is produced by a text formatting program written by W. Webb at the University of British Columbia for use with the TN print chain. This print chain introduces certain restrictions, some of which are exasperating (e.g., there is no genuine multiplication sign). To simulate the effect of different type fonts, a bracketing scheme is used. ALGOL 68 external objects (program fragments) are represented thus

¤begin real x ; x := 3.14 end¤ ,

ALGOL 68 internal objects (values) are represented thus

▪true▪

and paranotions and modes (syntactic parts) are represented thus

•strong-unitary-real-clause• .

This means that, e.g., a collection of three •identifiers• used for illustration, should be written

¤x¤, ¤a1b2c3¤, ¤an identifier¤ ,

but it will be easier on the eye if we assume that

¤, ¤

may be replaced by

,

so we shall generally use the more pleasing and less cluttered form

¤x, a1b2c3, an identifier¤,

unless the context calls for greater clarity.

The revised preliminary edition

This edition is a reprint of the preliminary edition after correction of some errors and misprints. Another edition is planned for the end of 1972 and may contain additional chapters. The author is grateful to those who sent corrections to the preliminary edition and would appreciate further correction of errors and suggestions for improvement.

1 Denotations

1.1 Language levels

Our purpose is to learn how to read and write ALGOL 68 •programs•. One might suppose that

¤begin real x; x := 3.14 end¤

is an ALGOL 68 •program•, because it is a valid ALGOL 60 •program• and, in a sense, this is the case. However, the similarities between ALGOL 60 and ALGOL 68 begin and end just about here, since

¤myprogram: (print(((real lengths > 1 | "multiple" | "single"), "␣precision␣environment")))¤

is also, in the same sense, an ALGOL 68 •program•. ALGOL 68 is not an extension of ALGOL 60, though the lessons learned in the design and use of ALGOL 60 have contributed to the final shape of the new language. It has, in relation to its contemporaries, a powerful syntactic structure, which enables the defining document of the language to be kept to a minimum. This Companion is an introduction to the language, which should be read only with the defining document, the Report [R], readily at hand. For example, the reader should now turn to the Introduction in the Report [R.0], to get some flavour of the new language.

In ALGOL 68 we may speak of •programs• in the "strict language" and in the "extended language" [R.1.1.1.a]. The strict language is that which agrees with the syntax of the defining document. In a natural language, like English, certain abbreviations, such as "e.g.", are commonly accepted. We usually write "e.g." rather than the longer words "for example", but the meaning is the same. The abbreviations of ALGOL 68, are known as "extensions" [R.9]. The application of these extensions to the strict language yields the extended language. This means that, though •programs• may be written in the extended language, their meaning will be explained in terms of the strict language.

Related to both of these is the "representation language". The first example given above, is a representation [R.3.1.1] of a •particular-program• [R.2.1.d] of ALGOL 68. We say that it is a representation because ¤begin¤ is a representation of the •begin-symbol•, ¤real¤ is a representation of the •real-symbol• and even the point within ¤3.14¤ is a representation of the •point-symbol•. Thus, the example

¤begin real x ; x := 3.14 end¤ ,

(which happens to be written in the extended language), is a representation of the following sequence of symbols

•begin-symbol, real-symbol, letter-x-symbol, go-on-symbol, letter-x-symbol, becomes-symbol, digit-three-symbol, point-symbol, digit-one-symbol, digit-four-symbol, end-symbol•.

We see at once, that it would be too tedious to write •programs• or parts of •programs• without using the representations. Nevertheless, the presence of the strict language, in which all the terminals end in the word •symbol•, will make it easier for us to formulate syntactic rules and to describe and to use the syntax.

1.2 Objects

ALGOL 68 is described in terms of an hypothetical computer which deals with two kinds of "objects"[R.2.2.1]. These are "internal" objects and "external" objects. Roughly speaking, an external object is the sequence of symbols represented by the marks which the programmer makes on his paper when creating a •program•[R.2.1] and an internal object is an arrangement of bits within the computer. For example, when the programmer writes ¤3.14¤, he makes, from four symbols, an external object, which is a •denotation•[R.5]. Within the computer this may be reflected in a certain arrangement of bits, known as a real value, the particular arrangement chosen depending on the kind of computer and the implementer's whim. Thus, ¤3.14¤, which is a sequence of symbols[R.3.1], is an external object and the arrangements of bits is the internal object.

There is an important relationship between external objects and internal objects. One says that an external object may "possess" [R.2.2.2.d] an internal object. Thus, the external object, the •denotation• ¤3.14¤, possesses an internal object which is a collection of bits within the computer. We shall speak of the internal object as a "real value" [R.2.2.3.a]. The form which the internal object takes is of no particular concern to the programmer. It is decided for him by the manufacturer of the computer and by the implementer of the language, i.e., by the compiler writer. In this text we shall represent this by means of a diagram as in figure 1.2, where the internal object

```
¤3.14¤
 ....
    :
    : (2)
    :
 ┌--┴---┐
 |      | (1)
 └------┘
```

Fig. 1.2

is suggested by a rectangle as at 1 and the relationship of possession by the dotted line at 2.

The reader should note that we have introduced, by means of quotes, some standard terminology from the Report[R]. Wherever possible, references to the Report will be given and every effort will be made, in what follows, to remain as close to the Report as possible in the use of this terminology. In this manner the reader may be encouraged to obtain more information about the language by reading the Report itself.

The use of a different type font, such as in •denotation•, indicates that we are talking about an object in ALGOL 68 which is described by the syntax of the language (see paranotions [R.1.1.6.c]). If the same word occurs in normal type font, then an English dictionary should be consulted for its meaning.

1.3 Names

Computers have a storage structure in which the memory is regarded as consisting of small pieces, each usually called a word or byte, with each piece being given a unique address, i.e., a means by which the computer can locate that word or byte. In our hypothetical computer, this situation is modelled by saying that the computer has "names" [R.2.2.3.5], each name[1] referring to some value. When we say that a name "refers" [R.2.2.2.1] to a real value, we are modelling the situation where the real value is an arrangement of bits which is stored at a certain storage place or address. The name is thus the address of the place where the value is stored and the value is the content of that storage place. We have now isolated another kind of internal object, i.e., a "name", and we note that there is a relationship between two internal objects, viz., a name may "refer" to a value. In the diagrams a name will be represented as in figure 1.3 at 1 and the relationship of

```
(1) o                   r-------,
  o o------->-------|       |
    o        (2)        L-------J
```

Fig.1.3

referring by a directed line as at 2. In passing, we mention that a name is also a value [R.2.2.3] and another name may refer to it, but we shall return to this point later.

1.4 Variables

Most programmers do not wish to work only with •denotations• such as □3.14□, but also with •variables• [R.6.0.1.e] such as □x□. In ALGOL 68, as in many other languages, if a programmer wishes to consider □x□ as a variable, he writes a •declaration• [R.7.4.1], e.g., □<u>real</u> x□. The effect of this •declaration• is to allocate a storage place, i.e., to create a name which may refer to a real value, this name being possessed by □x□. In figure 1.4 the relationship of possession

```
  □x□
   :
   :(1)
   :
   o          r-------,
 o o---->-----|       |
   o          L-------J
```

Fig.1.4

is indicated by the dotted line at 1. It is important that this name may not refer to a value of another mode (i.e., to a member of another class of values), such as •boolean• or •character•, for reasons of security in the elaboration [R.1.1.6] of

----

[1] except for •nil• [R.2.2.2.1]

•programs•. In this chapter we are concerned with •denotations•, so we leave the subject of •declarations• and •variables• for the next chapter.

1.5 Denotations

There are four mutually exclusive classes of "plain" values [R.2.2.3.1]. These are, "boolean", "integral", "real" and "character" values. The property of belonging to one of these classes is known as the "mode" [R.2.2.4.1] of the value. A real value is thus said to be of mode •real•. For each of these four classes, i.e., for each of the modes •boolean, integral, real• and •character• we have •denotations•, which are certain sequences of symbols possessing values of that mode. Examples are, □true, 12, 5.67□ and □"w"□. We consider each of these •denotations• in turn.

1.6 Boolean denotations

This is the simplest of the •plain-denotations•. There are two values (internal objects) of mode •boolean•, viz., ▪true▪ and ▪false▪. Consequently we need two external objects to possess them. These are the •true-symbol•, □true□ and the •false-symbol•, □false□. At the risk of tedious repetition, but for further emphasis, we observe that the external object □true□ possesses an internal object, which is the boolean value ▪true▪,

```
    □true□        (external)
     ....
       :
       :
   ┌───┴───┐
   |▪true▪|       (internal)
   └───────┘
```

Fig.1.6

a value of mode •boolean• (see figure 1.6). Of course, a similar statement applies to □false□.

The syntax of •boolean-denotations• is very simple, and supplies a starting point for a study of the syntactic description of the language. This is embodied in the rule [R.5.1.3.1.a]

•boolean denotation : true symbol ; false symbol.• ,

which may be read as "a •boolean-denotation• may be a •true-symbol• or a •false-symbol•".

1.7 Integral denotations

An •integral-denotation•, for example, □34□ or □0□ or □000123□, is a sequence of •digit-tokens•. This means that an •integral-denotation• is easy to recognise and to describe. Its syntax rule [R.5.1.1.1.a] is

•integral denotation : digit token sequence.• ,

which means the same as the rule

integral denotation : digit token ;
integral denotation, digit token. .

The full explanation of how to use this syntactic method of description will be found in Chapter 1 of the Report. It is important that the reader should, at some time, master this syntactic description method. For the moment we may be content to know that this rule describes an •integral-denotation• as a sequence of •digit-tokens•, a •digit-token• being represented by ◻0, 1, 2, 3, 4, 5, 6, 7, 8◻ or ◻9◻. The language makes no restriction on the length of the sequence of •digit-tokens•, although, in a particular implementation, such a restriction may well exist.

An •integral-denotation•, of course, possesses an integral value, as one might expect. Not surprisingly, the value possessed by ◻000123◻ is ▪123▪, which is equal to that possessed by ◻123◻.

1.8 Real denotations

There are two kinds of •real-denotation• [R.5.1.2]. Some examples are: ◻3.14, .000123, 123.45e6, 5e-16, 4.759¹⁰12◻(1). We classify the first two as •variable-point-numerals• and the remaining three as •floating-point-numerals•, the latter being the kind of •real-denotation• likely to be used by the physicist or engineer. This classification is stated [R.5.1.2.1.a] in the rule

•real denotation : variable point numeral ;
floating point numeral.• .

•Variable-point-numerals• have an optional •integral-part•, like ◻123◻, followed by a mandatory •fractional-part• like ◻.14◻ or ◻.000123◻. This is expressed [R.5.1.2.1.b] in the rule

•variable point numeral :
integral part option, fractional part.• .

Examples of •variable-point-numerals• are therefore ◻123.0, 3.456, .12335◻ and ◻.00023◻ but not ◻3.◻. The •integral-part-option• means that the •integral-part• may be present or absent. An explanation of the syntactic device involving the word •option• is to be found in the rule [R.3.0.1.b]

•NOTION option : NOTION ; EMPTY.•

and the fact that any notion may replace the metanotion •NOTION•, but the casual reader need not concern himself yet with these mysteries.

We complete the description of •variable-point-numerals• by the two rules [R.5.1.2.1.c,d]

•integral part : integral denotation.
fractional part : point symbol, integral denotation.• .

Because we have already seen the rule for •integral-denotation• and can guess that the representation of the •point-symbol• is ◻.◻, this syntax should now be clear.

----

(1) A superscript ¹⁰ is used here in place of a subscript 10 which is not available on the TN printer chain.

A •floating-point-numeral• consists of a •stagnant-part•, like □123□ or □123.45□, followed by an •exponent-part•, like □e+23, e2, e-16□ or □^{10}5□. Its syntax is in the rule

•floating-point-numeral : stagnant part, exponent part.• .

Examples of •floating-point-numerals• are therefore, □1e1, 2.3e-4□ and □.3e26□ but not □3.e14□. The •denotation• □.3e26□, for example, possesses a real value, usually associated with the number written in physics textbooks as .3*10^{26}. It could not be so written for computer input because of the inability of most input hardware to accept superscripts. The rule for •stagnant-part• [R.5.1.2.1.f] is

•stagnant part : integral denotation ;
  variable point numeral.• .

Thus both □123□ and □123.45□ are acceptable •stagnant-parts•. The •exponent-part• is described in the rules [R.5.1.2.1.g,h,i,3.0.4.c]

•exponent part : times ten to the power choice, power of ten.
times ten to the power choice :
  times ten to the power symbol ; letter e.
power of ten : plusminus option, integral denotation.
plusminus : plus symbol ; minus symbol.• .

The •times-ten-to-the-power-symbol• is represented by the subscripted ten □^{10}□, but since this is not commonly available, the •letter-e• is also permitted. The •plusminus-option• means that the •plusminus• may be omitted. Examples of •exponent-parts• are □e-5, e4, e+56□ and □^{10}2□.

To review the above, we give some more examples of •real-denotations•: □123.4, .56789, 464.64e-53□ and □987^{10}21□. Note that □123.□ is not a •real-denotation• and there is good reason that it should not be. The explanation is to be found in the representation of the •completion-symbol• [R.3.1.1.f], which is the same as that of the •point-symbol•, so that, were □123.□ permitted, ambiguities would arise. Also, □e15□, for example, is not a •real-denotation• because it might be confused with an •identifier•.

1.9 Character denotations

Some •character-denotations• are [R.5.1.4] □"a", "c", "$", "+", "3"□ and □""""□. All except the last appear easy enough to understand, according to the rule [R.5.1.4.1.a]

•character denotation :
  quote symbol, string item, quote symbol.• ,

provided one can guess the meaning of •string-item• [R.5.1.4.1.b]. However, the •denotation• □""""□ possesses the value which is possessed by the •quote-image•. This value is the character •"•. [R.5.1.4.2.a]. When we come to •string-denotations•, in section 1.11, we shall see that the device whereby the •quote-symbol• within a •character-denotation• is doubled is a convenience which enables every member of the available character set to be in a string.

1.10 Modes

Values within the computer, considered up to now, have been of four kinds, viz., truth values, integers, real numbers and characters. Each member of one of these classes is of the same "mode" [R.2.2.4.1] as any other member of the same class. These modes are •boolean, integral, real• and •character•, respectively. If computing were restricted to these four modes, it would be dull indeed. A useful computer language needs to consider values of other modes. For example, the symbol manipulator often considers values of mode •row of character•, which he thinks of as character strings, and the numerical analyst considers values of mode •row of row of real•, which he thinks of as matrices of real values.

In ALGOL 68, a row of values of one same mode, known as a multiple value [R.2.2.3.3], is also a value of an acceptable mode. Thus, we may have values which are of the mode •row of boolean, row of integral, row of real• or •row of character•. In the diagrams such a multiple value will be represented as in

```
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│      │      │      │      │      │      │      │
└──────┴──────┴──────┴──────┴──────┴──────┴──────┘
```

Fig.1.10

figure 1.10. Many more modes may be considered; in fact, the number of different modes is infinite. We shall not concern ourselves here with this interesting point, nor shall we discuss some of the other modes. Our purpose is to point out that •row of character• is a mode. There are •denotations• for values of this mode and we shall now consider them.

1.11 String denotations

The syntactic rule for •string-denotation• [R.5.3.1.b] is
•row of character denotation : quote symbol,
string item sequence proper option, quote symbol.•
From what has gone before, the reader will surmise that the following are examples of •string-denotations•: □"abc", "a+b", "this.is.a.quote.symbol."".""□. Observe that in the strict language, the representation of the •space-symbol• is □.□ [R.3.1.1.b]. The only feature in the above syntax, which we have

```
                □"abc"□
                 .....
                     :
                     :
        ..........:
        :
        ┌─────┬─────┬─────┐
        │ ▪a▪ │ ▪b▪ │ ▪c▪ │
        └─────┴─────┴─────┘
```

Fig.1.11

not yet encountered, is the use of the word •proper•. The exact explanation is to be found in the rule •NOTION LIST proper : NOTION, LIST separator, NOTION LIST.• [R.3.0.1.g]. It means that the sequence must contain at least two members. The use of the combination •proper option•, means then, that the sequence may contain either zero or two or more members. This implies that ¤"a"¤ is not a •string-denotation•, but that ¤""¤ is. Since we have already seen that ¤"a"¤ is a •character-denotation•, we can understand the reason for such an unusual choice of syntax. A •string-denotation• possesses a value which is of mode •row of character•. Our diagrams may represent it as in figure 1.11. The value possessed by ¤""¤ is a row of characters with no elements.

1.12 Other denotations

This discussion does not exhaust the •denotations• of ALGOL 68, but it is sufficient for us to go on to other elementary parts of the language. We shall return later to •long-integral-denotations• like ¤long 0¤ [R.5.1.0.1.b], •long-real-denotations• like ¤long .1¤, •bits-denotations• like ¤101¤ [R.5.2.1], •routine-denotations• like ¤((real a,b) real : (a > b | a | b))¤ [R.5.4] and •format-denotations• like ¤$16x3zd$¤ [R.5.5].

1.13 Program example

Though we are not yet ready to write •programs•, it is helpful to inspect one and perhaps therefrom to glean some ideas. The following will read some number of values from the standard input file and then print a count of the number, the arithmetic mean of the values and their standard deviation. Comments are enclosed by the symbol ¢ or the symbol #.

```
¤begin real s := 0 ¢for the sum of the values¢,
     ss := 0 ¢for the sum of squares¢,
     x ¢the current value¢;
 int n := 0 ¢for a count of the number of values¢;
 while ¬ logical file ended(standin) do
     ( get(standin, x) ¢R.10.5.2.2.b¢;
      s +:= x ; ss +:= x ** 2 ; n +:= 1 ¢R.10.2.11.d,e¢);
 put(standout, ¢R.10.5.2.1.b¢ ("count.=.",n,
        "..mean.=.",s / n,
        "..standard.deviation.=.",
          sqrt((ss - s ** 2 / n) / n) ¢R.10.3.b¢))
 end¤
```

Points of relevance to this chapter are that there are four •variables• ¤s, ss, x¤ and ¤n¤, some of which are initialized with the value zero. Also, the •integral-denotation• ¤0¤ occurs three times and the •integral-denotation• ¤1¤, once. There are three •row-of-character-denotations•. References to the Report are provided as explanation of other points to be covered in later chapters.

Review questions

1.1 Language levels

a) How does one recognize a terminal [R.1.1.2.f] in the syntax of ALGOL 68?
b) Are there two or three symbols of which the colon, □:□, is a representation[R.3.1.1]?
c) Are there any other representations which represent more than one •symbol• [R.3.1.1]?
d) Is the mark "(" a representation of a •sub-symbol• or of an •open-symbol• or of both [R.3.1.1, 9.2.g]?

1.2 Objects

a) What kind of object is possessed by the •denotation• □3.14□ [R.2.2.2.d]?
b) What object may possess a real value?
c) Is □3.14□ an internal object or an external object?
d) Does □**true**□ possess •true• or does •true• possess □**true**□?

1.3 Names

a) Can a real value refer to a name [R.2.2.3.5]?
b) Can a name refer to a name?
c) Is a name an external object?
d) Can an external object possess more than one name?
e) Does an external object always possess a name?

1.4 Variables

a) In the reach [R.4.4.2.a] of □**real** x□, can the name possessed by □x□ refer to an integral value?
b) May □**real** x, y, z□ be a •declaration• [R.9.2.c]?

1.5 Denotations

a) How many classes of plain values are there [R.2.2.3.1]?
b) Is there a class of plain values with finitely many members?
c) What distinguishes classes of values [R.2.2.4.1.a]?

1.6 Boolean denotations

a) In the syntax, how should the syntactic marks ":", ";" and "," be interpreted [R.1.1.4]?
b) Is •true• an internal object?

1.7 Integral denotations

a) Can two •integral-denotations• possess equal values?
b) Is □-123□ an •integral-denotation• [R.5.1.1.1]?
c) Can a sequence of one thousand digits be an •integral-denotation•?
d) Does every •integral-denotation• possess a value [R.5.1.0.2.b]?

1.8 Real denotations

a) Can two different •real-denotations• possess equal values?
b) Is ¤1.¤ a •real-denotation•?
c) Is ¤12¤ a •real-denotation•?
d) Is ¤12e-4¤ a •real-denotation•?
e) Is ¤-12e4¤ a •real-denotation•?

1.9 Character denotations

a) Is ¤""""¤ a •character-denotation•?
b) Does every •string-item• possess a character [R.5.1.4.2]?

1.10 Modes

a) How many different modes are there?
b) How many different modes can a programmer specify?

1.11 String denotations

a) Is ¤""""""¤ a •string-denotation•?
b) Is ¤""¤ a •string-denotation•?
c) What is the mode of the value possessed by a •string-denotation•?

1.12 Other denotations

a) Are the values possessed by ¤long 0¤ and ¤long long 0¤ the same?
b) What is the mode of the value possessed by ¤101¤ [R.5.2]?
c) What is the mode of the value possessed by ¤$16x3zd$¤?

1.13 Program example

a) What is the mode of the value possessed by "count.=."?
b) What are the modes of ¤s¤ and ¤n¤?
c) Does the example in 1.13 contain a •real-denotation•?
d) How many •integral-denotations• are there in the example?
e) Does the example contain a •character-denotation•?

2 Some fundamental concepts

2.1 Declarers

In chapter 1 we found that each value within the computer is of a certain mode. (There is an exception, viz., the value •nil• [R.2.2.3.5.a], but we shall discuss this exception later.) Thus, there are values of •integral• mode, •real• mode, •character• mode, •row-of-character• mode, and so on. The programmer needs to have some way of specifying modes, because when creating •variables• [R.6.0.1.e] he must help the computer to decide how much storage to allocate. The programmer specifies the modes by using •declarers• [R.7.1].

There are five primitive [R.1.2.2.a] •declarers•. These are ¤int¤, which specifies the mode •integral•; ¤real¤, which specifies the mode •real•; ¤bool¤, which specifies the mode •boolean•; ¤char¤, which specifies the mode •character• and ¤format¤, which specifies the mode •format• (of which we shall hear more later). The mode of a •real-variable•, however, is •reference to real• and not •real•. This mode is specified by the •declarer• ¤ref real¤. A •declarer• specifying the mode •row-of-real• is ¤[ ]real¤, or if actual bounds are required, then say, ¤[1:10]real¤. The mode of a real vector variable is •reference to row of real• and this mode is specified by a declarer like ¤ref[ ]real¤ or ¤ref[1:n]real¤. We see, therefore, that other •declarers• may be built from the primitives by using the symbols ¤ref¤ for •reference-to• and ¤[ ]¤ for •row-of•. Other possible prefixes are ¤proc, struct¤ and ¤union¤ but these may also involve the use of the symbols ¤(¤ and ¤)¤.

This is not a full description of •declarers•, but enough for our present purpose. As a taste of what other •declarers• are possible, we list a few examples:

¤ref ref real, [1:0 flex]char, proc(real)real, [1:n]format, proc, struct(real re, im), union(real, int, bool)¤.

2.2 Generators

At the heart of ALGOL 68 is the notion •generator• [R.8.5.1]. There are two kinds of •generators•, •local-generator• and •global-generator• [R.8.5.1.1.a]. Syntactically, a •local-generator• is a •local-symbol•, ¤loc¤, followed by a •declarer•, e.g., ¤loc int¤. A •global-generator• is an optional •heap-symbol•, ¤heap¤, followed by a •declarer•, e.g., ¤heap real¤ or ¤real¤. The difference in semantics concerns the method of storage allocation and particularly of storage retrieval. The inexperienced programmer is unlikely to make explicit use of •generators•, but •local-generators• appear implicitly in some frequently used •declarations•, so we shall introduce them now.

2.3 Local generators.

The syntactic rule for •local-generator• might be written informally as:

local generator : local symbol, actual declarer.

but the strict syntactic rule [R.8.5.1.1.b], in common with many other rules, contains a feature which the reader should now observe. The rule is

•reference to MODE local generator :
  local symbol, actual MODE declarer.• .

The feature to be noticed is the occurrence of the "metanotion" •MODE•, both to the left and to the right of the colon in the rule. A full description of this two-level syntax is contained in the Report [R.1.1]. For the moment we may be content with the explanation that the use of this metanotion is a device whereby several rules of the language may be combined into one. If we replace, consistently throughout the rule, the metanotion •MODE• by a mode (one of the terminal productions [R.1.1.3.f] of •MODE• like •integral• or •real•), then we obtain a rule of the strict language. For example, if we replace •MODE• by •real•, we obtain the production rule

•reference to real local generator :
  local symbol, actual real declarer.• .

If we replace it by •boolean•, we obtain the rule

•reference to boolean local generator :
  local symbol, actual boolean declarer.• .

This device, in this rule, enables the syntax to tell us something about the relationship between the mode of a •generator• and the mode of its •declarer•. Specifically, the mode of a •generator• is always •reference to• followed by the mode of its •declarer•. In the example of the •local-generator• □loc real□, its declarer, □real□, specifies the mode •real•, but the generator, after its elaboration, possesses a value (a name) of mode •reference to real•; but this is the subject matter of the next section.

2.4 The elaboration of a generator

The "elaboration" of a •program• consists of a sequence of actions performed by the hypothetical computer. These actions are explained in the sections, headed Semantics, in the Report. We shall now examine the effect of the elaboration of a •generator• [R.8.5.1.2]. A •generator• creates a name, i.e., it allocates computer storage. This name then refers to some value. This process is so fundamental to the understanding of the

```
(external)□loc real□(5)
                 ........
 - - - - - -     :  - - - -
                 : (4)
(internal)       : (possess)
                 :
                 o(1)       (3)      ┌───────┐(2)
               o o--------->-----┤       │
                 o  (refer to)      └───────┘
```

Fig.2.4.a

language, that we will attempt to make it clear by means of a diagram. We may picture the elaboration of the •generator• □loc real□, as in figure 2.4.a. In this figure, the name is at 1, the value to which it refers at 2, the relationship of reference at 3, the relationship of possession at 4 and the external object at 5. The broken line then separates the external object from the two internal objects. The elaboration of the •local-generator•, □loc real□, thus creates a name which refers to some real value. The external object, □loc real□, is then made to possess the name. This last action is thus pictured at 4. The value referred to is some undefined real value. We shall see later that this value may be changed ("superseded" [R.8.3.1.2.a]) by "assignment".

2.5 Identity declarations

•Generators• may occur in more than one context, but the most important context is the •identity-declaration• [R.7.4.1]. We give first an example of an easy •identity-declaration• containing no •generator•,

□int m = 4096□ .

The effect of the elaboration of an •identity-declaration• is to make two different external objects possess the same internal object. In the example at hand, we have an •integral-mode-identifier•, □m□, and an •integral-denotation•, □4096□. We have seen in chapter 1, that □4096□ possesses an internal object, which is an integral value. This situation may be pictured,

```
    □int m = 4096□                          □int m = 4096□
              ....                            .      ....
               :                              :       :
               :                              :       :
            r--┴----┐                    r------┤    r--┴----┐
            |•4096•|                     |•4096•|    |•4096•|
            └------┘                     └------┘    └------┘

       Fig.2.5.a                              Fig.2.5.b
```

before the elaboration of the •identity-declaration•, as in figure 2.5.a. After the elaboration of the declaration, □int m = 4096□, the situation is as in figure 2.5.b, where □m□ now possesses a new instance of the same integral value as that possessed by □4096□. It is important to note that □m□ does not possess a name and, as a result, □m□ may not appear as the •destination• of an •assignation•, as for example in □m := 0□. In fact, □m := 0□ would be just as improper as □4096 := 0□. The •identifier• □m□ is thus a •constant• [R.6.0.1.d].

Of greater interest is the declaration of a •variable•, of which

□ref real x = loc real□

is an example. As we have seen already in section 6.4, the programmer is permitted to write this in the extended form

□real x□

[R.9.2.a]. The first step in the elaboration of this •identity-declaration• is the elaboration of its •actual-parameter•, which is □loc real□. We have seen, in 2.4, that this will make □loc real□ possess a name which refers to some (undefined) real value. This stage is pictured in figure 2.5.c. After the elaboration of the •declaration•, the •reference-to-real-identifier• □x□ possesses the same value as that possessed by □loc real□. The result, in pictorial form, is shown in figure 2.5.d. Here, because □x□ now possesses a name, it may be used as the •destination• of an •assignation•, i.e., the value to which the name refers may be superseded [R.8.3.1.2.a] by another value

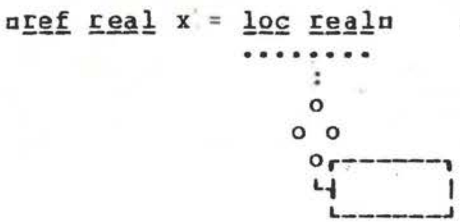


Fig.2.5.c

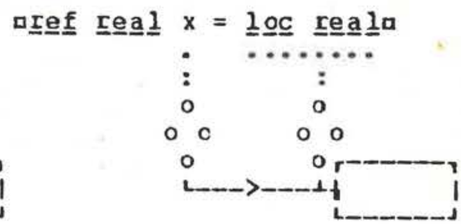


Fig.2.5.d

(provided that it is of mode •real•). When examining diagrams, such as the one in figure 2.5.c and d, we should keep in mind the fact that the name possessed by an •identifier•, which is a •variable•, is unlikely to be a piece of storage set aside in the data area. It is rather the value to which this name refers which may be in the data area. The name itself is more likely to be part of a machine code instruction. Since programs are not usually permitted to alter their own coded instructions, it is essential that the relationship of possession should not be violated. Thus the name possessed is never changed. If we want to reach down to the data area, then we must make use of the name in order to find that part of the data area to which it refers and which can be changed (superseded).

The possession of a name confers a special privilege. It is as though the name is the key to a storage cell without which it may not be unlocked. When it is unlocked, the content may be changed, but without this key, i.e., without the name, the content of that cell may not be changed, though it may be examined, as if through a window.

To recapitulate then, the elaboration of an •identity-declaration• makes its •identifier• possess the same value as that possessed by its •actual-parameter•. This is what occurred in both of the examples □int m = 4096□ and □ref real x = loc real□.

2.6 The syntax of identity declarations

We are perhaps getting a little ahead of ourselves, since we have not yet examined the syntax of •identity-declarations•. This might be described informally by

  identity declaration :
    formal parameter, equals symbol, actual parameter.

but the rule in the Report [R.7.4.1.a] is

  •identity declaration : formal MODE parameter,
    equals symbol, actual MODE parameter.•

We see here again the use of the metanotion •MODE•, which enables one to condense many rules into one. The metanotion must be replaced consistently by one of its terminal productions [R.1.1.5.a], e.g., by •integral• or •reference to real•. Using the latter replacement, we obtain the production rule [R.1.1.2.c]

•identity declaration : formal reference to real parameter, equals symbol, actual reference to real parameter.•

Two of the notions in this rule envelop [R.1.1.6.j] the mode •reference to real•. In the •declaration• □ref real x = loc real□, the mode of the •generator• □loc real□ is •reference to real• and that of the •formal-parameter• □ref real x□ is also •reference to real•. It follows from the rule on •formal-parameters• [R.5.4.1.e], that □x□ is then a •reference-to-real-mode-identifier•.

2.7 Formal parameters

We must follow this a little further by examining the rule for •formal-parameters• [R.5.4.1.e] which is

•formal MODE parameter :
formal MODE declarer, MODE mode identifier.•

and in which the metanotion •MODE• appears three times. By substitution we obtain the rule applicable to the •formal-parameter• □ref real x□, viz.,

•formal reference to real parameter :
formal reference to real declarer,
reference to real mode identifier.•

The •formal-reference-to-real-declarer• is □ref real□ and the •reference-to-real-mode-identifier• is □x□ [R.4.2.2].

2.8 An extension

The object

□ref real x = loc real□

is a representation of a •declaration• in the strict language. Although, as we have seen above, it enables one to explain the meaning of the •identity-declaration• clearly, it is rather much to write and would certainly not be popular with programmers. A similar situation exists with the elisions of a natural language. It is well known that the sentence "Who's that?", stands for the sentence "Who is that?", and that the former is used more often than the latter. Moreover, in explaining the meaning of the first sentence, we always use the second, strict form. Similarly in ALGOL 68 we may write

□real x□

to stand for

□ref real x = loc real□

with the assurance that the meaning is the same [R.9.2.a]. The

```
             r----->------┐
       (1)   |            V
□ref real x = loc real□
 XXXXXXXX   XXXXX    (2)
```

Fig.2.8

effect of this extension [R.1.1.7] (one must resist the temptation to call it a contraction) is that one may omit those parts which are underlined with X's in figure 2.8. and then move the •identifier• in the manner indicated (provided that the following symbol is □,□, □;□ or □:=□). It is important to note that in the extended •declaration• □real x□, the •formal-declarer• □ref real□ (see figure 2.8 at 1) is omitted, but the •actual-declarer• □real□ (see figure at 2) from the •generator• remains. This is of significance when the •declarers• are for multiple values.

Another extension, which we mention in passing, is that, e.g., □real x, real y□ may be written □real x, y□ [R.9.2.c].

In the examples which follow, the •declarations• □real x, y, int i, j, n, [1:10]real x1, y1□ will always be assumed. Thus, unless contradicted by another •declaration•, □x□ and □y□ will have the mode •reference to real•, □i, j□ and □n□ the mode •reference to integral• and □x1□ and □y1□ the mode •reference to row of real•.

2.9 An assignation

We have seen before that a name is, as it were, a key with which to unlock the value to which it refers. This key is needed when an assignment is made. An external object of the form

□x := 3.14□

(in the reach of the •declaration• □real x□), is an •assignation• [R.8.3.1] and its elaboration involves an assignment [R.8.3.1.2.b]. It consists of a •destination•, which is □x□, a •source•, which is □3.14□, and between the two a •becomes-symbol•, □:=□. First, both the •source• and the •destination• are elaborated in unspecified order, or "collaterally" [R.6.2.2.a] (see figure 2.9 at 1), i.e., we obtain the values possessed by them. The effect of the

```
......................reference-to-real-assignation
:                                |
:               ┌────────────────┴───┬────────────────┐
:               |                    |                |
:  reference-to-real-destination becomes-symbol real-source
:               |                    |                |
:               □x                   :=             3.14□
:(3)            :                                     :
:               :(1)                                  :(1)
:               o      ┌──────┐                   ┌───┴──┐
:..............o o->-┤      |==========<=========|      |
                o      └──────┘        (2)        └──────┘
```

Fig.2.9

•assignation• is the assignment of the value possessed by □3.14□ to the name possessed by □x□ (see figure 2.9 at 2). More precisely, the name possessed by □x□ is made to refer to a copy (new instance) of the value possessed by □3.14□ [R.8.3.1.2.c,d]. An •assignation•, after its elaboration, possesses a value and the value possessed is that of its •destination•, which is a name (see figure at 3).

2.10 The syntax of assignations

We should now examine the syntax of •assignations•, in particular, the rule

•reference to MODE assignation :
reference to MODE destination, becomes symbol, MODE source.•

[R.8.3.1.1.a]. Remembering that the metanotion •MODE• should be replaced consistently by some mode, we replace it by •real• and obtain the rule

•reference to real assignation :
reference to real destination,
becomes symbol, real source.•

The important point to notice about this rule, which is the rule governing the object ¤x := 3.14¤, is the fact that the mode enveloped by the •destination• is •reference to real•, while the mode enveloped by the •source• is •real•. We see therefore, the requirement that the •destination• must possess a name, while the •source• need not. Moreover the mode of the •destination• is always •reference-to• followed by the mode of the •source•. Finally, we note that the mode of the •assignation• itself, is the same as that of the •destination•, as might be expected from the discussion in the last paragraph.

We may now examine the construction

¤<u>int</u> m = 4096 ; m := 4095¤

and decide that ¤m := 4095¤ cannot be an •assignation•, because ¤m¤ does not possess a name, i.e., its mode does not begin with •reference-to•. In fact, the mode of ¤m¤ is •integral•. We are therefore justified in using the term •constant• [R.6.0.1.d] for the •identifier• ¤m¤.

2.11 References

These subtle distinctions between •constants• and •variables•, the insistence on the difference in mode provided by •reference-to• and the distinction between those values which are names and those which are not, may seem a high price to pay for the understanding of a programming language. Nevertheless, it is at the very heart of ALGOL 68 and should be understood well before proceeding further. Moreover, we shall find later that it pays a handsome dividend in chapter 5 when explaining the parameter mechanism in •calls• [R.8.6.2.2] of routines. Some readers may be a little baffled and impatient for the reason that many well known programming languages[1] appear either not to make this distinction or to consider it of no importance. Even mathematicians (but perhaps not logicians) are guilty of slurring over the differences in meaning between ¤2.3 + 4.5¤ and ¤x + y¤. Ingrained habits of thought are difficult to dislodge and it is not easy for us to suppress our ire while acknowledging that we have not properly understood something

----

[1] Except for the languages LISP, SNOBOL and TRAC.

elementary. We pursue this point a little further in our next paragraph.

2.12 Dereferencing

If □x := 3.14□ is an •assignation•, then surely □x := y□ (in the reach of the declaration □real y□) must be also. However, the mode of □x□ and that of □y□ is •reference to real•, while an •assignation• requires that the mode of the •destination• should be •reference to• followed by the mode of the •source•. This means that the mode of □y□ should be •real•. It would seem then, that this object does not fit immediately into the syntax of •assignations•. However, it is an •assignation•. Diagrammatically, the situation is shown in figure 2.12. The first step is the elaboration of the •source• and the •destination• collaterally [R.6.2.2.a] (figure 2.12 at 1,2,3 and 4). However, the •source•, in this object, requires an extra step in its elaboration. Since □y□ possesses a name (figure 2.12 at 2) referring to a real value, this name is "dereferenced" (figure 2.12 at 3), i.e., the value to which it

```
...........reference-to-real-assignation
:                                |
:             -------------------------------------------
:             |               |                         |
:  reference-to-          becomes-               real-source
: real-destination        symbol                        |
:             |               |                         |
:             |               |       (4).........real-base
:             |               |         :               | (3)
:(6)          |               |         : reference-to-real-base
:             |               |         :               |
:            □x               :=        :              y□
:             :(1)                      :               :(2)
:             o          --------        --------       o
:...........o o->-|        |===<==|        |-<--o o
              o          --------   (5)  --------       o
```

Fig.2.12

refers is yielded (figure 2.12 at 4). The act of dereferencing is known as a "coercion", of which we shall hear much more later [R.8.2]. There is thus an intermediate step during which □y□ , as a •source•, possesses a real number. This moment is pictured in figure 2.12 at 4. From this intermediate situation we are now ready to make the assignment (figure 2.12 at 5). The value of the •assignation• is a name of mode •reference to real• (see the figure at 6).

The syntactic analysis of the •assignation•, □x := y□, is not trivial and we are not ready to do it , though we have sketched it roughly in figure 2.12. The main point is to determine how □y□, which is of a priori mode •reference to real•, can be considered, a posteriori, of mode •real• (see the figure at 3). The crucial step is contained in the production rule

•strongly dereferenced to real base : reference to real base.• which is obtained from 8.2.1.1.a of the Report by suitable replacements of the metanotions. We do not intend to go into further detail here, for coercion is the topic of chapter 6. Our purpose is to affirm that ¤x := y¤ is indeed an •assignation• even though the a priori mode of ¤y¤ is not •real•.

The reader may wish to persuade himself, from what has gone before, that ¤x := y := 3.14¤ is also an •assignation•, and has a different meaning from that of the, rather foolish, •assignation• ¤(x := y) := 3.14¤.

2.13 Initialized declarations

The •actual-parameter• of an •identity-declaration• may also be an •assignation•. The pertinent rules are, in simplified form,

```
    actual parameter : unit ; ... .              R.7.4.1.b
    unit : unitary clause .                      R.6.1.1.e
    unitary clause : ... ; confrontation ; ... .
                                                 R.8.1.1.a, 8.2.0.d
    confrontation : assignation : ... .          R.8.3.0.1.a
```

Since ¤loc real := 3.14¤ is an •assignation•, this means that ¤ref real x = loc real := 3.14¤ is an •identity-declaration•. But we have seen that the object ¤ref real x = loc real¤ may be written ¤real x¤ [R.9.2.a]. This means that ¤real x := 3.14¤ is also an •identity-declaration• with the same meaning as that of ¤ref real x = loc real := 3.14¤. This meaning should now be evident once it is realized that the •assignation•, being the •actual-parameter•, is elaborated before the final step of the elaboration of the •identity-declaration•. ALGOL 68 may thus be considered as a language which contains initialized •declarations•, although the defining Report does not mention them.

2.14 Program example

The following •particular-program• computes the components (principal and interest) of the monthly repayments of a loan. It first reads the principal, ¤p¤, the interest rate per unit per year, ¤r¤, the number of times per year that the interest is converted, ¤t¤, the constant monthly payment, ¤mp¤ and the number of years, ¤y¤. It then prints an echo of the input, followed by a table of four columns consisting of the month number, the principal outstanding at the end of the month, the component of the monthly payment which is principal and that which is interest. A separate computation is made for the final monthly payment. Critical computations are made using values of mode •long-real•.

```
¤begin long real p ¢the principal¢,
                 r ¢the interest rate per unit per year¢,
                 mp ¢the constant monthly payment¢,
int t ¢the number of times per year that the interest is
    converted¢, y ¢the number of years¢ ;
start here : read((p, r, t, mp, y)) ;
```

32 al. 2

```
outf(standout,
     $l"repayment.schedule.of.a.loan.of."9zd.2d,
      l"interest.rate.per.unit."d.4d,
       ".converted."2zd".times.per.year",
      l"monthly.payment."7zd.2d ,".for."2zd".years."$,
      (p, r, t, mp, y)) ;
  if r > long 1.0
  then print((newline, "interest rate is too high"))
  else long real mi = ¢monthly increment multiplier¢
  longexp(leng(t / 12) * longln(long 1.0 + r / leng t)),
  long real ap ¢accumulated principal at the end of the month¢ ;
    if (mi - long 1.0) * p > mp
    then print((newline, "payment does not cover interest"))
    else int j := 0 ¢the month number¢,
    long real interest ; y *:= 12 ;
    outf(standout, $l 2x8a, 3(12a)$,
      ("month", "amount", "principal", "interest")) ;
    format(standout, $l 4zd, 3(7zd.2d)$)
    ¢this associates a format with the standard output file¢ ;
    again : ¢return to this point for each monthly calculation¢
    j +:= 1 ; ap := p * mi ; interest := ap - p ;
      if j ≥ y ¢number of years is satisfied¢
         or ap ≤ mp ¢the last payment is due¢
      then out(standout, (j, 0.0, p, interest))
      else ¢regular monthly payment¢ ; p := ap - mp ;
      out(standout, (j, p, mp-interest, interest)) ;
      go to again
      fi
    fi
  fi
end□
```

The output from a run of the above program should be

```
REPAYMENT SCHEDULE OF A LOAN OF        1000.00
INTEREST RATE PER UNIT 0.0800 CONVERTED   4 TIMES PER YEAR
MONTHLY PAYMENT        100.00 for   1 YEARS.
  MONTH    AMOUNT       PRINCIPAL   INTEREST
    1      906.62        93.38      6.62
    2      812.63        94.00      6.00
    3      718.01        94.62      5.38
    4      622.76        95.24      4.76
    5      526.89        95.88      4.12
    6      430.38        96.51      3.49
    7      333.23        97.15      2.85
    8      235.43        97.79      2.21
    9      136.99        98.44      1.56
   10       37.90        99.09      0.91
   11        0.00        37.90      0.25
```

Review questions

2.1 Declarers

a) Is □**real** **ref**□ a •declarer•?
b) Is □**ref**[ ]**ref** **real**□ a •declarer•?
c) Write down a •declarer• specifying the mode •reference to reference to row of character•.
d) Is □[ ]**format**□ a •declarer•?
e) Is □**ref** **format**□ a •declarer•?
f) Is □**real** **proc**□ a •declarer•?
g) Can a value be of more than one mode?
h) Does a mode specify a •declarer•?

2.3 Local generators

a) How many •real-generators• are there [R.8.5.1.1]?
b) Write down a •local-generator• which possesses a value of mode •reference to character•.
c) Write down a •reference-to-boolean-local-generator•.
d) Is there an •integral-local-generator•?
e) Is the following a production rule of the strict language [R.1.1.5.a]?
•reference to row of character local generator :
local symbol, actual format declarer.•
f) Is •real-procedure-with-boolean• a mode [R.1.2.1]?

2.4 Evaluation of a generator

a) Does the •generator• □**loc** **real**□, after elaboration, possess a real value?
b) Does the •generator• □**loc** **real**□, after elaboration, possess a value?
c) Can a real value refer to a •generator•?
d) Can a real value refer to a name?
e) Can a name refer to more than one value [R.2.2.3.5.a]?
f) Can a name refer to more than one instance of a value [R.2.2.3.5.a]?

2.5 Identity declarations

a) Can two different external objects possess the same internal object?
b) In the reach of □**int** m = 2□, can the value possessed by □m□ be changed?
c) In the reach of □**ref** **real** x = **loc** **real**□, can the value possessed by □x□ be changed?
d) Write down a •local-generator• which, after elaboration, possesses a value of mode •reference to row of procedure real•.

2.6 Syntax of identity declarations

a) Is □**mode** **a** = **real**□ an •identity-declaration•?
b) Is □**ref** **real** x□ a •declaration•?
c) In the •declaration• □**ref** **int** nn□, what is the mode of □nn□?

d) Write a •declaration• of □p□ as a •reference-to-row-of-procedure-real-mode-identifier•.

2.7 Formal parameters

a) Is □real n□ a •formal-parameter•?
b) Is □[ ]proc real pqr□ a •formal-parameter•?
c) Is □loc real□ a •formal-parameter•?
d) Is □int 1□ a •formal-parameter•?

2.8 An extension

a) Write the •declaration• □ref real xx□ in the strict language.
b) Write the •declaration• □real x, y□ in the strict language.
c) Write the •declaration• □real x, y := 3.14□ in the strict language.
d) Write □ref ref real xx = loc ref real + 3.14□ in the extended language [R.9.2.a].

2.9 An assignation

a) Is □2.3 := 3.4□ an •assignation•?
b) Does an •assignation•, after elaboration, possess a value?
c) Can an •assignation•, after elaboration, possess a real value?
d) Is □(x := 3.14) := 3.15□ an •assignation•?

2.10 Syntax of assignations

a) Is □loc real := 2.3□ an •assignation•?
b) Is □loc ref real := x□ an •assignation•?
c) Is □loc ref real := 3.14□ an •assignation•?
d) What is the •source• in the •assignation• □x := y + 2□?
e) What is the mode of the •assignation• □xx := x□ (in the reach of □ref real xx, real x□)?
f) In the reach of □bool t = true□, is □t := false□ an •assignation•?

2.12 Dereferencing

a) What is the essential difference between the elaboration of □x := y□ and □x := 3.14□?
b) Is any dereferencing necessary in the •assignation• □xx := x□, in the reach of □ref real xx, real x□?

2.13 Initialized declarations

a) What are the modes of □m□ and □n□ in the •declarations• □int n = 2□ and □int m := 2□?
b) Make a diagram illustrating the •assignation• □nn := n := 1□, in the reach of □ref int nn, int n□.
c) Is it possible to apply an extension[R.9.2.a] to □ref real x = real := 3.14□?

2.14 Program example

a) How many occurrences of an •assignation• are there in this •particular-program•?
b) What coercions are involved in the elaboration of ¤p := ap - mp¤?
c) What is the effect of ¤j +:= 1¤ [R.10.2.11.d]?
d) Are there any •identifiers• which are •constants•?
e) What is the mode of ¤p¤?

3 Unitary clauses

3.1 Introduction

The •unitary-clause• [R.8] is one of the basic building blocks of the language. It corresponds roughly to what is known as the statement or the expression in ALGOL 60. Some examples of •unitary-clauses• are, □x := y, x + y, re of z, 123□ and □(x := 1 ; y := 2)□. •Unitary-clauses• are classified further into •confrontations, formulas, cohesions, bases• and other objects like •closed-clauses•. Thus, □x := y□ is a •confrontation•, □x + y□ is a •formula•, □re of z□ is a •cohesion•, □123□ is a •base• and □(x := 1 ; y := 2)□ is a •closed-clause•.

We now give a simplified syntax of •unitary-clauses•, using the ordinary typefont, to remind the reader that this is only an approximation to the syntax. The exact rules are in the Report [R.8.1.1], but a simplified syntactic tree is in figure 3.1.

```
  unitary clause : tertiary ; confrontation.
  tertiary : secondary ; formula.
  secondary : primary ; cohesion.
  primary : base ; closed clause ;
    conditional clause ; collateral clause.
```

```
        unitary-clause
              |
              |--------------------.
              |                    |
        tertiary              confrontation
              |
              |--------------------.
              |                    |
        secondary               formula
              |
              |--------------------.
              |                    |
         primary                cohesion
              |
   .----------+--.-----------------.--------------------.
   |             |                 |                    |
  base closed-clause conditional-clause collateral-clause
```

Fig.3.1

The purpose of this chapter is to study some of the simpler aspects of •unitary-clauses• and to observe the usefulness of the classification introduced by the syntax just given. This classification will help us to decide, for example, the order of elaboration in a •clause• like

□a or b := c of d of e[f] - g□[1] ,

where the modes of □a, b, c, d, e, f□ and □g□ are unknown. In fact the order is as if we wrote

----

[1] Note that the operator □or□ may be declared in such a way that it delivers a name.

¤(a or b) := ((c of (d of (e[f]))) - g)¤ .

The purpose of this syntactic classification, then, is to relieve the programmer of the necessity for supplying these parentheses himself. In addition, it aids the compiler by excluding certain mode dependent parsings.

•Unitary-clauses• which deliver no value are known as •statements• [R.6.0.1.c], while other •unitary-clauses• are known as •expressions• [R.6.0.1.b]. This distinction is largely historical and is of no significance in ALGOL 68.

3.2 Bases

•Bases• are the most elementary •unitary-clauses•, so we begin with them. Some examples of •bases• are ¤pi, 123, a[i], sin(x)¤ and ¤(: random)¤. A simplified syntax for base is

base : mode identifier ; denotation ;
slice ; call ; void cast pack. ,

but the strict syntax of the Report should be studied [R.8.6.0.1]. •Identifiers• are as in other programming languages, e.g., ¤random¤ and ¤j14283c¤. •Denotations• we have met before in section 1.5, e.g., ¤758¤ is an •integral-denotation•, ¤3.1¤ is a •real-denotation•, ¤false¤ is a •boolean-denotation•, ¤"q"¤ is a •character-denotation• and ¤"abc"¤ is a •string-denotation•. Thus we are already familiar with several objects which are •bases•. The objects ¤x1[i]¤ and ¤x2[d:e,j]¤ are •slices•, ¤sin(x)¤ is a •call• and ¤(: random)¤ is an example of a •void-cast-pack•. The classification of these objects as •bases• tells us where they stand in the order of elaboration, and we shall see later, also, that a •base• is one kind of •coercend• [R.8.2], i.e., an object upon which all coercions must be expended. But coercion is a subject for chapter 6.

3.3 Identifiers

A •mode-identifier• [R.4.1.1.b] is so called in order to distinguish it from a •label-identifier•, which is not a •base•. Both of these •identifiers• might be described by the following simplified syntax rule

identifier : letter ; identifier, letter ; identifier, digit.

which means that an •identifier• is what one expects it to be from the use of that term in other programming languages, i.e., a letter followed, perhaps, by any number of letters or digits. The strict syntax, in the Report [R.4.1.1.b,c,d], looks more complex, for a reason which will appear in later discussions concerning •field-selectors• [R.7.1.1.i]. Some examples of •identifiers• are, ¤algol 68, a, a3b7d9, random, st pierre de chartreuse¤ (note that spaces are of no significance within •identifiers•).

A •mode-identifier• usually possesses a value. This value is the same as that possessed by the same •identifier• at its defining occurrence. In the •assignation• ¤x := y + 3¤, the •mode-identifier• ¤x¤, supposedly in the reach of the •declaration• ¤real x¤, possesses a name which refers to some real value. The value (name, see figure 3.3 at 1) which it possesses is, in fact, a copy [R.8.6.0.2.a] of the value (see figure at 2) possessed by ¤x¤ at its defining occurrence, i.e., its occurrence as the •identifier• of an •identity-declaration•. The effect of the elaboration of the second occurrence of ¤x¤ in ¤real x ; x := y + 3¤ is shown pictorially in the figure 3.3,

```
¤real x ;                      x := y + 3¤
     :                          :
     o          (3)             o
 (2) o  o <-(identity)->  o  o (1)
     o                          o
     |        --------          |
     --->----|        |----<----
              --------
```

Fig.3.3

where the identity of the two instances of the same name is indicated at 3. In this figure one should note that the second occurrence of ¤x¤ possesses a copy of the name possessed by the first occurrence of ¤x¤. Consequently both names refer to the same instance of a real value [R.2.2.2.1]. The reader should consult the Report [R.4.1.2] which contains a careful description of the method by which this identification of •identifiers• is made.

3.4 Slices

We continue our discussion of •bases•; the next are •denotations•, but we have seen these before in chapter 1, so we go on to •slices•. In the reach of the •declarations• ¤[1:n]real x1, [1:m,1:n]real x2¤, the following are examples of •slices•

¤x1[i], x2[i,j], x2[,j], x1[2:n], x2[i,@0], x2[i]¤ .

A simplified syntax of •slice• is

```
  slice : primary, sub symbol, indexer, bus symbol.
  indexer : trimscript ; indexer, comma symbol, trimscript.
  trimscript : trimmer ; subscript.
```

but the strict syntax of the Report [R.8.6.1.1] contains much more than the skeleton shown above.

The most important point to notice about a •slice• is that its first constituent notion, e.g., the ¤x1¤ in ¤x1[i]¤, is a •primary•. Also notice that a •slice•, being a •base•, is itself a •primary•. Following the •primary• of a •slice• is a •sub-symbol•, represented by ¤[¤, then an •indexer• and finally a •bus-symbol•, represented by ¤]¤. Thus all of the following, in the above examples, are •indexers•: ¤i¤, ¤i,j¤, ¤,j¤, ¤2:n¤, ¤i,@0¤. An •indexer• is one or more •trimscripts•, separated by •comma-symbols•. A •trimscript• is a •trimmer• or a •subscript•. The objects ¤i¤ and ¤j¤ are •subscripts• and ¤2:n¤ and ¤@0¤ are •trimmers•. A •subscript• is an •integral-tertiary•.

In order to accommodate those users whose computers have a limited character set, a •slice• like ¤x1[i]¤ may also be written ¤x1(i)¤ [R.9.2.g]. However, we shall not use this possibility in this text since it then becomes difficult to distinguish between a •slice• and a •call•, like ¤sin(x)¤.

3.5 Multiple values

A multiple value, as we have seen in chapter 1, is a row of values [R.2.2.3.3.a]. We may represent it diagrammatically as in

Fig.3.5.a

figure 3.5.a, though we shall see later that this picture is not complete. Sometimes a name may refer to a multiple value, in which case we may think of it as a multiple •variable•. The difference between the effect of slicing a multiple •variable• and that of slicing a multiple •constant• is important and we shall now investigate it by example. Suppose we have the two •declarations• ¤[1:3]int n1 := (1, 2, 3)¤ and ¤[1:3]int u1 = (1, 2, 3)¤. The object ¤(1, 2, 3)¤ looks and acts like a •denotation• of a row of integers, but it is actually a

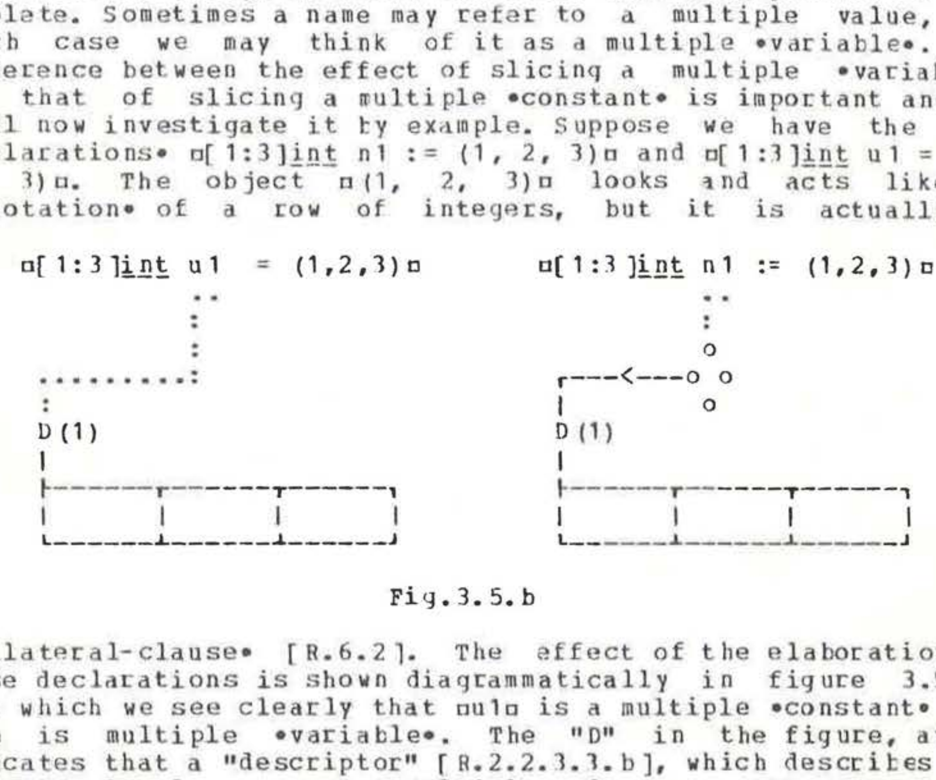


Fig.3.5.b

•collateral-clause• [R.6.2]. The effect of the elaboration of these declarations is shown diagrammatically in figure 3.5.b, from which we see clearly that ¤u1¤ is a multiple •constant• and ¤n1¤ is multiple •variable•. The "D" in the figure, at 1, indicates that a "descriptor" [R.2.2.3.3.b], which describes the elements, is also part of a multiple value. For the moment we shall ignore the presence of a descriptor. If we subscript a multiple •constant• we would expect to obtain a •constant•, e.g., ¤u1[2]¤ but if we subscript a multiple •variable•, we obtain a •variable• [R.2.2.3.5.c], e.g., ¤n1[2]¤. Thus ¤n1[2] := 4¤ is an •assignation• but ¤u1[2] := 4¤ is not. This is shown diagrammatically in figure 3.5.c, where the name possessed by ¤n1[2]¤ (at 1) is constructed from the name possessed by ¤n1¤ and the •subscript• ¤2¤ [R.2.2.3.5.c]. The effect is obtained syntactically by the fact that the •primary• of a •slice• is in a weak position. It involves the concept of weak coercion [R.8.2], which we will discuss more fully in chapter 6.

```
              □u1[2]                                      n1[2]□
               .....                                       .....
                 :                                           :
                 :                                          o(1)
                 :                                         o o
                 :                                          o
                 :                                          |
  ┌ - - - ┌---:---┐- - - ┐             ┌ - - - ┌---+---┐- - - ┐
          |       |                            |       |
  └ - - - └-------┘- - - ┘             └ - - - └-------┘- - - ┘
```

Fig.3.5.c

Observe now the use of the word •weak• in the rule 8.6.1.1.a of the Report.

3.6 Trimmers

A programmer who is manipulating multiple values may wish to choose certain subsets of a multiple value and to allow an external object to possess that subset or a name to refer to it. For example, one may wish to choose a row or a column of a matrix or even a submatrix of a given matrix. This may be done by using a •trimmer•, although, if that subset is to consist of a single element, then •subscripts• are sufficient. To illustrate the use of •trimmers•, consider the •declaration• □[1:3]int n1 := (5, 7, 9)□. The •slice• □n1[2]□ is a •variable• referring, at the moment, to ■7■, but the •slice• □n1[2:3]□ is a •variable• referring to a row of two integral values ■7■ and ■9■; moreover, being a •primary• itself, it may be subscripted (if one insists on being foolish), so that □n1[2:3][1]□ is a •variable• referring to the same integral value ■7■ and the •formula• □n1[2:3][1] = n1[2]□ possesses the value ■true■. In fact, it will always be ■true■ no matter what assignments are made to □n1□. Another way of saying this is that the •identity-relation• □n1[2:3][1] :=: n1[2]□ possesses the value ■true■.

The effect of the •trimmer• □l:u□ is then to restrict the range of values of the subscript to run from the value of □l□ to the value of □u□ and to renumber, starting from ■1■. If the renumbering from ■1■ is not desired, then the •trimmer• should be written □l:u@b□, where the value of □b□ is to be taken as the new lower bound. This means that, e.g., □n1[2:3@0][0] :=: n1[2]□ possesses the value ■true■. We may think of this in the sense that if □@b□ is omitted, then the default value of □b□ is ■1■, but the fact that the •new-lower-bound-part• may be empty is actually built into the syntax [R.8.6.1.1.f]. A further examination of the syntactic rule for •trimmers• reveals that the □l□, the □u□ and the □@b□ may be omitted, i.e., the •lower-bound• or the •upper-bound• or the •new-lower-bound-part• may be empty [R.8.6.1.1.f]. If the •lower-bound• of a •trimmer• is empty, then the lower bound of the •slice•, in that subscript position, is the same as that of the •primary• which is being sliced; if the •upper-bound• is empty, then the corresponding upper bound of the •slice• is the same as that of the •primary•; if the •new-lower-bound-part• is empty, then the subscripts of the •slice•, in that subscript position, will start from ▪1▪. It is even possible for all three to be empty at the same time. Thus ¤n1[:] :=: n1[1:3]¤ will possess the value ▪true▪. Extension 9.2.f, in the Report, allows the •up-to-symbol• to be elided, under certain circumstances, so that the above •identity-relation• might be written ¤n1[] :=: n1[1:3]¤.

If the •declaration• ¤[1:m,1:n]real x2¤ is used as that of an m by n matrix, then ¤x2[i]¤ refers to the i-th row of the matrix, ¤x2[:,j]¤, or even ¤x2[,j]¤ [R.9.2.f], to the j-th column and ¤x2[a:b,c:d]¤ may refer to a certain submatrix, if the values of ¤a, b, c¤ and ¤d¤ are appropriate. The rules for •trimmers• [R.8.6.1.1.f,g,h] should be examined to see that ¤l, u¤ and ¤b¤ in ¤l:u@b¤ are all •integral-tertiaries•. In particular, a •formula• is a •tertiary• but an •assignation• is not, so that ¤x2[i +:= 1, j of r]¤ is an acceptable •slice• but ¤x2[i := i + 1, j of r]¤ is not. The latter, to be acceptable, should appear as ¤x2[(i := i + 1), j of r]¤.

3.7 Calls

A simplified syntax of a •call• is

```
call : primary, open symbol, actual parameters, close symbol.
actual parameters : actual parameter ;
  actual parameters, gomma, actual parameter.
gomma : go on symbol ; comma symbol.
```

but the strict syntax is to be found in the Report [R.8.6.2.1.a, 5.4.1.c, 5.4.1.d]. Examples of •calls• are ¤sin(x), char in string ("a", i, s)¤ and ¤f(n; a, b)¤. These are familiar features from other programming languages, except perhaps the possibility of using a •go-on-symbol•, represented by ¤;¤, to separate the •actual-parameters• of a •call•. This possibility is present so that the programmer may, if he so wishes, match a similar use of a •go-on-symbol• in the corresponding •routine-denotation• [R.5.4.1], where its use will force the elaboration of the •actual-parameters• serially rather than collaterally. Thus, in the •call• ¤f(n; a, b)¤, the ¤n¤ might be used as a bound for the arrays ¤a¤ and ¤b¤, provided that a •go-on-symbol• was used in a similar position in the •routine-denotation• possessed by ¤f¤. Note that the •go-on-symbol• in a •call• has a decorative effect only. It is the presence of a •go-on-symbol• in the •formal-parameters-pack• of a •routine-denotation• which has the controlling effect.

•Routine-denotations• are important and must be understood before we examine the semantics of •calls•; however, •routine-denotations• will be discussed in chapter 5, so we will postpone our explanation of these semantics until that time.

The most important point to notice about the syntax of a •call• is that its first constituent notion, e.g., ¤sin¤ in ¤sin(x)¤, must be a •primary•. Also notice that a •call• itself is a •primary• so that ¤a(b)(c)(d)¤ might well be a •call• in which the order of elaboration is that suggested by ¤((a(b))(c))(d)¤. As we have already remarked, in section 3.4, in some programs it may not be possible to determine whether

¤a(b)¤ is a •slice• or a •call•, without knowing the mode of ¤a¤, but since the parsing tree is similar for these two, this is of no great hardship for the compiler. We shall see later that the object ¤if x < pi/2 then cos else sin fi¤ is a •primary• and therefore ¤if x < pi/2 then cos else sin fi (x)¤ is a •call•. It so happens that ¤begin r := s + 2 ; sin end (x)¤ is also a •call•, and perhaps some programmer will find it useful.

3.8 Void cast packs

An example of a •void-cast-pack• is

¤(¢void¢ : x := 2 * x + 1)¤ .

Its purpose is to void the mode of the •unit• contained therein in those situations where this is not done implicitly, such as in ¤; x := 2 * x + 1 ;¤, where the •assignation• is turned into a •statement• by the fact that it is preceded and followed by •go-on-symbols•. An example where a •void-cast-pack• is needed is

¤proc ¢void¢ p = (¢void¢ : x := 2 * x + 1)¤

where ¤p¤ is made to possess a routine, which contains an •assignation• but the •assignation• should not itself be elaborated until ¤p¤ is called. The object ¤proc ¢void¢ p = (x := 2 * x + 1)¤ is not an •identity-declaration• (the programmer might find it confusing anyway). A full explanation of the above •declaration• involves the concept of coercion which we shall take up in chapter 6. Readers whose curiosity is aroused may wish to follow the syntactic analysis suggested by 74a,b, 61e, 81a,b,c,d, 820d, 823a, 860b, 834a, 61e, 81a, 820d, 828a, and those who could have found it for themselves need not be reading this book!

A simplified syntax of •void-cast-pack• is

void cast pack :
  open symbol, cast of symbol, unitary clause, close symbol.

but the strict syntax is found in more than one place in the Report [R.8.3.4.1.a, 3.0.1.h, 7.1.1.z].

The •void-cast-pack• may appear to play the role of a •routine-denotation• in the case of those routines which deliver no value and have no •parameters•. An examination of the Report [R.5.4.1] will reveal that there are indeed no such •routine-denotations•. There is however, a proceduring coercion and this, together with the •void-cast-pack• fills the need. But more about this later.

3.9 Cohesions

A •cohesion• is either a •generator•, e.g., ¤real¤, or a •selection•, e.g., ¤re of z¤. The strict syntax is:

•MODE cohesion : MODE generator ; MODE selection.•

[R.8.5.0.1.a]. A •cohesion•, like a •base•, is also a class of •coercend• upon which all coercion must be expended, but we shall discuss coercion later. We have already examined •generators•, so we now turn to •selections•.

3.10 Selections

An example of a •selection• is □re of z□ in the reach of the •declaration• □struct(real re, im) z□. A simplified syntax of •selection• is

selection : field selector, of symbol, secondary.

but in the strict syntax of the Report [R.8.5.2.1.a] several metanotions are used with penetrating effect. In order to understand the meaning of a •selection•, we need to know that some values, unlike multiple values, may be built from several values whose modes may be different. Thus we may build a "structured" value consisting of one or more "fields" [R.2.2.3.2] in which the value of each field has, possibly, a different mode. The fields of a structured value are then selected by •field-selectors•, which look like •identifiers• but which, syntactically, are not •identifiers•. For example, in the •selection• □re of z□, the •field-selector• is □re□.

An example of a •declarer• which specifies a structured mode is □struct(real value, string name)□. Values of such a mode then consist of two fields, one whose mode is •real• and another whose mode is •row of character•. If one wishes to obtain, or assign to, the •real• field of a •variable• □r□ referring to a value of such a mode, this is done by using the •selection• □value of r□; the string field is obtained by the •selection• □name of r□. Note the similarity with the •slice• □x1[i]□, where an element is selected from the value of the •primary• according to the value of the •subscript• □i□. In the selection □value of r□, an element is selected from the value of the •secondary• □r□, using the •field-selector• □value□. There is, however, one essential difference in that the value of the subscript, □i□, may vary dynamically, whereas the •field-selector•, □value□, cannot. This makes field selection an inherently efficient process.

As with a •slice•, the value of a •selection• from a •secondary• which is a •variable•, is also a •variable•, but the value of a selection from a •secondary• which is a •constant•, is a •constant•. Thus with the •declarations• □struct(int i, bool b) ib := (1, true)□ and □ struct(real r, char c) rc = (1.2, "k")□, □i of ib□ is a •variable• and □i of ib := 2□ is an acceptable •assignation•; however, □c of rc□ is a •constant• and □c of rc := "m"□ is not permitted. The reader may wish to note that these effects are obtained, syntactically, through the use of the metanotion REFETY and the word •weak• in the rule 8.5.1.1.a of the Report. The same remark applies to the rule 8.6.2.1.a for •slice•.

It is important to observe that a •selection• is always made from a •secondary• and in this way it differs from a •slice•, since only a •primary• can be sliced. This means that the order of elaboration of the object □a of b[c]□ must be the same as that of □a of(b[c])□, for □a of b□ is not a •primary•. Also, a •selection• is itself a •secondary• so that □a of b of c of d□ may be a •selection• whose order of elaboration is suggested by □a of (b of (c of d))□. Observe that if □d□ is a

•variable• then □a of b of c of d□ is also a •variable•.

3.11 Formulas

A simplified syntax of •formula• is

```
  formula : operand, dyadic operator, operand ;
    monadic operator , operand.
  operand : tertiary.
```

but the strict syntax contains much more information [R.8.4.1]. •Formulas• with two •operands• are known as •dyadic-formulas• and those with one •operand• are •monadic-formulas•. Since the same symbol may be used both as a •dyadic-operator• and as a •monadic-operator•, as for example in □(- a - b)□, one must rely upon some context to determine the full extent of a •formula•.

A major new feature of ALGOL 68 is the fact that operations may be declared. This means that any •operator•, e.g., □+□, may not mean what we think it means unless we have examined the •ranges• in which it occurs. An example of an •operation-declaration• is

□op or = (real a, b)real : if a > b then a else b fi□ ,

but since this involves •routine-denotations•, which we have not yet discussed, we shall postpone a full examination of •operation-declarations•.

The syntax given above shows that an •operand• must be a •tertiary•. Also, the syntax given in section 3.1 [R.8.1.1.b] shows that a •formula• is itself a •tertiary•. From this we may deduce that the elaboration of the •formula• □a of b[i] + c□ is in the order suggested by □(a of (b[i])) + c□. The reader may find the following summary useful:

a •primary• may be sliced and a •slice• is a •primary•,
a •secondary• may be selected from and a •selection• is a •secondary•,
•operands• are •tertiaries• and a •formula• is a •tertiary•,
[R.8.6.1.1.a, 8.6.0.1.a, 8.5.2.1.a, 8.5.0.1.a, 8.4.1.f, 8.1.1.b,c,d].

A set of standard operations, which the programmer might

DYADIC									MONADIC
1	2	3	4	5	6	7	8	9	(10)
□-:=	or	&	=	<	−	*	up	i	¬ − + / down up
+:=			≠	≤	+	+	lwb		abs bin repr
*:=				≥		+:	upb		lwb upb lws ups
/:=				>		/	lws		leng short
+:=						elem	ups		odd sign round
+::=									re im conj
+=:									btb ctb□

Fig.3.11

expect of any programming language, is provided [R.10.2] and standard priorities (from 1 to 9) are given [R.10.2.0]. This standard set is to be found, in summary, in 8.4.2 of the Report and is reproduced here for convenience. There are nine priorities (from 1 to 9) for the •dyadic-operators•. The •monadic-operators• all have the same priority (effectively 10) and when used consecutively, are elaborated from right to left. A typical •priority-declaration• is

¤priority + = 6¤

and in fact, this is to be found in the •standard-prelude• [R.10.2.0.a]. Operations whose •operators• have the highest priority are elaborated first. This means, e.g., that the •formula• ¤a < b = c > d¤ is elaborated in the order suggested by ¤(a < b) = (c > d)¤. Also, the value of ¤(-1 up 2 + 3)¤ and ¤(3 - 1 up 2)¤ are •4• and •2• respectively, a fact which may come as a surprise to users of some other languages(1). In justification of this choice one must observe that, when •operators• and their priorities may be declared, a simple rule for the priority of •monadic-operators• is essential. Consider, for example, the formula

¤x a b c y d e z¤ .

We know immediately that the order of elaboration is that suggested by

¤x a (b (c y)) d (e z)¤ ,

since the monadic operations are performed first, while the priorities of the •dyadic-operators• ¤a¤ and ¤d¤ will settle any doubt which may remain.

It would take too long to describe all the operations which are provided in the •standard-prelude•, and indeed this would be a waste of time, for their precise definition is given in Chapter 10 of the Report. We shall be content with mentioning some of the less familiar •operators•, beginning with those of the highest priority. i.e., the •monadic-operators•. The •operator• ¤leng¤ operates on an integral, a real or a complex value delivering a value whose length (precision) is increased, while ¤short¤ has the opposite effect. In some installations this may mean the change from single precision to double precision and the reverse [R.10.2.3.q, 10.2.4.n, 10.2.7.n]. One should be careful to distinguish between ¤leng 1.0¤ which is a •formula•, and ¤long 1.0¤, which is a •denotation• [R.5.1.0.1.b]. The value of ¤odd 4¤ is •false• [R.10.2.3.s]. The value of ¤bin 5¤ is that of ¤101¤, i.e., ¤bin¤ operates on integral values and delivers bits [R.10.2.8.1]. The value of ¤abs "a"¤ is some integral value, which is implementation dependent, and that of ¤repr abs "a"¤ is •a•, i.e., ¤repr abs¤ is the identity operation on any character [R.10.1.j,k]. Also, ¤abs true = 1, abs false = 0¤ [R.10.2.2.f] and ¤abs 101 = 5¤ [R.10.2.8.i], all have the value •true•; in fact, ¤bin abs¤ is the identity operation on certain bits values. The operator ¤btb¤ converts •row of boolean• to bits, e.g., ¤btb(true, false, true) = 101¤ [R.10.2.8.1] and ¤ctb¤ converts •row of character• to bytes [R.10.2.9.d]. The inverses of ¤btb¤ and ¤ctb¤ are not

----

(1) Except for users of, e.g., JOVIAL, SNOBOL and APL.

necessary since that job is done by coercion [R.8.2.5.1.c,d]. The •monadic-operators• □up, down□ and □/□ operate on semaphores and are concerned with synchronization (parallel processing). We shall not discuss them further here [R.10.4]. The operators □upb, lwb, ups□ and □lws□ are concerned with arrays. We may best illustrate them by considering the •declaration• □[2:5 flex]int n1□, so that □n1□ is a •variable• referring to a row of integral values whose index has a lower bound of ▪2▪, which is fixed and an upper bound of ▪5▪, which is flexible. Then □upb n1 = 5, lwb n1 = 2, ups n1 = false, lws n1 = true□(1). These •operators• are also dyadic and □1 upb n1 = upb n1□, for all arrays □n1□, while the •formula• □2 upb n2□ delivers the value of the upper bound in the second subscript position of the array □n2□.

There is one standard •dyadic-operator• □i□ or □!□ of priority 9 (the programmer may create more if he wishes). The value of □x i y□ is a complex number with real part □x□ and imaginary part □y□ [R.10.2.5.f]. In the standard •declarations• the result of the •dyadic-operator• □/□, •divided-by•, is real (or complex) and that of □÷□ is integral (integral division of two integral operands). The operator □elem□ delivers an element from bits or bytes, e.g., □2 elem 101□ delivers ▪false▪. Note that □2 elem b := true□ is not an •assignation• [R.10.2.8.k, 10.2.9.c]. Manipulation of bits can be achieved with the operators □or, and, up□ and □not□ [R.10.2.8.d,e,h,m]. The value of □n ÷: m□ is □n□ modulo □m□, i.e., the remainder obtained on dividing □n□ by □m□ [R.10.2.3.n]. Apart from the fact that □abs□ is an operator on real, integral and complex values, rather than a •call•, i.e., it is not □abs(x)□, the remainder of the •operators• are probably familiar to most programmers with the exception of a set of •operators• of lowest priority ▪1▪. A typical example is □+:=□, which we can explain by saying that the •formula• □x +:= 1□ has the same effect as □x := x + 1□. Another •dyadic-operator• with priority ▪1▪ is □+=:□, which may be used with two •operands• of mode •row of character• [R.10.2.11.r,t]. After elaboration of the •formula• □s +=: t□, in the reach of □string s := "abc", t := "def"□, we have □s = "abc"□ and □t = "abcdef"□. On the other hand, after the elaboration of the •formula• □s +:= "g"□, we have □s = "abcg"□.

The reader should be careful to note that several •operators• have more than one representaticn, e.g., the •plus-i-times-symbol• has three representations and the •up-symbol• four [R.3.1.1.c] (morevoer, many representations are not available in this preliminary edition due to the limitations of the TN print chain).

3.12 Confrontations

There are four kinds of •confrontation• according to the strict rule

----

(1) Here it is more convenient to say □2*2 = 4□ rather than the longer but correct statement □2*2 = 4□ possesses the value ▪true▪.

•MODE confrontation : MODE assignation ;
MODE conformity relation ;
MODE identity relation ; MODE cast.•
[R.8.3.0.1.a]. The object ¤x := y + 2¤ is an •assignation•, ¤r ::= i¤ is a •conformity-relation•, ¤a :=: b¤ is an •identity-relation• and ¤real : i¤ is a •cast•. Enough has been said about •assignations• already in sections 2.9 and 2.10. •Conformity-relations• have to do with united modes, which we have not yet introduced, so it is as well to postpone this discussion to chapter 7. We shall therefore confine our attention here to •identity-relations• and •casts•. Before passing to these, we should see that since a •confrontation• is not a •tertiary•, and therefore not an •operand•, the elaboration of the •assignation• ¤xx or yy := x¤ is done in the order suggested by ¤(xx or yy) := x¤. Such an •assignation• might well be possible if the •operator• ¤or¤ has been declared in such a way that it will deliver a name.

3.13 Identity relations

There are two •identity-relators•, the •is-symbol•, represented by ¤:=:¤ and the •is-not-symbol•, represented by ¤:≠:¤. A simplified syntax of the •identity-relation• is
identity relation : tertiary, identity relator, tertiary.
but the strict syntax of the Report contains more detail to account for the balancing [R.6.4.1] of modes.

The elaboration of the •identity-relation• is normally quite simple. We ask the question whether two names, of the same mode, are the same. This means, in most implementations, asking whether two storage addresses are the same rather than whether they have the same content. As an example, suppose the •declaration• ¤real x, y¤ has been made. The •identity-relation• ¤x :=: y¤ then has the value ▪false▪, despite the possibility that we may have elaborated the •assignations• ¤x := 3.14, y := 3.14¤. This is because the •declaration• ¤real x¤ (strictly ¤ref real x = loc real¤) involves the elaboration of the •generator•, ¤loc real¤, which creates a name different from all other names [R.7.1.2.d Step 8]. The same applies to ¤real y¤. Hence, the name possessed by ¤x¤ is not the same as the name possessed by ¤y¤. After the •declaration• ¤ref real a = x¤, the name possessed by ¤a¤ is the same as the name possessed by ¤x¤, but a different instance of that name. Consequently, the value of the •identity-relation• ¤x :=: a¤ will be ▪true▪ and will remain ▪true▪ no matter what assignments are made to ¤a¤ or to ¤x¤. Notice that an assignment to ¤a¤ is at the same time an assignment to ¤x¤.

Now suppose that the •declaration• ¤ref int ii, jj, int i¤ is elaborated followed by the •assignations• ¤ii := i, jj := i¤. The •identity-relation• ¤ii :=: jj¤ possesses the value ▪false▪, for a similar reason to that explained above, but the •identity-relation• ¤jj :=: i¤ then possesses the value ▪true▪. That this is so can be seen by a close examination. We present this in figure 3.13. We see in the figure at 1 and 2 that the a priori modes of the •identifiers• on each side of the •is-symbol• are not the same. Since an •identity-relation• must have •tertiaries• of the same mode [R.8.3.3.1.a] (each of which begins with •reference-to•), there is a coercion, known as "dereferencing" [R.8.2.1.1], of the •base•, □jj□ (see the figure at 3), whereupon the •identity-relation• delivers the value ▪true▪ (see the figure at 4). Observe that there is, strictly speaking, a coercion on the right also, but since the a priori mode and the a posteriori mode are the same its semantic effect is therefore absent. Since the dereferencing may occur either on the left or on the right, but not on both sides, there are two alternatives in the strict syntax of •identity-relations• [R.8.3.1.1.a]. The reader should notice that in this syntax, one of the •tertiaries• is "soft" and the other is "strong".

```
                    boolean-identity-relation..............
                                 |                        : (4)
       ┌-------------------------+-------------------┐  ▪true▪
       |                         |                   |
 strong-reference-to-            |            soft-reference-to-
  integral-tertiary    identity-relator       integral-tertiary
       |                         |                   |
 strong-reference-to-            |           soft-reference-to-
    integral-base.........       |              integral-base
       |                 :       |                   |
    (coercion) (3)       :       |              (coercion)
       |          (1)    :       |                   |
 reference-to-reference  :       |              reference-to-(2)
   to-integral-base      :       |              integral-base
       ⊥                 :      ─⊥─                  ⊥
      □jj                :      :=:                  i□
       :                 :                           :
       o                 o                           o
     o o------->-----o o   <---(identity)--->      o o
       o                 o                           o
                         |                           |
                         |       ┌------┐            |
                         └--->---|      |----<-------┘
                                 └------┘
```

Fig.3.13

In the case of □jj :=: i□, the □i□ is soft and the □jj□ is strong. This is a matter concerned with coercion and the balancing of modes which will be discussed in chapter 6.

3.14 Casts

The object

□<u>real</u> : 2□

is a trivial example of a •cast• [R.8.3.4.1.a], but it is good enough to illustrate that a •cast• consists of a •declarer• followed by a •cast-of-symbol• followed by a •unitary-clause•. The purpose of a •cast• is to coerce the value of its •unitary-clause• into a value of mode specified by its •declarer•. The example given is trivial because its value could be obtained more easily from the •real-denotation• □2.0□.

•Casts• play an important role in •routine-denotations•, which are discussed in chapter 5. We shall see also that they are used instead of •routine-denotations• for those routines which lack •parameters•. Otherwise, a •cast• is occasionally useful to effect a coercion which is not implied by the context. For example, ¤string : "a"¤ is a multiple value, i.e., a row of characters with one element, and objects like ¤(ref cell : next of cell) :=: nil¤ are essential to list processing (see R.11.12). A •cast• may have a •void-declarer•, in which case it is a •void-cast•, e.g., ¤:x := y¤. A •void-cast• yields no value. An examination of the syntax will reveal that a •void-cast• occurs only as a •void-cast-pack• [R.8.6.0.1.b], e.g., ¤(: x := y)¤, or as part of a •routine-denotation• [R.5.4.1.b], e.g., ¤: get bin(stand back, x)¤ in ¤([ ]intype x) : get bin(standback, x)¤ [R.10.5.4.2.a]. A •void-cast-pack• is a •base•, as we have already seen in section 3.8. •Casts• which are not •void-casts• "envelop" [R.1.1.6.j] a mode and are •confrontations•. One reason for the exclusion of •void-casts• from •confrontations• is the ambiguity which might otherwise lurk in the object ¤x :=: y¤ or ¤x := :y¤.

For those •casts• which envelop a mode, a simplified syntax is

cast : virtual declarer, cast of symbol, unitary clause.

[R.8.3.4.1.a]. A •virtual-declarer• [R.7.1.1] is a •declarer• in which all •indexers• contain •bounds• which are empty. To find typical examples of •casts• we need only examine •declarations• involving routines, of which there are a large number in Chapter 10 of the Report. One of them is

¤op abs = (bool a)int : if a then 1 else 0 fi¤

[R.10.2.2.f] in which the •cast• is ¤int : if a then 1 else 0 fi¤.

The elaboration of a •cast• is that of its •unitary-clause• [R.8.3.4.2], always remembering that the mode of the value delivered, if any, is that specified by the •declarer• of the •cast•. Since the a priori mode of its •unitary-clause• is often not the same as that specified by its •declarer•, the final steps in the elaboration of a •cast• often involve some kind of coercion. For this reason it will appear frequently in our discussion of coercion in chapter 6.

Because a •cast• is a •confrontation• and therefore also a •unitary-clause•, it follows that ¤real : real : x¤ is a •cast•, but its value is the same as that of ¤real : x¤. Note that a •cast• which envelops a mode is not a •primary• or even a •tertiary•; consequently, ¤ref real : xx := 3.14¤ is not an •assignation•. The effect perhaps intended could be obtained by writing ¤(ref real : xx) := 3.14¤.

3.15 Program example

----

(1) The ALGOL 60 version of this procedure is due to G.F.Schrack.

The following is a •procedure-denotation•(1). The routine which is possessed by □p□ calculates the real coefficients of a polynomial whose zeros are the elements of a given complex vector □z□. These zeros may be real or complex, but if complex must appear consecutively as conjugate pairs. For example, if the given vector is □(1, 0 i 1, 0 i -1)□, then the polynomial will be □z**3 - z**2 + z - 1□. Thus, in the •range• of □[1:3]compl w := (1, 0 i 1, 0 i -1)□, the value of the •call• □p(w)□ will be that of □([ ]real : (1.0, -1.0, 1.0, -1.0))[@0]□. The existence of a non-local •procedure•, □error□, is assumed, for use upon encountering invalid data.

```
□proc p = (ref[1:]compl z)[ ]real :
¢calculates the coefficients of the real polynomial whose zeros
are the elements of the vector z¢
  begin [0:upb z]real a ; a[0] := 1 ; int i := 1 ;
  ¢the coefficients are calculated into the vector a¢
  while i ≤ upb z do
    begin compl zi = z[i] ; a[i] := 0 ;
    if im zi = 0
    then ¢ a real zero¢
      for k from i by -1 to 1 do
        a[k] -:= re zi * a[k-1]
    else ¢a pair of complex zeros¢
      if i = upb z then error fi ;
      if zi ≠ conj z[i+:=1] then error fi ;
      real s = re zi ** 2 + im zi ** 2, t = 2 * re zi ;
      a[i] := 0 ;
      for k from i by -1 to 2 do
        a[k] -:= t * a[k-1] - s * a[k-2] ;
      a[1] -:= t
    fi ; ¢and now for the next one¢ i +:= 1
    end ¢the iteration on i¢ ;
  ¢the coefficients are now ready in the vector a¢
  a end□
```

From □[ ]real :□, on the first line, to the final □end□ is the •cast• of a •routine-denotation• [R.5.4.1.b]. It begins with □[ ]real :□ to ensure that the value delivered by the routine is of mode •row of real•. Note the use of the •operator• □upb□ in the •declaration• □[0:upb z]real a□, which creates a vector •variable• with index running from ▪0▪ to the upper bound of □z□. The •declaration• □compl zi = z[i]□ [R.10.2.7.a] indicates that, for each value of □i□ in the iterative statement, □zi□ is a constant. This avoids repeated calculation of □z[i]□ later. Observe that, in the •formula• □zi ≠ conj z[i+:=1]□, the •formula• □i+:=1□ is elaborated first. The value of the •variable• □i□ is thus incremented by 1. The value of this •formula• is the name possessed by □i+:=1□, which is the same as the name possessed by □i□. It is then dereferenced. The object □z[i+:=1]□ is a •slice• whose value is the next zero of the polynomial sought. The •declaration• □real s = re zi ** 2 + im zi ** 2□ declares a •real-constant• □s□ whose value is the square of the modulus of one of the conjugate pairs. The value delivered by the routine is that of □a□; consequently □a□ appears as an •expression• preceding the final □end□.

Review questions

3.1 Introduction

a) Is a •cohesion• a •primary•?
b) Is a •closed-clause• also a •tertiary•?
c) Indicate by parentheses the order of elaboration of □a + b of c[d] - e□.
d) What is the difference between a •statement• and an •expression•?
e) Is a •base• also a •unitary-clause•?

3.2 Bases

a) Is □x + y□ a •base•?
b) How many kinds of •bases• can be distinguished?
c) List all the •bases• in the object
□(a[i] > b of c | sin(x) | cos(x + pi/2))□.
d) Is □3.□ a •base•?
e) Is □a(b)□ a •call• or a •slice•?

3.3 Identifiers

a) List the •identifiers• in the object □l:ca := char of file of f + "a5"□.
b) What is the mode of □x□ in □real x := 3.14□?
c) What is the mode of □n2□ in □[1:3, 1:4]int n2 = m2[3:5, 3:6]□?
d) Do □u□ and □v□ have the same mode in the •declaration• [1:10]char u, [1:10 flex]char v□?
e) Is □$line□ an •identifier•?

3.4 Slices

In the reach of the •declaration• □[1:m, 1:n]real x2, y2□ :
a) is □x2[1][1]□ a •slice•?
b) is □x2[1]□ a •slice• and if so what is the mode of its value?
c) is □begin x2 end[1,1]□ a •slice•?
d) is □if i > 0 then x2 else y2 fi [1,1]□ a •slice•?
e) Which of the following can be subscripts?
□35□, □item of a□, □i + n * 2□, □i := 2□, □i +:= 2□.

3.5 Multiple values

In the reach of the •declaration• □[1:m, 1:n]real x2, [1:3]int u1 = (1, 2, 3)□ :
a) is □u1□ a •variable•?
b) is □x2[1, 2]□ a •variable•?
c) is □u1[2] := 2□ an •assignation•?
d) is □x2[2][1] := 3.14□ an •assignation•?
e) is □x2[1, 1] := 3.14□ an •assignation•?

3.6 Trimmers

Using the •declaration• given in 3.5 above:
a) what is the value of □u1[2:]□?
b) what can be said about the •formula•
□x2[2:3][2,1] = x2[2,1]□?
c) what is the value of □u1[:2@0][1]□?
d) what is the value of □u1[@2][3]□?
e) is □x2[i:=1:j+:=1, 3]□ a •slice•?

3.7 Calls

a) Is □cos(x := pi/4)□ a •call•?
b) Is □random□ in □x := random□ a •call•?
c) Is □cos(x > 0 | x | pi/2)□ a •call•?
d) Under what conditions is □a(b)□ in □a(b) := c□ a •call•?
e) Under what conditions is □a(b)(c)□ a •call•?

3.8 Void cast packs

a) Is a •void-cast-pack• a •primary•?
b) Is □(: x) := y□ an •assignation•?
c) Is □x := (: y)□ an •assignation•?
d) Is □(:(x))□ a •void-cast-pack•?
e) Is □proc p := x := 3.14□ a •declaration•?

3.9 Cohesions

a) Is a •cohesion• a •primary•?
b) Is a •cohesion• a •tertiary•?
c) Is □(x + y)□ a •cohesion•?
d) Is □[1:3]ref struct(int a, real b)□ a •cohesion•?
e) Under what conditions is □a of b := c□ an •assignation•?

3.10 Selections

a) Is a •selection• a •primary•?
b) Is the □a□ in □a of b□ an •identifier•?
c) Indicate by parentheses the order of elaboration of
□a of b [c]□ and of □e of g(x)□.
d) Is □(a of b) of c□ a •selection•?
e) Is □a of (b of c)□ a •selection•?

3.11 Formulas

a) Is a •formula• a •tertiary•?
b) What is the value of □2 elem bin 5□?
c) What is the value of □lwb - 3.14□?
d) Is □4 +:= 2□ a •formula• and if so what is its value?
e) What is the value of □¬(1<2 and 3>4 or 5=6≠7>8 or true)□?

3.12 Confrontations

a) Is a •secondary• a •confrontation•?
b) Is □x1[i:=i+1] a •slice•?
c) Is □real□ a •confrontation•?
d) Is □proc : random□ a •confrontation•?
e) Is □p := x :=: y□ an •identity-relation• or an •assignation•?

3.13 Identity relations

In the reach of the •declaration• ¤<u>int</u> i, j ; <u>ref</u> <u>int</u> ii := i, jj := i¤ :

a) what is the value of ¤ii :=: jj¤?
b) what is the value of ¤i :=: jj¤?
c) what is the value of ¤i :≠: j¤?
d) Is ¤x :=: 3.14¤ an •identity-relation•?
e) Is ¤x :=: x1[2]¤ an •identity-relation•?

3.14 Casts

a) Is a •cast• a •primary•?
b) Is ¤<u>int</u> : 3.14¤ a •cast•?
c) Is ¤x := :y¤ an •assignation• or an •identity-relation•?
d) Is ¤[1:1]<u>real</u> : 3.14¤ a •cast•?
e) Is ¤<u>ref</u> <u>int</u> : ii := 2¤ an •assignation•?

3.15 Program example

a) How many occurrences of a •cohesion• are in this •particular-program•?
b) How many occurrences of a •slice• are there?
c) Is ¤t¤ a •constant• or a •variable•?
d) What is the mode of ¤s¤?
e) How many occurrences of an •identity-relation• are there?

4 Clauses

4.1 Conditional clauses

The •conditional-clause• [R.6.4] is a fundamental programming concept or primitive pertaining to flow of control. It is present in some form or other in most languages and allows for a choice in the elaboration of one out of two •serial-clauses•, depending on the value of a •condition•. An example of a •conditional-clause• is

□if a > b then a else b fi□

or, using another representation

□(a > b | a | b)□ ,

which therefore has the same meaning. A simplified parse is shown in figure 4.1.a.

```
                       conditional-clause
                               |
      ┌──────┬─────────────────┼─────────────────┬──────────┐
      |      |                 |                 |          |
  if-symbol  condition   then-clause       else-clause   fi-symbol
      |      |                 |                 |          |
      |      |          ┌──────┴─────┐     ┌─────┴─────┐    |
      |      |          |            |     |           |    |
      |   serial-     then-     serial-   else-     serial- |
      |   clause      symbol    clause    symbol    clause  |
     _|_    _|_        _|_         |        _|_        |    |
    □if     a>b        then        a        else       b   fi□
```

Fig.4.1.a

There are two features of the •conditional-clause• which are noteworthy. The first is that such a •clause• is closed, in the sense that it begins with an •if-symbol•, represented by □if□ or □(□, and ends with a •fi-symbol•, represented by □fi□ or □)□. As a consequence of this, a •conditional-clause• can be, and is, a •primary• and is therefore found in syntactic positions which might otherwise be considered unusual in some programming languages. The second is that no essential distinction is made between •conditional-expressions• and •conditional-statements•. The only difference is that, if a •conditional-clause• is used as a •statement• [R.6.0.1.c], then its value is voided; otherwise, it may be an •expression• [R.6.0.1.b][1] and may deliver a value. There is only one genuine syntactic rule [R.6.4.1]. This merging of concepts permits •conditional-clauses• like

□if a > 0 then sqrt(a) else go to error fi□ ,

which may be used in a situation like

□a1 := if a > 0 then sqrt(a) else go to error fi□ .

----

(1) Note that rules in the Report marked with an asterisk are present only for the convenience of the semantic description of the language. The notions involved never appear in the parse of a •program•.

Some uses of a •conditional-clause• which might be considered unusual, but which stem from the fact that it is a •primary• are: □(p | x | y) := 2.3 , (q | cos | sin)(x) , (r | x | y) + (s | u | v)□, in which we have used, for preference, the shorter representations.

A simplified syntax of the •conditional-clause• is

```
  conditional clause :
    if symbol, condition, then clause, else clause, fi symbol.
  condition : serial clause.
  then clause : then symbol, serial clause.
  else clause : else symbol, serial clause.
```

but the strict syntax in the Report [R.6.4.1] should be studied also. One should observe that a •conditional-clause• contains three •serial-clauses• (see figure 4.1.a). Any one such •serial-clause• may contain •declarations• and forms a •range• [R.4.1.1.e]. Since a •serial-clause• may contain more than one •unitary-clause•, this means that frequent use of □begin end□ pairs (•packages•), as in ALGOL 60, is not necessary. An example of a •conditional-clause• containing a non-trivial •condition• might be:

```
                □if string s ; read(s) ; s = password
                 then go to regular
                 else go to irregular
                 fi□
```

where the value of the •condition• is that of its last •unit•, □s = password□.

A •conditional-clause• is elaborated by first elaborating the •condition•. If the value of the •condition• is ▪true▪, then the •then-clause• is elaborated; otherwise, the •else-clause• is

```
             ▪true▪-->--┐         ┌-------->--------┐
                :       |         |                 |
              .....     |         |                 V
   □(         x > 0     |    x    |      -x       )□
              .....                               | |
                :                 |               | |
             ▪false▪---------->---┘               └->┘
```

Fig.4.1.b

elaborated (see figure 4.1.b). In the first instance, the value, if any, of the •conditional-clause• is that of the •serial-clause• of the •then-clause•; otherwise, it is that of the •else-clause•. For example, the •clause•

□(x ≥ 0 | x | -x)□

has as its value the absolute value of □x□.

4.2 Simple extensions of the conditional clause

A •conditional-clause• like

```
          □if a then b else if c then d else
             if e then f else g fi fi fi□
```

may occur frequently in programming situations. For this reason an extension [R.9.4.b] is available whereby the same •clause•

may also be written

□if a then b elsf c then d elsf e then f else g fi□ .

The essence of this extension is that □else if□ may be written □elsf□, if the corresponding □fi□ is elided. Using the other representations, the strict language is

□(a | b | (c | d | (e | f | g)))□ ,

which may be written

□(a | b |: c | d |: e | f | g)□ ,

in the extended language. This saves the programmer the bother of counting □fi□s so that they match the number of □if□s. A schematic flow of control for this •clause• is shown in figure 4.2 in the case where □a□ possesses the value ■false■ and □c□

```
        ┌---->----┐ ■true■          ┌------------->-----------┐
        |           :               |                          V
□(  a   |   b   |:  c   |   d   |:  e   |   f   |   g   )□
    :                 | |
  ■false■             └>┘
```

Fig.4.2

possesses the value ■true■. Note that in this case the •condition• □e□ is not elaborated.

A similar extension [R.9.4.b] exists, whereby the symbols □then if□ may be replaced by □thef□ if the corresponding □fi□ is elided, but this extension may not be so useful. Because of it,

□if a thef b then c else d fi□

has the same meaning as

□if a then if b then c else d fi fi□ .

In other representations we have that

□(a |: b | c | d)□

means the same as

□(a |(b | c | d))□ ,

where the symbol □|:□ is used as a representation of the •then-if-symbol•. It is also a representation of the •else-if-symbol• but no confusion can arise. It is worth noting that, provided the elaboration of □a□ and □b□ involves no side effects, the effect of □(a |: b | c)□ is the same as that of □(a and b | c)□, but the former may be faster.

In the strict language the •conditional-clause• always contains an •else-clause•; however, another extension [R.9.4.a] allows □else skip fi□ to be replaced by □fi□, so that the clause

□if p then go to l else skip fi□ ,

may be written

□if p then go to l fi□ .

In the •assignation• □x := (a > 0 | sqrt(a))□ therefore, some undefined real value will be assigned to □x□, if the value of □a□ is not positive. This occurs because the □skip□ will be made to possess some undefined real value [R.8.2.7.2.a].

4.3 Case clauses

A case clause is also an extension of a •conditional-clause•, intended to allow for efficient implementation of a

certain kind of •conditional-clause• which may appear frequently. The •clause•
□if i = 1 then x elsf i = 2 then y elsf i = 3 then z else a fi□
may be written

□case i in x, y, z out a esac□ ,

or in another representation,

□(i | x, y, z | a)□

[R.9.4.c,d]. The flow of control in such a •clause• is indicated

```
                r---->----,-----,-----,----------,
                |         |     |     |          |
                |        •1•   •2•   •3•         |
                |         |     |     |          |
   □case        i    in   x,    y,    z   out  a   esac□
                          |     |     |          |     |
                          L-----L-----L----------L-->--J
```

Fig.4.3

in figure 4.3. Observe that □(i | x | a)□ is not a case clause for case clauses contain at least two •unitary-clauses• between the □in□ and the □out□.

If the reader is now confused over the use of certain symbols, the difficulties can be cleared away by observing that each of the symbols, •if-symbol, then-symbol, else-symbol• and •fi-symbol• has more than one representation. The representations are [R.3.1.1.a]:

•if-symbol•	□(	if	case□	,
•then-symbol•	□|	then	in□	,
•else-symbol•	□|	else	out□	,
•fi-symbol•	□)	fi	esac□	.

This means that the case clause given above might be written

□case i then x, y, z | a fi□ ,

and, though most humans would find this difficult to read, the computer should not.

Because □|□ is a representation of the •else-symbol• and □)□ a representation of the •fi-symbol•, the case clause □(i | x, y, z | skip)□ may be written □(i | x, y, z)□, using the extension [R.9.4.a] already mentioned above. Note then, that in the •assignation• □x := (i | 1.2, 3.4)□, some undefined real value will be assigned to □x□ if □i□ is not •1• or •2•, but in the •assignation• □(i | x, y) := 3.4□, there may be no detectable effect [R.8.3.1.2.c] if the value of □i□ is not •1• or •2•.

There are further extensions of the case clause involving •conformity-relations• [R.9.4.e,f,g], but we shall delay discussion of these until •conformity-relations• themselves have been explained.

4.4 Repetitive statements

Repetitive statements, such as

□for i to n do s□ ,

are not mentioned in the syntax of the language. Such statements are in the extended language [R.9.3.a,b] and can stand in the syntactic position of •unitary-statements• [R.6.0.1.c]. A simple example of a repetitive statement is

```
                 □to 10 do random□                    .
```

It is defined to be the equivalent of the •unitary-statement•

```
      □begin int j := 1 ;
      m: if j ≤ 10 then random ; j +:= 1 ;
         go to m fi
      end□                                          ;
```

however, the reader who consults the Report [R.9.3.a] will find that the above is a gross simplification and that there are many details, such as increments other than □1□, which must also be considered.

A more illustrative example is

```
        □for i from a by b to c do x[i] := sqrt(i)□        .
```

This is defined to be the equivalent of

```
      □begin int j := a, int k = b, l= c ;
      m: if ( k > 0 | j ≤ l |: k < 0 | j ≥ l | true )
         then int i = j ; x[i] := sqrt(i) ; j +:= k ;
         go to m fi
      end□                                          ;
```

however, this is still not the complete story and may give the wrong effect if it is considered to be the equivalent of the above repetitive statement in a •serial-clause• in which operations have been redeclared. With this remark in mind the reader should now examine the extensions, as given in the Report [R.9.3.a,b], to notice how all eventualities have been covered.

There are essentially two repetitive statements. They are:

```
            □for i from a by b to c while d do e□
```

and

```
            □for i from a by b while d do e□          .
```

These differ in that the first form contains a □to□ and the second does not. In both forms □from 1□ or □by 1□ or □while true□ may be elided [R.9.3.c (the statement of this extension is more precise in the Report)] and if the •identifier• □i□ does not appear in the •unitary-clause• □e□, or the •serial-clause• □d□, then □for i□ may be elided. Notice that the control •variable• (□j□ in the above example) of a repetitive statement is hidden from the programmer, so that he may make no assignment to it. Also notice that the use of □for i□ means that □i□ is, for each elaboration of □d□ and □e□, an •integral-constant• declared within a range which contains both □d□ and □e□. Consequently no assignment may be made to □i□. This fact was used in the examples given above.

Before leaving repetitive statements, we should observe that the •unitary-clauses• □a, b□ and □c□ are elaborated collaterally [R.6.2.2.a] and once only, which means, in particular, that a change in the step size □b□ or in the upper bound □c□, after the initial elaboration, will not affect the further elaboration of the repetitive statement.

4.5 Closed clauses

Some examples of •closed-clauses• are □(x + y)□, □(((a)))□ and □begin real x, y ; read((x, y)) ; print(x + y) end□. Note that either □()□ pairs (•packs•)[1] or □begin end□ pairs (•packages•) may be used, but that □(x + y end□ is not a •closed-clause• [R.6.3.1.a, 1.2.5.i, 3.0.1.h,i]. A simplified syntax of the •closed-clause• is

closed clause : open symbol, serial clause, close symbol ;
    begin symbol, serial clause, end symbol.

but the strict syntax of the Report, involving the use of •pack• and •package•, should be consulted [R.6.3.1.a]. A simple parse of the •closed-clause•, □(x + y)□, is shown in figure 4.5. Since

```
                      closed-clause
                            |
                   serial-clause-pack
                            |
          ┌-----------------+------------------┐
          |                 |                  |
     open-symbol      serial-clause      close-symbol
          |                 |                  |
          ⊥               --⊥--                ⊥
          □(              x + y                )□
```

Fig.4.5

the elaboration of a •closed-clause• is that of its •serial-clause•, there is little else to be said about •closed-clauses•, except perhaps, that a •closed-clause• is a •primary• (as is a •conditional-clause•) and that the •serial-clause• of a •closed-clause• is a •range• [R.4.1.1.e] and therefore plays a role in the identification of •identifiers• [R.4.1,2,3]. The former means that, for example, □a * begin b + c end□ is an acceptable •formula•, though most programmers would prefer to write it as □a * (b + c)□.

4.6 Collateral phrases

A •collateral-clause• [R.6.2.1.b,c,d,f] consists of two or more •unitary-clauses• (•units• [R.6.1.1.e]) separated by •comma-symbols• and enclosed between a □()□ pair (•pack•) or a □begin end□ pair (•package•). An example of a •collateral-clause• is □(1.2, 3.4)□. It may be used in the situations □[1:2]real x1 = (1.2, 3.4)□ or □compl z = (1.2, 3.4)□. In the first situation the value of the •collateral-clause• is a row of values, whereas in the second it is a structure. Thus, the semantic interpretation of a •collateral-clause• may be determined by its context. Notice that □(a)□ is not a •collateral-clause•, for, otherwise, there would be an ambiguity in that □(a)□ is already a •closed-clause•.

----

(1) Strictly speaking, "pack" and "package" are protonotions but not paranotions [R.1.1.6], so you will not find them used in the semantic text of the Report.

A simplified syntax of the •collateral-clause• is

```
collateral clause :
   open symbol, unit list proper, close symbol ;
   begin symbol, unit list proper, end symbol.
unit list proper :
   unitary clause, comma symbol, unitary clause ;
   unit list proper, comma symbol, unitary clause.
```

but the strict syntax is rather more complicated [R.6.2.1] since it must take care of the two situations hinted at above together with the balancing of modes [R.6.1.1.g, 6.2.1.e, 6.4.1.d], an interesting topic in itself, which should be postponed. A simple parse of a •collateral-clause• is shown in figure 4.6. If a •collateral-clause• is used as a •statement•, then it may be preceded by a •parallel-symbol•, represented by □<u>par</u>□, if parallel processing is intended [R.10.4].

```
                       collateral-clause
                               |
        ┌----------------------+----------------------┐
        |                      |                      |
   open-symbol          unit-list-proper         close-symbol
        |                      |                      |
        |               ┌------┴------┬------┐        |
        |               |             |      |        |
        |       unit-list-proper      |      |        |
        |               |             |      |        |
        |         ┌-----+-----┐       |      |        |
        |         |     |     |       |      |        |
        |       unit    |   unit      |    unit       |
        ⊥         ⊥     ⊥     ⊥       ⊥      ⊥        ⊥
       □(        1.2    ,     x       ,      y        )□
```

Fig.4.6

The important feature of a •collateral-clause• is that the order of elaboration of the •unitary-clauses• of the •unit-list-proper• is undefined[R.6.2.2.a]. This means, for example, that the value of □(<u>int</u> i := 0, j := 0, k:= 0 ; (i := j+1, j := k+1, k := i+1))□ could be that of any one of several rows of three integral values, such as that of □(1, 1, 1)□ or □(2, 1, 3)□, etc.

In like manner, a •collateral-declaration• consists of two or more •unitary-declarations• separated by •comma-symbols•, with the order of elaboration undefined. This means, for example, that the •collateral-declaration• □<u>int</u> n := 10, [1:n]<u>real</u> x1□ may, or may not, have the effect perhaps intended by the programmer. The object □<u>int</u> n := 10 ; [1:n]<u>real</u> x1□ would make more sense. Observe that a •collateral-declaration• is not enclosed by an •open-symbol, close-symbol• pair or •begin-symbol, end-symbol• pair, i.e., neither a •pack• nor a •package•.

4.7 Serial clauses

•Serial-clauses• are put together from •unitary-clauses• with the aid of •go-on-symbols, labels, completion-symbols• and •declarations• [R.6.1.1]. We shall examine this construction by starting from the simplest constituents. It is expedient, as in the Report [R.6.1.1.e], to speak of a •unitary-clause• as a •unit•. For the convenience of our explanation, we introduce the notion •paraunit• (not in the Report), for a •unit• which may be preceded by zero or more •labels•. Thus

¤x := 3¤

is a •unit•, but for us,

¤x := 3¤

and

¤l2: x := 3¤

are both •paraunits•. The simplified syntax is then:

```
unit : unitary clause.
paraunit : unit ; label, paraunit.
label : label identifier, label symbol.
```

and although this is a slight deviation from the strict syntax of the Report, we shall have no essential difference when we are through.

A •clause-train• [R.6.1.1.h] is one or more •paraunits• separated by •go-on-symbols•. The following are therefore examples of •clause-trains•:

¤x := 3¤

¤l2: x := 3¤

¤l1 : y := 2 ; x := 3¤

¤open(myfile,"abc",tape8) ; restart : get(myfile,name)¤

[R.10.5.1.2.b, 10.5.2.2.b]. We may now add another simplified syntactic rule, viz.,

```
clause train : paraunit ;
   clause train, go on symbol, paraunit.
```

(cf., [R.6.1.1.h]). The semantics of a •clause-train• is simple. The elaboration of the •units• proceeds from left to right, i.e., in the normal sequential order, as in most programming languages.

A •suite-of-clause-trains• [R.6.1.1.f,g] consists of one or more •clause-trains• separated by •completers•, where a •completer• is a •completion-symbol•, represented by ¤.¤, followed by a •label•. The following are therefore examples of a •suite-of-clause-trains•:

¤x := 3¤

¤l1: y := 2 ; x := 3¤

¤(i > 0 | l1 | x := 1) . l1: y := 2 ; x := 3¤ .

A simplified syntax of a •suite-of-clause-trains• is

```
suite of clause trains : clause train ;
   suite of clause trains, completer, clause train.
completer : completion symbol, label.
```

[R.6.1.1.f,g]. The semantics of a •suite-of-clause-trains• is dramatically different. The effect of the •completer•, as opposed to the •go-on-symbol•, is to force the completion of the elaboration of the •serial-clause• containing it and to yield, as the value of that •serial-clause•, the value of the •unit•

most recently elaborated. In the last example above, if the value of ¤i¤ is ▪-1▪, then the value of the •serial-clause• is the value of ¤x := 1¤ and the •clause-train• ¤y := 2 ; x := 3¤ is not elaborated; otherwise, it is the value of ¤x := 3¤. In fact, the effect is the same as that of ¤(i > 0 | y := 2 ; x := 3 | x := 1)¤. One might think that any •suite-of-clause-trains• may be re-written as a •conditional-clause• (suggesting redundancy in the language) and though this may be true in theory, the example
¤<u>for</u> k <u>to</u> <u>upb</u> s <u>do</u> (c = s[k] | i := k ; l) ; <u>false</u> . l: <u>true</u>¤ [R.10.5.1.2.n], shows that the •completer• is indeed a useful tool in practical programming. It plays a similar role to that of the return statement in PL/I or FORTRAN, though in these languages the return statement applies only to procedures (subroutines, functions).

A •serial-clause• [R.6.1.1.a] is, roughly speaking, a •suite-of-clause-trains• preceded by zero or more •declarations• and/or •statements• but these •statements• may not be labelled. Examples of •serial-clauses• are

```
                        ¤x := 3¤
                   ¤l1: y := 2 ; x := 3¤
      ¤( r > .5 | l1 | x := 1 ) . l1: y := 2 ; x := 3¤
 ¤real x, y ; ( r > .5 | l1 | x := 1 ) . l1: y := 2 ; x := 3¤
                 ¤r := random ; real x, y ;
      ( r < .5 | l1 | x := 1 ) . l1: y := 2 ; x := 3¤
```

and

```
            ¤real r ; r := random ; real x, y ;
      ( r < .5 | l1 | x := 1 ) . l1: y := 2 ; x := 3¤         .
```

A simplified syntax of •serial-clause• is:

```
  serial clause : suite of clause trains ;
    declaration prelude sequence, suite of clause trains.
  declaration prelude sequence : declaration prelude ;
    declaration prelude sequence, go on symbol,
    declaration prelude.
  declaration prelude : single declaration, go on symbol ;
    statement prelude, single declaration, go on symbol.
  single declaration :
    unitary declaration ; collateral declaration.
  statement prelude : unit, go on symbol ;
    statement prelude, unit, go on symbol.                    .
```

The rules just given are close to those in the Report [R.6.1.1.a,b,c,d]. The reader should now examine the rules of the Report to observe how the metanotions •MODE• and •SORT• have been carried through the syntax and that balancing of modes may be necessary when •completers• are present [R.6.1.1.g].

The elaboration of a •serial-clause• begins with the protection [R.6.0.2.d] of all •identifiers• and •indications• declared within it. The protection is done to ensure that, for example, all •identifiers• declared within a •serial-clause•, cannot be confused with similar •identifiers• outside it. Users of ALGOL 60 or PL/I will recognize this as the matter of scope, but the reader is warned that the word "scope" has a wider meaning in ALGOL 68 [R.2.2.4.2].

4.8 Program example

The •procedure-denotation• which follows possesses a routine which expects a row of integral values which are the coefficients of the polynomial

□a[0]*x**n+a[1]*x**(n-1)+ ... +a[n]□ .

It then finds all the rational linear factors (those of the form p*x-q, where p and q are integral). It delivers an integral result, which is the degree of the residual polynomial, whose coefficients remain in □a□. The number of linear factors is in □r□, any constant factor is in □c□ and the factors □u[i]*x-v[i]□ are found in the row of integral values □u□ and □v□ (1).

```
□proc factors = (ref[0:]int a ¢the coefficients of the given
   polynomial¢, ref int r ¢for the number of rational linear
   factors¢, c ¢for the constant factor¢, ref[ ]int u, v ¢for
   the linear factors (u[i]*x-v[i]), 1≤i≤r¢) int :
  begin int n := upb a ¢the degree of the given polynomial¢;
  r := 0 ; c := 1 ; ¢initialization¢
  while a[n] = 0 do ¢remove the common power of x¢
    begin u[r +:= 1] := 1 ; v[r] := 0 ; n -:= 1 end ;
  for p to abs a[0] do
    if a[0] ÷: p = 0
    then  ¢p divides a[0]¢
    int q := 0 ; while (q := abs q + 1) ≤ abs a[n] do
      if a[n] ÷: q = 0
      then ¢q divides a[n]¢
      int f, g  ¢for temporary storage later¢ ;
        if q ≠ 1 and p = 1
        then ¢look for a constant factor¢
        MORE : for j from 0 to n do
          if a[j] ÷: q ≠ 0
          then ¢q does not divide a[j]¢
          go to NOCONSTANT fi ;
        ¢remove the constant factor q¢
        for j from 0 to n do a[j] ÷:= q ; c *:= q ;
        ¢q may be a multiple factor so¢ go to MORE
        fi ¢end the search for a constant factor¢ ;
      NOCONSTANT : ¢try (p*x-q) as a linear factor¢
      g := 1 ; f := a[0] ; ¢try x = q / p¢
      for i to n do f := f * q + a[i] * (g *:= p) ;
        if f = 0
        then ¢(p*x-q) is a factor¢
        u[r +:= 1] := p ; v[r] := q ; n -:= 1 ;
        for i from 0 to n do ¢compute the residual¢
          begin ref int ai = a[i] ;
          ai := f := (ai + f * q) ÷ p end ;
        ( n = 0 | REDUCED | NOCONSTANT )
        else ¢if we are here, then (p*x-q) is not a factor
        so try (p*x+q)¢  ( (q := - q) < 0 | NOCONSTANT )
```

----

(1) This procedure is derived from algorithm number 75 in the Communications of the Assoc. for Computing Machinery, Vol 5(1962)48, revised by J.S.Hillmore Vol 5(1962)392 and further revised for the version given above.

```
        fi ¢end else part¢
      fi ¢end iteration on q¢
    fi ¢end iteration on p¢ ;
  REDUCED : (n = 0 | c *:= a[0] ; a[0] := 1 ) ;
  ¢the degree of the residual polynomial is¢ n
  end□
```

In the range of the •declaration• □[0:3]int a1 := ([ ]int : (1, -1, 2, -2))[@0], int k, number, constant, [1:3]int m1, n1□, a •call• of the above •procedure• might be

□k := factors(a1,number,constant,m1,n1)□ ,

whereupon we should have □k = 2, a1 = ([ ]int :(1, 0, 2, 0))[@0], number = 1, constant = 1, m1 = (1), n1 = (1)□, corresponding to the factoring

□x**3 - x**2 + 2*x - 2 = (x**2 + 2)(x - 1)□ .

Observe that in the •clause• □begin ref int ai = a[i] ; ai := f := (ai + f * q) ÷ p end□, the programmer may optimize his subscript calculation, rather than leave this delicate matter to the whim of the compiler writer. On a non-optimizing compiler, of which there may be many, this possibility has clear dividends. Note also the •assignation• □f := f * q + a[i] * (q *:= p)□, which replaces two statements in the original ALGOL 60 version.

Review questions

4.1 Conditional clauses

a) What is the value of □(0 < 0 | 1 i 2 | 3)□?
b) Is □if x < 0 then go to error□ a •conditional-clause•?
c) Is □(x > 0 | a | b) of c□ a •selection•?
d) Is □a of (x > 0 | b | c)□ a •selection•?
e) Is □(r | m | n) < (s | i | j)□ a •formula•?
f) Is □if x > 0 then x else y fi := 3.14□ an •assignation•?

4.2 Simple extensions of conditional clauses

a) What is the value of □(1 < 2 |: 3 < 4 | 5 | 6)□?
b) What is the value of □(1 > 2 |: 3 < 4 | 5 | 6)□?
c) What is the value of □(true | 5 | 4) + (false | 3 | 6)□?
d) Simplify the following using the extensions:
   □if p then a else if q then if r then b else c fi else skip fi fi□.
e) Remove the extensions in □(a |: b | c |: d | e)□.

4.3 Case clauses

a) Is □(1 | 2 | 3)□ a case clause?
b) What are all the representations of the •if-symbol•?
c) What is the value of □(2 | 3, 4, 5 | 6)□?
d) What is the value of □(0 | 3, 2, 1 | 2)□?
e) Is □(2 | a, b, c) of d□ a •selection•?

4.4 Repetitive statements

In each of the following, is the object a repetitive statement, and if so, how many times is the •unitary-clause• □e□ elaborated?

a) □for i do e while (i < 9)□
b) □for i to 10 by 2 do e□
c) □do e□
d) □while false do e□
e) □to 0 do e□

Comment on the scopes of □i□ in the following:

f) □for i from 1 by 1 to 10 do i := 2 * i + 1□
g) □int i := 5 ; for i from 1 by i to i -:= 1 do a[i] := i * i□.

4.5 Closed clauses

a) Is □(x / y)□ a •closed-clause•?
b) Is □(p | 1)□ a •closed-clause•?
c) Is □(x := 1 ; y := 2 ; z) := 3□ an •assignation•?
d) Is □if x := y ; z := 2 fi□ a •closed-clause•?
e) Is □begin x := 1 ; y := 2)□ a •closed-clause•?
f) Is □(a ; b , c)□ a •closed-clause•?

4.6 Collateral phrases

a) Is □(x)□ a •collateral-clause•?
b) Is □(1 ; 2 , 3)□ a •collateral-clause•?
c) Is □(1 | 2 , 3)□ a •collateral-clause•?
d) What is the value of □("a", "b", "c") + ("d", "e")□?
e) Is it possible that the value of

    □(int i := 2, j := 3 ; (i +:= j, j +:= i))□

might be the same as that of □(7,5)□?

4.7 Serial clauses

a) Is □x□ a •serial-clause•?
b) Is □(p | x | 1) . l: h□ a •serial-clause•?
c) Is □3.e□ a •serial-clause•?
d) Is □(x := 1 ; y := 2)□ a •clause-train•?
e) Rewrite the following •conditional-clause• as a •serial-clause• containing a •completer•.

    □(x or y | n := 1 ; r | n := 2 ; s)□

4.8 Program example

a) How many occurrences of a •conditional-clause• are there in this •particular-program•?
b) What is the mode of □a□?
c) What is the mode of □ai□?
d) How many occurrences of a •closed-clause• are there following the •label• □NOCONSTANT :□?
e) How many occurrences of a •collateral-clause• are there?

5 Routine denotations and calls

5.1 The parameter mechanism

We begin this chapter with a simple illustrative example of the •declaration• and use of a nonsense •procedure• □up□ which has two •parameters• □a□ and □b□, and whose effect is to increment the •real-variable• □a□ by the •real-constant• □b□. In ALGOL 68 the defining occurrence of such a •procedure• is in the •identity-declaration•

□proc up = (ref real a, real b) : a +:= b□

and its •call• might be □up(x, 2)□ or □up(x1[i], y)□. In ALGOL 60, a procedure with similar effect would be declared by

□procedure up(a, b) ; value b ; real a, b ; a := a + b□

and its procedure call might also be □up(x, 2)□ or □up(x1[i], y)□. In PL/I the same procedure might be written

UP : PROC(A, B) ; A = A + B ; END ;

and its call, CALL UP(X,2E0) or CALL UP(X1(I), (Y)). In FORTRAN it would be

```
      SUBROUTINE UP(A, B)
      A = A + B
      RETURN
      END
```

with call, CALL UP(X, 2.0) or CALL UP(X1(I), Y).

We have described this procedure in more than one language in order that its intended effect should be clear to all. The reader will notice that we are concerned with that which, in ALGOL 60 terminology, is known as a "call by name" and a "call by value". This has become the accepted way of describing the fact that in the •call• □up(x, 2)□, □x□ is passed by name to □a□ and □2□ is passed by value to □b□. The manner in which values are passed at the time of a •call• is generally known as the "parameter mechanism".

We shall not describe here the various parameter mechanisms in other languages, except to say that the student is likely to find this to be the most confusing and perplexing subject area in the study of programming languages. Each language has its own philosophy and usage, with treacherous traps for the unwary. We hope to show, in this chapter, that the parameter mechanism of ALGOL 68 is exceptional in its clarity, encouraging the programmer to state precisely the mechanism he wishes to use, rather than to rely upon the conventions of a given language or the whim of an implementer. There are essentially no new ideas involved beyond those which we have encountered in earlier chapters. A thorough understanding of the •identity-declaration• is all that is needed. The reader may soon wish to forgive us for spending so much time on the explanation of it in chapter 2. The ALGOL 68 parameter mechanism is defined in terms of a logical application of the •identity-declaration• to that internal object, known as a "routine", which is the value possessed by a •routine-denotation•.

5.2 Routine denotations

The object

¤((ref real a, real b) : a +:= b)¤

is an example of a •routine-denotation• [R.5.4.1.a] and is essentially what stands on the right of the •equals-symbol• in the •declaration• of ¤up¤ given in section 5.1 above. One may notice that the enclosing symbols ¤(¤ and ¤)¤ have been omitted in section 5.1, but this is only because of an extension [R.9.1.d] which allows such omission in this situation. A •routine-denotation•, like any other •denotation•, possesses a value, a routine, which is an internal object. This internal object is a certain sequence of symbols, easily derived [R.5.4.2] from the •denotation•. For example, the routine possessed by

¤((ref real a, real b) : a +:= b)¤

is

•(ref real a = skip, real b = skip ; a +:= b)•

and it is important to notice that it has the shape of a •closed-clause•, in which each of the •parameters• ¤a¤ and ¤b¤ forms part of an •identity-declaration•.

As we have seen in section 2.5, an •identity-declaration• causes the value of its •actual-parameter• (the part to the right of the •equals-symbol•) to be possessed by the •identifier• of its •formal-parameter• (the •identifier• to the left of the •equals-symbol•). This means that in the •identity-declaration•

¤proc up = ((ref real a, real b) : a +:= b)¤ ,

the •identifier• ¤up¤ is made to possess the routine

•(ref real a = skip, real b = skip ; a +:= b)• .

Figure 5.2 shows a simple parse of this •identity-declaration•. The •routine-denotation• is shown at 1 and the routine which it possesses at 2. After the elaboration of the •identity-declaration•, the •identifier• ¤up¤, possesses the same routine

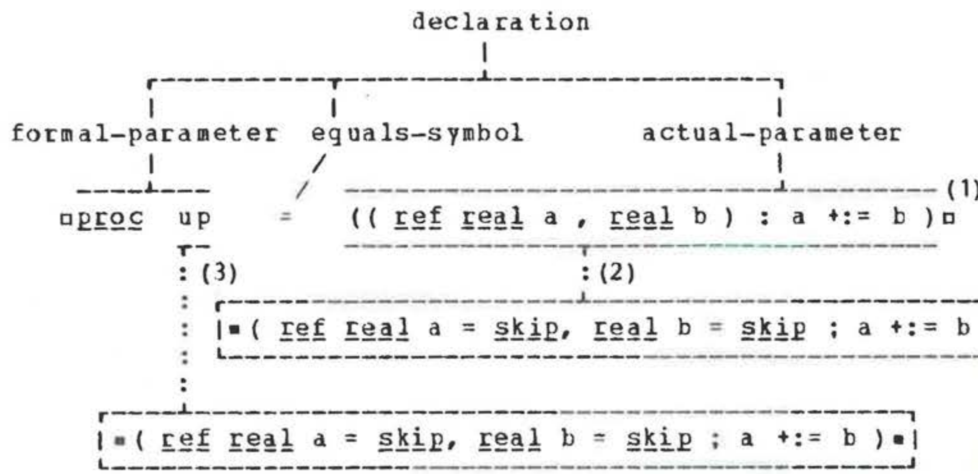


Fig.5.2

(see figure at 3). The elaboration of the •call• □up(x, 2)□ is now easy to describe. Its effect is to replace the two □skip□s, in a copy of the routine, by □x□ and □2□ respectively and then to elaborate the resulting external object

□(ref real a = x, real b = 2 ; a +:= b)□

as if it were a •closed-clause• standing in the place of the •call• □up(x, 2)□.

It is perhaps now clear why the left part of an •identity-declaration• is known as its •formal-parameter• and the right part as its •actual-parameter•, for these are precisely the roles which they play in the parameter mechanism. Not only does the •identity-declaration• play a central role in such a mechanism, but its power, which the implementer of any language must of necessity provide, is placed in the hands of the programmer to use as he sees fit. Thus, □ref real x1i = x1[i]□ might usefully be used to optimize address calculation while working with the vector □x1□. An example might be

□x1i := 3 * x1i + 2 * x1i ** 2□

rather than

□x1[i] := 3 * x1[i] + 2 * x1[i] ** 2□ .

5.3 More on parameters

It is perhaps worth dwelling on the name-value relationship created by the parameter mechanism for the example in section 5.1. The •closed-clause• which is elaborated as a result of the •call• □up(x, 2)□ is

□(ref real a = x, real b = 2 ; a +:= b)□

and the elaboration of the •collateral-declaration• which follows its •open-symbol• results in the relationships depicted

```
□ref  real  a  =  x ,  real  b  =  2□
            :     :          :     :
            o     o          :     :
           o o(1)o o         :     :
            o     o          :     :
            └─>┬<─┘          : (2) :
          ┌────┴────┐    ┌───┴───┐ ┌┴───────┐
          │         │    │  ▪2▪  │ │  ▪2▪   │
          └─────────┘    └───────┘ └────────┘
```

Fig.5.3.a

in figure 5.3.a. During the elaboration of the •call• □up(x, 2)□, □a□ possesses the same name as that possessed by □x□ (see figure 5.3.a at 1), and □b□ possesses the same value as that possessed by □2□ (see the figure at 2). This means that the •formula• □a +:= b□ has the same effect as if it were written □x +:= 2□. Both □a□ and □x□ have a mode which begins with •reference-to•, a requirement of the left •operand• of the •operator• □+:=□ [R.10.2.11.e]. Note also that if the •call• were □up(x, y)□, then the •closed-clause• would contain the •declaration• □real b = y□ and this would involve a dereferencing of □y□, depicted in figure 5.3.b at 1. Observe, in this figure, that □y□ , considered as an •identifier•, possesses a name of mode •reference-to-real• (see 2) but considered as an •actual-parameter•, it possesses a value of mode •real• (see 3). The coercion occurs at 1. We may say, in general, that if a •parameter• □a□ is considered as a •variable• referring to a value of mode specified by □m□, e.g., if an assignment is to be made to □a□, then the •formal-parameter• should be □ref m a□,

```
                              identity-declaration
                                       |
         ┌-----------------------------┴----------------------------┐
         |                             |                            |
 formal-real-parameter          equals-symbol      actual-real-parameter
         |                             |             :              |
   ┌-----┴---------┐                   |             : strong-real-base
   |               |                   |             :              |
 formal-real-   real-mode-             |             :     (coercion)  (1)
   declarer     identifier             |             :              |
     |              |                  |             :     reference-to-
     |              |                  |         (3) :       real-base
   --┴--            ┴                  ┴             :              ┴
   □real            b                  =                           y□
                    :                                :              :
               ┌----┴----┐                      ┌----┴---┐          o
               | ■3.14■  |                      | ■3.14■ |---<--o o (2)
               └---------┘                      └--------┘          o
```

Fig.5.3.b

but if □b□ is used only as a •constant• of mode □m□, then the •formal-parameter• may be □m b□.

5.4 The syntax of routine-denotations

A •routine-denotation• consists of a •formal-parameters-pack• followed by a •cast•, both together enclosed between the symbols □(□ and □)□. Thus in

□((ref real a, real b) : a +:= b)□

the object □(ref real a, real b)□ is the •formal-parameters-pack• and □: a +:= b□ is the •cast•. A simplified syntax of a •routine-denotation• is

routine denotation :
  open symbol, formal parameters pack, cast, close symbol.
formal parameters pack :
  open symbol, formal parameter list, close symbol.
formal parameter list : formal parameter ;
  formal parameter list, gomma, formal parameter.
gomma : go on symbol, comma symbol.

but the strict syntax [R.5.4.1] contains metanotions which ensure that the number and the modes of •parameters• in •calls• match those in the •routine-denotation•. Figure 5.4 shows a simple parse of a •routine-denotation•. We have already alluded, in section 3.7, to the fact that •actual-parameters• in a •call• may be separated by either a •go-on-symbol• or by a •comma-symbol•. Now that we have seen that the elaboration of a •call• amounts to the elaboration of a •closed-clause• in which the

•formal-parameters• of the •routine-denotation• become transformed into •identity-declarations•, it is at once apparent that a •comma-symbol• separating •formal-parameters• becomes a •comma-symbol• of a •collateral-declaration•. This means that the •parameters• are elaborated collaterally. The •go-on-symbol•, on the other hand, would result in •declarations• which are elaborated serially. To take a specific example, the

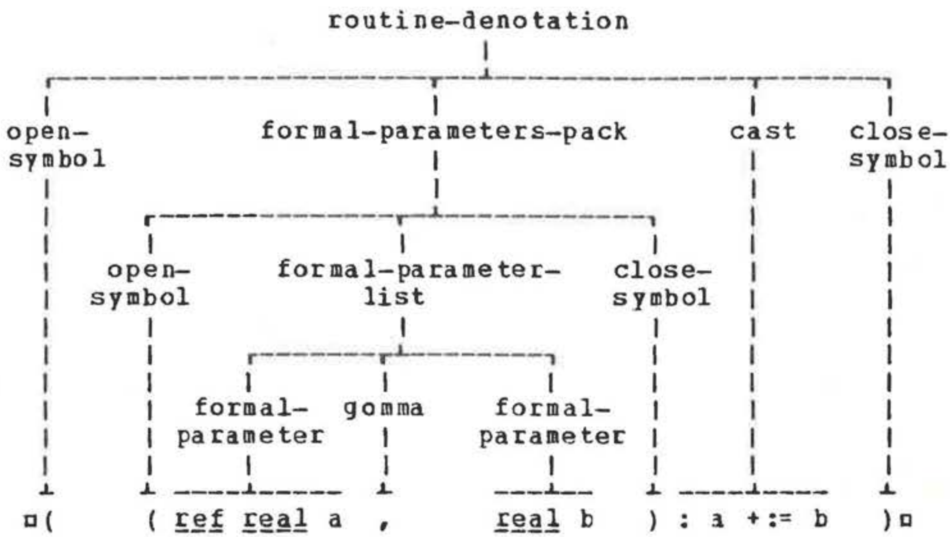


Fig.5.4

•formal-parameters-pack•

□(int n, [1:n]real u)□

may be transformed into

□int n = 10, [1:n]real u = x1 ;□ ,

but the •formal-parameters-pack•

□(int n ; [1:n]real u)□

may be transformed into

□int n = 10 ; [1:n]real u = x1 ;□ ,

which is more useful since its elaboration is well defined. The particular choice of the •gomma• which separates •formal-parameters• is therefore of significance but that which separates the •actual-parameters• of a •call• has no semantic significance.

The semantics of a •routine-denotation• [R.5.4.2] tells us how the routine which it possesses is obtained. The essential points are, that an •equals-symbol• followed by a •skip-symbol• is inserted after each •formal-parameter•, that the •open-symbol• which begins the •formal-parameters-pack• is deleted and that its •close-symbol• is changed into a •go-on-symbol•. The more precise statement in the Report [R.5.4.2] should be studied.

A further example of a •routine-denotation• is

□((real x)real : random * x)□ ,

where the second occurrence of □real□ (part of the •cast•)

indicates that the routine is to deliver a value of mode •real•. The example in section 5.1 delivers no value and therefore uses a •void-cast• (whose •virtual-declarer• is empty). Note that

¤<u>real</u> : random * 100¤

is not a •routine-denotation• despite the fact that it may appear in the •declaration•

¤<u>proc</u> <u>real</u> r100 = <u>real</u> : random * 100¤ ;

however, the coercion known as "proceduring" [R.8.2.3.1.a] enables the identifier ¤r100¤ to possess the routine

▪(<u>real</u> : <u>real</u> : random r * 100)▪ .

Actually, it is only necessary to write

¤<u>proc</u> <u>real</u> r100 = random * 100¤

and then the routine possessed by ¤r100¤ will be

▪(<u>real</u> : random * 100)▪ .

5.5 What happened to the old call by name?

In explaining the parameter mechanism of ALGOL 60, it is customary to consider an example something like

¤<u>procedure</u> upa(a, b) ; <u>value</u> b ; <u>real</u> a, b ;
<u>begin</u> i := i + 1 ; a := a + b <u>end</u>¤

and to explain that, in the scope of the fragments ¤<u>real</u> <u>array</u> x1 [1:10] ; <u>integer</u> i ; i := 1¤, the procedure call ¤upa(x1[i], 2)¤ will, to the astonishment of most, increment the value of ¤x1[2]¤ rather than that of ¤x1[1]¤. This is a result of the semantic description of procedure calls in ALGOL 60 [N.4.7.3.2] involving what is usually referred to as the "copy rule". In ALGOL 68 a routine which achieves a similar effect, for simple •variables• (not •slices•) passed to ¤a¤, is

¤<u>proc</u> upa = (<u>ref</u> <u>real</u> a, <u>real</u> b) : (i +:= 1 ; a +:= b)¤

but the •call• ¤upa(x1[i], 2)¤in the range of ¤[1:10]<u>real</u> x1 ; <u>int</u> i := 1¤, will increment the value referred to by ¤x1[1]¤ and not ¤x1[2]¤. Thus the passing of the •parameter• ¤x1[i]¤ by name, as it was known in ALGOL 60, is not achieved, in ALGOL 68, by using the •formal-parameter• ¤<u>ref</u> <u>real</u> a¤. The resulting •identity-declaration• ¤<u>ref</u> <u>real</u> a = x1[i]¤ is elaborated at the time of entry to the routine and the old copy rule of ALGOL 60 does not apply.

In the case of expressions and subscripted variables, this copy rule of ALGOL 60 amounted to the passing of a procedure body to the formal parameter and was used by a generation of instructors to impress students with the idea that ALGOL 60 is a nice language in which nice things can be done in a nice way. However, the niceties of it were often too subtle for the beginner, who thus fell into the trap of using a powerful device when it was not necessary for him to do so. We may now perhaps look back upon it as a design imperfection in ALGOL 60. There should have been a <name part> rather than a <value part> [N.5.4.1]. A language should be such that the least effort by the programmer calls up the simplest implementation schemes. If he wishes to use a more powerful scheme, then he should be made aware of it by the necessity for writing a little more in his source program.

To recapture the strange effect of the call by name of

ALGOL 60, the example mentioned above should appear as
□proc upb = (proc ref real a, real b) : (i +:= 1; a +:= b)□ , for then the first •declaration• arising from the •call• □upb(x1[i], 2)□ is □proc ref real a = x1[i]□. In this case the elaboration of □x1[i]□ occurs at the time of the deproceduring [R.8.2.2] of □a□ in □a +:= b□, and not at the time of parameter transfer. Thus □x1[2]□ is incremented and not □x1[1]□.

The occurrence of □x1[i]□ in □proc ref real a = x1[i]□ is another example of a •procedured-coercend• for □x1[i]□ is not a •routine-denotation•. Nevertheless, the •identifier• □a□ is made to possess the routine ■(ref real : x1[i])■ by a coercion known as proceduring [R.8.2.3].

5.6 Program example

The following algorithm finds all trees which span a non-directed graph □g□ (1). The edges radiating from node ■i■ in the graph are represented by bits in the i-th bits structure of the row-of-bits □g□. A set of nodes is also represented by bits of a bits structure, the j-th node being represented by the j-th bit, which is ■true■ if that node is present.

The set of nodes in the growing trees (saplings) is □s□. The edges in a family of saplings are recorded in □a□, which, like □g□, is of mode •row-of-bits•. The boundary of □s□ is the set □b□ of nodes neighbouring the nodes of □s□. Initially □s□ contains only node ■1■ and □b□ its neighbours, i.e., □g[1]□. The recursive routine □grow□ iterates over the nodes in □b□. For each node ■i■ in □b□ it finds all possible edges (new growth) from □s□ to node ■i■. This new growth is recorded in □a□ and removed from □g□. The node ■i■ is removed from the boundary □b□. The procedure □grow□ is then called recursively with the nodes of the saplings augmented by node ■i■ and the boundary augmented by neighbours of node ■i■.

Since the standard □bits width□ (or □long bits width□) may be larger than the number of nodes, a □mask□ is necessary to mask out the redundant bits when testing bit patterns.

If the number of nodes exceeds □bits width□, then the •mode-declaration• for □b□, in the first line, should be changed accordingly. If sufficient precision is then not available, one may use the mode •row-of-boolean•, with suitable declaration of the operations involved.

As an example, for the graph

1(2,3,4), 2(1,3), 3(1,2,4), 4(1,3) ,

the algorithm generates eight trees in four families

1(),	2(1),	3(1,2),	4(1,3)	(4 trees)
1(),	2(1),	3(4),	4(1)	(1 tree)

------

(1) Translated from Algorithm 354 by M. Douglas McIlroy. Comm. Assoc. Computing Machinery, Vol 12(1969) p. 511.

```
   1(),    2(3),    3(1),      4(1,3)      (2 trees)
   1(),    2(3),    3(4),      4(1)        (1 tree)
```

```
 ▫begin mode b = bits ¢or long bits, if necessary¢ ;
 proc trees = ([1:] b g ¢the given graph¢,
                proc([]b) f ¢the action for each family¢):
   begin int n = upb g ¢the number of nodes in the graph¢;
   [1:n]b a ¢the growing family, saplings¢;
   b t; b flips = t or ¬ t ¢all flips¢ ;
   b unit = ¬(flips up -1) ¢a flip followed by flops¢,
     mask = ¬(flips up -n) ¢for masking redundant bits¢;
   proc grow = (ref[1:n]b g ¢the residual graph¢,
                b s ¢the nodes of the saplings¢,
                ref b b ¢boundary of the saplings¢):
     if s ≥ mask
     then ¢the family is complete, so¢ f(a)
     else for i to n do
       if i elem b
       then ¢examine each node of the boundary¢
       b uniti = unit up(1-i) ¢only the i-th bit is flip¢;
       b := b and ¬ uniti ¢remove node i from the boundary¢ ;
       a[i] := g[i] and s ¢this is the new growth¢;
       g[i] := g[i] and ¬ s ¢remove the new growth¢;
       grow(loc [1:n]b := g ¢pass a copy of the residue¢,
            s or uniti ¢the family now includes node i¢,
            loc b := b or g[i] ¢the boundary is augmented by
            the neighbours of node i¢ ) ;
       ( ¬ g[i] ≥ mask | ¢we cannot move¢ out )
       fi ;
     out : skip
     fi ;
   ( n ≥ 1 | a[1] := ¬ flips ) ;
   grow(loc [1:n]b := g ¢start with a copy¢,
        unit ¢start with node 1¢,
        loc b := g[1] ¢the neighbours of node 1¢)
   end
 end▫
```

In the above, the procedure ▫grow▫ has two •calls•. The •call• preceding the final ▫end▫, which starts the whole process, and another recursive •call• within the •routine-denotation•. In both of these •calls•, notice that the first and third •parameters• must be •variables•. Moreover, new copies of these •variables• must be passed. A convenient way to do this is to use •local-generators•. The second •parameter• is a •constant•, and no assignment is made to it.

Review questions

5.1 The parameter mechanism

a) Is the following an •identity-declaration•?

▫real proc p = (real a) real : a * a▫

b) Is the following an •identity-declaration•?
□proc(real a)real p = a * a□?
c) Give a •declaration• for a •procedure• □r2□ which has no •parameters• and delivers a random real value between ■0■ and ■2■.
d) Give a •declaration• for a •procedure• □max□ with two •real-parameters• which delivers the larger of the two.
e) Give a •declaration• of a •procedure• □recip□ which accepts a •real-variable• and replaces it by its reciprocal.

5.2 Routine denotations

a) Is □ref real xy = x * y□ an •identity-declaration•?
b) What is the •formal-parameter• of [1:3]real x1 := (1, 2, 3)□?
c) If □p□ possesses the routine ■(real a = skip, real b = skip ; a * b)■, what •closed-clause• is elaborated by the •call □p(x+1, y)□?
d) What is the value possessed by the •denotation• □((real a) real : a * a)□?
e) What is the value possessed by the •denotation• □(int n, m ; ref[1:n]real a1) real : (n < m | a1[n] | a1[m])□?

5.3 More on parameters

In the reach of □real x := 1.2, y := 3.4□, what is the value of □p(x, y)□
a) in the reach of □proc p = (real a, b) : 1.1□?
b) in the reach of
□proc p = (real a, ref real b) real : (b +:= a ; b)□?
c) in the reach of
□proc p = (ref real a, b) ref real : (1 > 2 | a | b)□?
d) in the reach of □proc p =(ref ref real a, ref real b)real : a := b□?
e) in the reach of □proc p = ([ ]real a, b)real : b[1] - a[1]□?

5.4 syntax of routine denotations

a) Translate the following into ALGOL 68:
□procedure p(a, b) ; value a ; integer a, b ;
b := b * 2 * a□.

5.6 Program example

a) Is □unit□ a •constant• or a •variable•?
b) Why is a □ref□ not necessary in the •formal-parameter• □b s□?
c) Why is an •actual-parameter• □loc := g[i]□ used in the last •call•?
d) Why was □t□ not initialized?
e) If □n□ is ■3■ and □bits width□ is ■8■, what is the value of □mask□?

6 Coercion

6.1 Fundamentals

Coercion is a process whereby, from a value of one mode, is derived the equivalent value of another mode, e.g., the real value possessed by ¤2.0¤ is equivalent to [R.2.2.3.1.d] the integral value possessed by ¤2¤. Derivation of an equivalent value is usually accomplished automatically, i.e., by no conscious effort of the programmer. An example is

¤<u>real</u> x := 2¤

where the value possessed by ¤2¤ is of mode •integral•, but the value which is assigned must be of mode •real•. Such coercions are well known in other languages and are usually described semantically. In PL/I there are extensive tables [P.Part II, Section F] in which the programmer may find what action to expect given the attributes of a source and those of its target. Coercion in ALGOL 68 is described by means of the syntax, most of which is in section 8.2 of the Report.

The particular coercions which are elaborated are generally determined by three things, viz., 1) the a priori mode, 2) the a posteriori mode and 3) the syntactic position, or "sort". A •cast•, which was discussed in section 4.13, is a useful object in which to illustrate coercion, for that is usually its main purpose. We recall that a •cast• consists of a •declarer• followed by a •cast-of-symbol• followed by a •unitary-clause•, which is in a strong position. For example, in the •cast•

¤<u>real</u> : 2¤

the a priori mode of ¤2¤ is •integral•, the a posteriori mode of its •unitary-clause• is that specified by its •declarer•, viz., •real•, and the "sort" of its •unitary-clause• is "strong". The particular coercion called into play is "widening" from •integral• to •real• and is governed by a syntactic rule [R.8.2.5.1.a], whose details we will not now unravel.

6.2 Classification of coercions

There are eight different coercions. They are "dereferencing", as in

¤<u>real</u> : x¤ ,

"deproceduring", as in

¤<u>real</u> : random¤ ,

"proceduring", as in

¤<u>proc</u> <u>real</u> : x1[i]¤ ,

"uniting", as in

¤<u>union</u>(<u>int</u>, <u>bool</u>) : <u>true</u>¤ ,

"widening", as in

¤<u>real</u> : 2¤ ,

"rowing", as in

¤<u>string</u> : "a"¤ ,

"hipping", as in

¤<u>real</u> : <u>skip</u>¤

and "voiding", as in the •void-cast-pack•

¤(: p)¤ .

These coercions are classified into subsets as follows:

dereferencing and deproceduring are together known as "fitting"; these two together with proceduring and uniting are known as "adjusting"; and all eight are together known as "adapting". The reader will find that this terminology is used in the metanotions [R.1.2.3.k,l,m]. A diagrammatic scheme is shown in figure 6.2. Some of the above examples would not normally appear in useful programs. They are chosen for illustrative purposes.

COERCION TREE

```
strong......ADAPTED
                |------------------------------------------
                |           |          |          |          |
                |        widened     rowed     hipped     voided
firm........ADJUSTED
                |-------------------------------
                |             |                |
                |         procedured        united
weak........FITTED
                |----------------------
                |                     |
                |               dereferenced
soft.......deprocedured
```

Fig.6.2

6.3 Fitting

The result of dereferencing a name is to yield the value to which it refers. This has been touched upon already in section

```
                 strong-real-unit............
                        |        (2)         :
                 strong-real-base            :
                        |                    :
        strongly-dereferenced-to-real-base   :
                        | (3)                :
              reference-to-real-base         :
                        |                    :
        reference-to-real-mode-identifier    :
                        |                    :
                       □x□                   :
                        : (1)                :
                        o      ------        :
                       o o->-|      |......:
                        o      ------
```

Fig.6.3

2.12 and elsewhere. Figure 6.3 shows the parse of □x□ as a •strong-real-unit•. At 1, in the figure, □x□, as an •identifier•, possesses a name and envelops the mode •reference-to-real• and at 2, as a •unit•, □x□ possesses a real value and envelops the mode •real•. The coercion is shown at 3.

The result of deproceduring is the elaboration of a routine (without parameters), e.g., the •cast• □real : random□ forces the elaboration of the routine possessed by □random□ and delivers the next random real value as the value of the •cast•. Both dereferencing and deproceduring are classified together as "fitting" [R.1.2.3.m], and are the two coercions which occur most frequently.

6.4 Adjusting

Both proceduring and uniting, together with fitting (dereferencing and deproceduring) are known as "adjusting" and are so grouped because they can all occur in certain syntactic positions.

The result of proceduring is a routine. For example, the value possessed by the •cast• □proc real : x1[i]□ is the routine ▪(real : x1[i])▪. It may be recalled, from section 5.2, that a routine is syntactically similar to a •closed-clause• and that, in the case where there are no •parameters•, there are no •routine-denotations•. The proceduring coercion makes them unnecessary.

Uniting has only a syntactic effect. In the terms of the Report, the elaboration of a united •coercend• is the same as that of its pre-elaboration [R.1.1.6.i]. This means that no change of value is involved. Actually, an implementation will find it necessary, upon uniting, to attach to the value some record of its mode, so that this may be tested later, especially if a •conformity-relation• is involved, but the particular details of the implementation mechanism is not of concern to the programmer. He should, however, be aware that it probably occurs and thus not make use of united modes unnecessarily. The subject of unions is an advanced topic which we shall postpone to chapter 7. Uniting occurs, for example, in □union(int, bool) : true□.

6.5 Adapting

The coercions known as widening, rowing, hipping and voiding, together with adjusting are collectively known as "adapting" and form the set of all possible coercions in the language. These are so grouped because they can all occur in certain syntactic positions.

The effect of widening is to deliver a value of one mode which corresponds to a given value of another mode. One may widen from •integral• to •real• [R.8.2.5.1.a] and from •real• to complex [ibid. b]. Consequently, each of the following possesses the value ▪true▪:

□(real : 2) = 2.0□ ,
□(compl : 2) = 2.0 i 0.0□ .

One may also widen from bits to •row of boolean• [ibid. c] and from bytes to •row of character• [ibid. d]. If □bits width□ is ▪4▪, then □([ ]bool : 101)□ has a value which is that of □(false, true, false, true)□. Similarly, if □bytes width□ is ▪4▪, then

¤(string : ctb "abc") = "abc." ¤ possesses the value •true• (assuming that the ¤null character¤ [R.10.1.1] is "."). More than one coercion may be involved in one •cast•, e.g., ¤compl : i¤ requires first a dereferencing of ¤i¤ to yield an integral value, a widening of the value to •real• and another widening to complex.

The effect of rowing is to deliver a multiple value which is a row of zero or one elements. It occurs, for example, in ¤[ ]real :¤ and in ¤[ ]int : 2¤. The value in the first case is a row of zero elements, each of mode •real•. In the second case one obtains a row of one element of mode •integral•. Note that ¤[,]int : [ ]int : 2¤ involves two consecutive rowings which result in a one by one matrix. The same effect can be obtained by ¤[,]int : 2¤, since rowing is recursive [R.8.2.6.1.a]. The •cast• ¤[,]bool :¤ will deliver a boolean matrix with one row which has no columns. Note that when a constant is rowed, the result is a •constant• multiple value, but if a •variable• is rowed the result is a multiple •variable•. This effect is achieved syntactically by the metanotion •REFETY• in the rule for rowing [R.8.2.6.1.a]. Thus, ¤ref[ ]real : x¤ will have the effect of creating a new multiple value whose only element is ¤x¤ and the •identity-relation• ¤(ref[ ]real : x)[1] :=: x¤ possesses the value •true• no matter what value is referred to by ¤x¤. Of course, it is arranged [R.8.2.6.1.b] that an empty cannot be rowed to a •variable•, i.e., ¤(ref[ ]real :)¤ is syntactically invalid.

The coercion known as hipping takes care of the •skip•, ¤skip¤, the •nihil• ¤nil¤, and •jumps• like ¤go to novosibirsk¤. This coercion is somewhat different from the others in that, if it occurs, then no other coercions may take place. Both the •skip• and the •jump• may be coerced to any mode, but the •nihil• may be coerced only to a mode which begins with •reference-to•. The elaboration of a •skip• delivers some (undefined) value of the required mode, e.g., the value of ¤real : skip¤ is some real value. The value of a •nihil•, represented by ¤nil¤, is a unique name which refers to no value. This means that ¤(ref real : nil) :=: (ref real : nil)¤ is •true•, although ¤(ref real : skip) :=: (ref real : skip)¤ is unlikely to be(1). Observe that ¤(ref int : nil) :=: (ref real : nil)¤ is not an •identity-relation• because the modes of its •tertiaries• do not agree. Also, ¤(ref real : ref ref real : nil)¤ cannot be elaborated, since no dereferencing can be done on a •nihil• [R.8.2.1.2 Step 2]. The elaboration of a coerced •jump• is a jump except in a case like ¤(proc ¢void¢ : go to 1)¤, where the value delivered is a routine and the jump itself is not performed [R.8.2.7.2.b]. Note however that ¤(ref proc ¢void¢ : go to 1)¤ does not deliver a routine.

There remains one other coercion, viz., voiding. The effect of voiding is to discard whatever value is involved. Thus

----

(1) It will be interesting to try out some of the compilers on this point.

□(: 2)□ will not deliver the value •2•. The •void-cast-pack• □(: random)□ delivers neither a routine nor a real value, but causes □random□ to be elaborated (deprocedured) once, whereupon the real value delivered is discarded (see •NONPROC• [R.8.2.8.1.b]). This may indeed be just what the programmer desires. In the reach of □proc real p := random□, the □p□ in □(: p)□ is dereferenced, deprocedured and then voided. The •declaration• □proc ¢void¢ q = (: p)□, however, delays these coercions until □q□ is elaborated. He who can correctly perform the syntactic and semantic analysis of □proc real p := random ; proc ¢void¢ q = (: p) ; (: q) ; skip□, has no need of further advice concerning coercion.

6.6 Syntactic position

The coercions which may occur depend upon the syntactic position of an object in the •program•. There are four sorts of syntactic position, viz., strong, firm, weak and soft. In what has gone before, we have concentrated our attention on the •cast• because its •unitary-clause• is strong and in this position all coercions can occur; moreover, strong coercion is the main purpose of the •cast•. In firm positions only those coercions collectively known as adjusting are relevant. In weak positions fitting is relevant. A soft position permits only deproceduring (see figure 6.2).

Some examples of strong positions are •actual-parameters•, e.g., □2□ in □real x = 2□, •sources•, e.g., □2□ in □x := 2□, •conditions•, e.g., □x=y□ in □(x=y | 1)□ and •subscripts•, e.g., □i□ in □x1[i]□. In these positions the a posteriori mode (i.e., the mode after coercion), is dictated by the context. Examples of firm positions are •operands•, e.g., □x□ in □abs x□, and •primaries• of •calls•, e.g., □cos□ in □cos(x)□. Examples of weak positions are •primaries• of •slices•, e.g., □x1□ in □x1[i]□ and •secondaries• of •selections•, e.g., □cell□ in □next of cell□. Examples of soft positions are •destinations•, e.g., □x□ in □x := y□ and •tertiaries• of •identity-relations•, e.g., □x□ in □x :=: xx□. Figure 6.6.a shows an •assignation• in which many of these positions occur.

```
□( x :=: re of z | xx | x1[i] ) := sin( x + (real : pi) )□
   T            T   T-    T- T        -T-  T             T-
   S            W   S     W  S         F   F             S
        ---T---          --T--                  -----T-----
           O               O                         F
   -------T-------                          -------T-------
          S                                        S
 ---------------.---------------  -----------.-----------
                O                             S
      {S = strong,  F = firm,  W = weak,  O = soft}
```

Fig.6.6.a

It is clear that •operands• cannot be strong, for otherwise one could not determine which operation is to be performed in

□1 + 2□. Since both •operands• could be widened, is it addition of real values or addition of integral values? Because of this uncertainty, the coercions involved in •operands• must be restricted to those classed as adjusting. This is achieved by making •operands• firm [R.8.4.1.d,f]. The only coercions permitted for •operands• are therefore dereferencing, deproceduring, proceduring and uniting. In particular, since a •skip• can only be hipped and hipping can only occur in strong positions, we conclude that the object □skip + skip□ is not a •formula•.

We may recall that if a •variable•, say □x1□, is sliced, then the result, say □x1[i]□, is a •variable•. Similarly the •selection• □next of cell□ from the •variable• □cell□ is also a •variable•. This means that we need a position in which both deproceduring and dereferencing are permitted, but that dereferencing, in this position, must stop short of removing a final •reference-to• from the a priori mode. Remember that we may wish to write □x1[i] := 3.14□ or □next of cell := cell1□ and that the mode of a •destination• must begin with •reference-to•. Such a position is known as weak. It involves only those coercions known as fitting, with the special proviso concerning dereferencing.

Finally, in the •destination• of an •assignation•, e.g., □x□ in □x := y□, only deproceduring can be permitted and such a position is known as soft.

Note that the word "strong" is used in the sense of strongly coerced, so that a strong position indicates strength from outside and not strength from inside.

In the above we have considered the syntactic positions arising from the strict language only. The programmer, however, is generally more concerned with the extended language, for that is what he uses. It is therefore appropriate to examine the syntactic positions for constructs in the extended language. In particular, the repetitive statement [R.9.2], shown in figure 6.6.b, contains the objects □a, b, c, d□ and □e□, all of which are in a strong position. Note that □i□ is the •identifier• of an •identity-declaration• and is therefore not coerced. Its mode is •integral• (not •reference-to-integral•) and therefore

```
                strong-unitary-void-clause
                            |
     -----------------------------------------------------
     □for  i  from  a  by  b  to  c  while  d  do  e□
           T           T      T      T          T     T
     integral-     strong-unitary-       strong-   strong-
       mode-       integral-clause       serial-   unitary-
     identifier                          boolean-  void-
                                         clause    clause
```

Fig.6.6.b

no assignment may be made to it. Moreover, the value of this □i□

is unavailable outside of the •clauses• □d□ and □e□, no matter how the elaboration of the repetitive statement is completed. Also observe that the repetitive statement itself is strongly voided and therefore cannot deliver a value. This is traditional for several programming languages, so will be understood easily.

6.7 Coercends

Coercions are introduced at certain syntactic positions but are not carried out except upon •coercends•. For example, in □proc ref real p = (i < 9 | x1[i] | y1[i])□, the •conditional-clause• □(i < 9 | x1[i] | y1[i])□ is strong and the mode required is that specified by □proc ref real□. However, a •conditional-clause• is not a •coercend• itself. In fact, if the value of □i□ is •2•, then the routine possessed by □p□ is □(ref real : x1[i])□. It is therefore the •base• □x1[i]□ which is coerced and not the •conditional-clause• because a •base• is a •coercend•.

•Coercends• are easily distinguished and we have met them all before, although we have not, as yet, classified them as such. A •coercend• is either a •base•, e.g., □x1[i]□, a •cohesion•, e.g., □next of cell□, a •formula•, e.g., □abs x□ or a •confrontation•, e.g., □x := y□ [R.8.2.0.1.a, 1.2.4.a]. A certain set of coercions may be implied by the syntactic position (sort) of the object, but none of these coercions will be elaborated on that object unless it is a •coercend•. The sort is therefore passed to the •coercends• within the object. When a •coercend• is met, then all coercions implied by that syntactic position must be completely expended.

6.8 A significant example

Perhaps we should now look closely into the reason why

□proc ¢void¢ p = random□

is not an •identity-declaration•. The intention was, perhaps, □proc ¢void¢ p = (: random)□ or □proc real p = random□. First we must observe that no extension could have been applied since □random□ is not a •routine-denotation• [R.9.2.d], so this must be parsed as an •identity-declaration• in the strict language. An attempt to parse □proc ¢void¢ p = random□ must begin with the facts that □p□ is a •procedure-void-mode-identifier• and •random• is a •procedure-real-mode-identifier•. Since □random□ is a •base•, we must therefore attempt to find production rules in the hope of showing that a •procedure-real-base• is a production of •strong-procedure-void-base•. The production rule for any given notion can be obtained from only one rule of the Report. If we take that rule [R.8.2.0.1.d] and replace the metanotion •COERCEND• appropriately, we have

•strong procedure void base : procedure void base ;
    strongly ADAPTED to procedure void base.•

Since □random□ is not a •procedure-void-base•, we must now see whether it can be produced from the second alternative. This means replacing •ADAPTED• by each one of its eight terminal productions, i.e., by •dereferenced, deprocedured, procedured, united, widened, rowed, hipped• and •voided•. We look at each of

these in turn. In the rules for dereferencing [R.8.2.1.1.a], we have
•strongly dereferenced to procedure void base :
strongly FITTED to reference to procedure void base• .
Thus the mode enveloped has become longer, i.e., from •procedure-void• to •reference-to-procedure-void•. The same will apply to deproceduring [R.8.2.2.1.a]. Because these two rules feed into each other, we can only lengthen the mode (in the sense used above) by using them. Thus we cannot reach our goal through this route.

The rules for proceduring [R.8.2.3.1.a] yield
•strongly procedured to procedure void base :
void base ;
strongly dereferenced to void base ;
strongly procedured to void base ;
strongly united to void base ;
strongly widened to void base ;
strongly rowed to void base.• .
Each of these must now be examined. In the first place, □random□ is not a •void base•, so we dismiss the first alternative. For the others the words (protonotions) •dereferenced-to-void•, •procedured-to-void•, •united-to-void•, •widened-to-void• and •rowed-to-void• lead us nowhere in the appropriate sections [R.8.2.1.1, 8.2.3.1, 8.2.4.1, 8.2.5.1, 8.2.6.1].

By examining the left hand sides of the rules for widening [R.8.2.5.1], rowing [R.8.2.6.1.] and voiding [R.8.2.8.1], we can see that productions for •strongly ADAPTED to procedure void base• through any of these routes cannot be found. Finally, the rules for hipping [R.8.2.7.1] cannot be used since they apply only to •skips•, •nihils• and •jumps• and □random□ is not one of these. This completes our deduction that □<u>proc</u> ¢<u>void</u>¢ p = random□ is not an •identity-relation•.

Note that for □<u>proc</u> ¢<u>void</u>¢ p = (: random)□, the significant production is
•strongly procedured to procedure void base :
void base.• .
[R.8.2.3.1.a]. Also, for □<u>proc</u> <u>real</u> p = random□ only the empty coercion is required for □random□ is already of a priori mode •procedure-real•.

6.9 The syntactic machine

The coercions are, with the exception of balancing of modes, all contained in the syntactic rules in section 8.2 of the Report. A thorough understanding of coercion therefore requires a knowledge of these rules and a certain dexterity in their use. The reader is encouraged to try some syntactic analysis (parsing) for himself, but to help him on the road we give below a complete analysis, as a •strong-real-unit•, of □i□ in the •cast• □<u>real</u> : i□, where □i□ is in the reach of the •declaration• □<u>int</u> i□. The •identifier• □i□ is thus a •reference-to-integral-mode-identifier• and its a priori mode is •reference-to-integral•. The □<u>real</u>□ in the •cast• indicates that the a posteriori mode is •real•. The references within braces are to the particular rules of the Report which are used.

```
•strong real unit• ......................................... 1
•strong unitary real clause•{6.1.1.e} ...................... 2
•strong real tertiary• {8.1.1.a} ........................... 3
•strong real secondary• {8.1.1.b} .......................... 4
•strong real primary• {8.1.1.c} ............................ 5
•strong real base• {8.1.1.d} ............................... 6
•strongly widened to real base• {8.2.0.d} ****************** 7
•strongly dereferenced to integral base• {8.2.5.1.a} ****** 8
•reference to integral base• {8.2.1.1.a} ................... 9
•reference to integral mode identifier• {8.6.0.1.a} ........10
•letter i• {4.1.1.b} .......................................11
•letter i symbol• {3.0.2.b} ................................12
```

In the above analysis the two coercions occur in lines 7 and 8. In lines 1 to 6, the sort, i.e., •strong•, is carried through the parse until it meets with the •coercend• (in this example a •base•) in line 6. In lines 9 to 12 all the coercions implied by the •strong• in line 1 have been expended. The elaboration naturally follows the parse in the reverse order. At line 10 the •identifier• ¤i¤ is identified with its defining occurrence and the a priori mode, •reference-to-integral•, is established. (This is usually accomplished by an early pass of the compiler.) In line 8 the dereferencing occurs and this is followed by widening in line 7. No further semantics is involved in lines 6 down to 1.

6.10 Balancing

Balancing is the word used to describe the process of finding one mode (the balanced mode) to which each one of a given set of modes may be coerced [1]. The process of finding the balanced mode will be determined by the sort of syntactic position involved. Balancing in a strong position is a simple process (some may even claim that it is not really balancing), whereas the programmer may need to exercise care in the balancing of modes in firm positions, for the final balanced mode may not be immediately clear.

In the reach of the •declaration• ¤<u>bool</u> p, <u>real</u> x, y, <u>ref</u> <u>real</u> xx, [ ]<u>real</u> x1, <u>ref</u>[ ]<u>real</u> xx1¤, an example of soft balancing is

¤(p | xx | x) := 3.14¤ ,

an example of weak balancing is

¤(p | xx1 | x1)[i]¤ ,

an example of firm balancing is

¤2.3 + (p | 3.14 | x)¤

and an example of strong balancing is

¤y := <u>if</u> p <u>then</u> 3.14 <u>else</u> x <u>fi</u>¤ .

----

[1] Strictly speaking, only •coercends• are coerced. We shall find it convenient to speak of coercion of modes, by which is meant the mode enveloped by a •coercend•.

In general, given a set of modes, a balanced mode must be found which is such that each one of the given modes may be coerced to it. In achieving this, at least one of the given modes must be coerceable using the given sort, whereas the others may be strongly coerced, i.e., the limitations of the syntactic position must be accepted by at least one of the given modes, otherwise the balancing is not possible. An example in which a balance is not possible is □2.3 + (p | skip | go_to k)□, which is therefore not a •formula•.

6.11 Soft balancing

A simple example of soft balancing is

□(p | xx | x) := 3.14□ .

Examination of this object suggests an •assignation• in which the mode of the •destination•, □(p | xx | x)□, should be •reference-to-real•. A successful parse is thus assured if the balanced mode of the •conditional-clause• is •reference-to-real•. However, the mode of □xx□ is •reference-to-reference-to-real•, whereas that of □x□ is •reference-to-real•. The mode of □xx□ may be coerced to the balanced mode by dereferencing (once) and that of □x□ by the empty coercion. If we recall that the only coercion which is relevant in soft positions is deproceduring, then it is clear that □xx□ cannot be softly coerced to the balanced mode. One must therefore allow □x□ to be softly coerced and □xx□ may then be strongly coerced (dereferenced). A sketch of the parse of the •destination•

```
                  reference-to-real-destination
                               |
           soft-conditional-reference-to-real-clause
                               |
  ---------------------------------------------------------------
  |         |                      |                            |
if-symbol condition           soft-choice-                  fi-symbol
  |         |           reference-to-real-clause                |
  |         |                      |                            |
  |         |              -----------------                    |
  |         |              |               |                    |
  |         |        strong-then-      soft-else-               |
  |         |        reference-to-    reference-to-             |
  |         |         real-clause      real-clause              |
  |         |              |               |                    |
  |         |         -----------     -----------               |
 □(         p        |      xx       |       x                 )□
```

Fig.6.11

is shown in figure 6.11. The rule which is relevant in this parse is

•FEAT choice CLAUSE : strong then CLAUSE, FEAT else CLAUSE.•

[R.6.4.1.d], in which •FEAT• is replaced by •soft• and •CLAUSE• by •reference-to-real-clause•. This same rule has an alternate production. The complete rule is

•FEAT choice CLAUSE : strong then CLAUSE, FEAT else CLAUSE ;
FEAT then CLAUSE, strong else CLAUSE.• .

The second alternate is clearly necessary for parsing the •assignation•

□(p | x | xx)□ := 3.14□ ,

for in this case □xx□ must be strongly coerced.

Now consider the •assignation•

□(p | x | y) := 3.14□ .

Here either □x□ or □y□ may be chosen to be soft. It follows that □(p | x | y)□ may be parsed as a •reference-to-real-destination• in two distinct ways, i.e., either the □x□ or the □y□ may be chosen as soft with the other strong. This is one of the rare examples of syntactic ambiguity in ALGOL 68. The ambiguity might have been avoided, but at the cost of considerable complexity in the grammar. Since no semantic ambiguity is involved, greater clarity in the grammar is achieved by allowing a harmless syntactic ambiguity.

6.12 Weak balancing

A simple example of weak balancing is

□re of (p | 1 i 2 | 3)□ .

Here the •clause• □(p | 1 i 2 | 3)□ is the •secondary• of a •selection• and is therefore in a weak position [R.8.5.2.1.a]. The mode of □1 i 2□ is •complex•[1], but that of □3□ is •integral•. It is clear that the object □3□ must be widened (twice) to •complex•, but widening cannot occur in a weak position. Thus □1 i 2□ must be weakly coerced (the coercion is empty) and □3□ may then be strongly coerced (widened twice). The balanced mode of □(p | 1 i 2 | 3)□ is therefore •complex•. A sketch of the parse of this •secondary• is shown in figure 6.12.

```
                       weak-complex-secondary
                                 |
                   weak-conditional-complex-clause
                                 |
   ------------------------------------------------------------
   |            |                      |                      |
if-symbol   condition             weak-choice-            fi-symbol
   |            |                 complex-clause              |
   |            |                      |                      |
   |            |            ----------------------           |
   |            |            |                    |           |
   |            |       weak-then-          strong-else-      |
   |            |     complex-clause      complex-clause      |
   |            |            |                    |           |
   |            |       -----------          -----------      |
   □(           p       |    1 i 2           |      3         )□
```

Fig.6.12

The rule used in this parse is the same as that given in paragraph 6.11 above, but this time •FEAT• is replaced by •weak•

----

[1] Here •complex• stands for •structured-with-real-field-letter-r-letter-e-and-real-field-letter-i-letter-m•.

and •CLAUSE• by •complex-clause•.

A weak balance which involves a harmless syntactic ambiguity is

□re of (p | z1 | z2)□

in the reach of the •declaration• □compl z1, z2□. In this case the balanced mode is •reference-to-complex• since weak coercion does not remove the last •reference-to• [R.8.2.1.1.b]. The coercion of both □z1□ and □z2□ is thus empty and either one of them may be chosen as weak.

6.13 Firm balancing

A simple example of firm balancing is

□2.3 + (p | 4.5 | 6)□ .

In this example the •conditional-clause•, □(p | 4.5 | 6)□, is an •operand• of a •formula• and is therefore in a firm position [R.8.4.1.d]. The •operator• □+□ is that declared in the •standard-prelude• [R.10.2.4.i]. It requires a right •operand• of mode •real•. Thus □4.5□ is of the required mode while □6□ must be widened. Since widening may not occur in a firm position, we must choose □4.5□ as firm and then allow □6□ to be strong. A sketch of the parse of this •operand• (•secondary•) is

```
                         firm-real-secondary
                                  |
                    firm-conditional-real-clause
                                  |
 ┌──────────────┬─────────────────┴──────┬──────────────────────┐
 |              |                        |                      |
if-symbol   condition      firm-choice-real-clause         fi-symbol
 |              |                        |                      |
 |              |             ┌──────────┴─────────┐            |
 |              |             |                    |            |
 |              |         firm-then-         strong-else-       |
 |              |         real-clause        real-clause        |
 |              |             |                    |            |
 ┴              ┴         ────┴────          ─────┴────         ┴
□(              p         |     4.5          |       6         )□
```

Fig.6.13

shown in figure 6.13. The relevant rule is again the same as that given in paragraph 6.11 above, but •FEAT• is replaced by •firm• and •CLAUSE• by •real-clause•.

An example of a firm balance in which there is a harmless syntactic ambiguity is

□2.3 + (p | xx | x)□

for dereferencing is permitted in a firm position and both □xx□ and □x□ may be firmly coerced to •real• by dereferencing.

6.14 Strong balancing

A simple example of a strong balance is

□y := (p | x | 1)□ .

Here the •conditional-clause•, ¤(p | x | 1)¤, is a •source• and is therefore in a strong position [R.8.3.1.1.c]. Both ¤x¤ and ¤1¤ must therefore be strongly coerced to the balanced mode which is •real•. This means that ¤x¤ is dereferenced and ¤1¤ is widened.

Observe that strong balancing is a trivial process for one is not faced with the necessity of deciding which of the given modes should retain the sort of the syntactic position. They all retain strong. In the example above, as in most cases of strong balancing, the balanced mode is determined by the context. Balancing in firm, weak and soft positions, however, is different. In these positions the balanced mode is not given by the context but must be decided by examining the given modes alone.

6.15 Positions of balancing

In the example above we have considered balancing only in a •conditional-clause•. This is a typical situation and is sufficient to illustrate the principles involved. However, balancing may occur in other situations and we shall list each of them here.

•choice-clause• in a •conditional-clause• [R.6.4.1.c,d]
e.g., ¤<u>abs</u>(p | 1 | -2.3)¤.
•balance• in a •collateral-clause• [R.6.2.1.e]
e.g., ¤<u>upb</u>(1, 2.3, x)¤.
•suite-of-clause-trains• in a •serial-clause• [R.6.1.1.g]
e.g., ¤((p | 1) ; 3.14 . l : 1)¤.
•identity-relation• [R.8.3.3.1.a]
e.g., ¤xx :=: x¤.

Although these are the only balancing positions in the strict language, the programmer should be aware of their implications in the extended language. For example

¤(p | i |: q | x |: r | 3.14 | 5) + 2.35¤

requires a firmly balanced mode of •real• for the left •operand• of the •operator• ¤+¤. This is achieved by dereferencing and then widening ¤i¤, by dereferencing ¤x¤, by the empty coercion upon ¤3.14¤ and by widening ¤5¤. Since an •operand• must be firm, either ¤x¤ or ¤3.14¤ could be chosen to be firm, and the others could then be strong. Note that since widening cannot be done in a firm position, both ¤i¤ and ¤5¤ must be strong. Another example of firm balancing in the extended language is

¤(i | 1, 3.4, x, random, xx, <u>skip</u> | <u>go to</u> error) + 1¤

in which either ¤3.14¤ or ¤x¤ or ¤random¤ or ¤xx¤ may be firm but the others including the •jump• must be strong.

Notice that a •collateral-clause• may be only firmly or strongly balanced [R.6.2.1.c,d]. Examples, in the reach of ¤[1:3]<u>real</u> x1¤ are

¤<u>upb</u>(x, i, 1)¤

for firm balancing and

¤x1 := (x, i, 1)¤

for strong balancing.

Balancing may occur in a •serial-clause• which contains a •completer•. A trivial example is

□((p | 1) ; 3.14 . 1 : 1) + 2□

Here, if □p□ is •true•, the □1□ is widened to •real• before the addition is performed (despite the fact that the right •operand• is •integral•), for the firmly balanced mode of the left •operand• must be decided without reference to the context.

The balancing of an •identity-relation• is soft. An example is

□xx :=: x□ .

Here the left •tertiary• must be dereferenced once and therefore cannot be soft. The right •tertiary• is therefore chosen to be soft and the coercion upon it is empty. In the •identity-relation•

□x :=: xx□

the choice must be made in the opposite order . The •identity-relation•

□x :=: y□

is syntactically ambiguous since either the left or the right •tertiary• may be soft; however, as in the other case mentioned above, no semantic ambiguity exists. A typical •identity-relation• which might arise in list processing is

□(ref cell : next of cell) :=: nil□

in which the □nil□ can only be strongly coerced. This forces the left •tertiary• to be soft.

6.16 Program example

The following program calculates the greatest common divisor of a set of integers[1]. The original algorithm is in FORTRAN. The ALGOL 68 version given here retains the labels as used in the FORTRAN program (preceded by the letter l) in order to help in the comparison of the two. It is interesting to note that all the jumps of the original naturally disappear except for □go to l10□ in the innermost •conditional-clause•. This could perhaps be eliminated by using a •call• of a recursive •procedure• at the •label• □l10:□.

```
□proc gcdn = (ref [1:] int a ¢the given set of integers¢ ;
              ref[1:upb a] int z ¢the resulting multipliers¢)
¢the gcd result¢ int :
  begin int n = upb a ¢the number of integers¢ ;
  int m := 0, k, sgn ;
  ¢find the first non-zero integer¢
  for i to n while a[i] = 0 do (l1: z[i] := 0, m := i) ;
  ¢the first non-zero integer, if any, is in position m+1¢
    if (m +:= 1) > n ¢now it is in position m¢
    then ¢all are zero, so exit with result¢ 0
    elsf l3: m = n
    then ¢only the last one is non-zero¢ z[m] := 1 ; a[n]
    else l4: ¢check the sign of a[m]¢
```

------

[1] Translated from algorithm 386 by G.H.Bradley, Communications of the Association for Computing Machinery, Vol 13, No 7, 1970.

```
    ref int am = a[m] ;  sgn := sign am ;
    int c1 := am := abs am ; k := m + 1 ;
    15: ¢calculate via n-m iterations of the gcd algorithm¢
    for i from m+1 to n while c1 ≠ 1 do
      begin ref int ai = a[i] ;
      int q, y1 := 1, y2 := 0, c2 := abs ai ; k := i ;
        17: if ai = 0
        then ai := 1 ; z[i] := 0
        else 110:
          if q := c2 ÷ c1 ; (c2 ÷::= c1) ≠ 0
          thef y2 -:= q * y1 ; q := c1 ÷ c2 ; (c1 ÷::= c2) ≠ 0
          then y1 -:= q * y2 ; go to 110 ¢eliminate the jump?¢
          else 115: (c1 := c2, y1 := y2)
          fi ;
        120: z[i] := (c1 - y1 * am) ÷ ai ;
        ai := y1 ; am := c1 fi ;
      130: skip end ;
    ¢ if k=n, then the following iteration is empty¢
    125: 160: for j from k+1 to n do (165: z[j] := 0) ;
    140: for j from k-m by -1 to 2 do
      (z[j] *:= a[j+1] ; 150: a[j] *:= a[j+1]) ;
    z[m] := a[m+1] * sgn ;
    1100: am
    fi
  end□
```

Review questions

6.1 Fundamentals

a) What three things determine the particular coercions?
b) What are the four sorts of syntactic position?
c) Is □real : int□ a •cast•?
d) Is □real : bool□ a •cast•?
e) What coercion occurs in □[ ]bool : 101□?

6.2 Classification of coercions

a) How many different coercions are there?
b) What coercions occur in □real : int□?
c) What coercions are classified as fitting?
d) What coercion occurs in □[ ]real : 3.14□?
e) What coercion occurs in □int : go to k□?

6.3 Fitting

a) What coercions occur in □real : ref ref ref real□?
b) In the reach of □ref ref real xxx□, what coercions occur in □ref real : xxx□?
c) In the reach of □ref proc int rpi□, what coercions occur in □int : rpi□?

d) In the reach of ¤proc ref bool prb, what coercions occur in ¤bool : prb¤?
e) What rules are used in the parse of ¤real : random¤ as a •real-cast•?

6.4 Adjusting

a) What coercions occur in ¤union(real, bool) : random¤?
b) Is uniting a fitting coercion?
c) What kind of value results from a proceduring?
d) Is ¤proc ¢void¢ : sin¤ a •cast•?
e) Is ¤proc ¢void¢ : random¤ a •cast•?

6.5 Adapting

a) Is hipping an adjusting coercion?
b) What coercion occurs in ¤bool : go to k¤?
c) What coercions occur in ¤x := (1 > 2 | 3.4 | 5)¤?
d) What coercions occur in ¤[ ]real : random¤?
e) What coercions occur in ¤union([ ]real, bool) : random¤?

6.6 Syntactic position

a) What coercions may occur in weak positions?
b) Of what sort is ¤i¤ in ¤x1[i+1]¤?
c) Of what sort is ¤n1¤ in ¤x1[n1[i]]¤?
d) In the range of ¤ref ref [ ]real rr1x¤, what coercions occur in ¤rr1x[2] := 2.3¤?
e) Of what sort is ¤x¤ in ¤x := y¤?

6.7 Coercends

a) What are the four kinds of •coercend•?
b) List all the •coercends• in ¤if a of b then x := 2 else x := y + 3 fi¤.
c) Is ¤x := nil¤ an •assignation•?
d) Is ¤xx := nil¤ an •assignation•?
e) Is ¤nil := 1¤ an •assignation•?

6.9 The syntactic machine

a) What rules are used in parsing ¤compl : i¤?
b) Is ¤compl : union(int, bool)¤ a •cast•?
c) What rules are used in the parse of ¤proc ¢void¢ p = (: x := 1)¤?
d) What rules are used in the parse of ¤random¤ as a •strong-void-unit•?
e) Is ¤x + nil¤ a •formula•?

6.10 Balancing

a) Can the modes •real•, •integral• and •format• be strongly balanced to real?
b) Can the modes •real• and •integral• be strongly balanced?
c) What is the softly balanced mode from the two modes •reference-to-real• and •procedure-real•?

d) What is a firmly balanced mode from the set of modes •real•, •integral•, •procedure-integral• and •reference-to-integral•?
e) Can the modes •real• and •boolean• be balanced?

6.11 Soft balancing

a) Is the parsing of □(p | xx | y) := 3.14□ ambiguous?
b) In the reach of □proc ref real px□, how is □(p | px | xx) := 3.14□ balanced?
c) In the reach of □proc ref real px□, how is □(p | px | go to k) := 2□ balanced?
d) Can the pair of modes •procedure-row-of-real• and •reference-to-real• be softly balanced?
e) Can the modes •reference-to-procedure-reference-to-boolean• and •reference-to-reference-to-boolean• be softly balanced?

6.12 Weak balancing

a) In the reach of □[ ]real x1□, how is □(p | x1 | 2)[i]□ balanced?
b) Can the modes •reference-to-real• and •union-of-real-and-integral-mode• be weakly balanced?
c) Is □1 + re of (p | 1.2 | 3.4 i 5)□ a •formula•?
d) Is □re of (p | 1 i 2 | 3 i 4)□ syntactically ambiguous?
e) How is □im of (p | random | 0 i 2)□ balanced?

6.13 Firm balancing

a) Is □skip / skip□ a •formula•?
b) Can •union-of-reference-to-real-and-reference-to-integral-mode• and •real• be firmly balanced?
c) Can •procedure-real• and •reference-to-real• be firmly balanced to •procedure-real•?
d) Is □2 + (p | x | 3.14)□ syntactically ambiguous?
e) Is □abs (p | true | "a")□ a •formula•?

6.15 Positions of balancing

a) Can the set of modes •reference-to-reference-to-procedure-reference-to-real•, •reference-to-procedure-reference-to-real•, •reference-to-reference-to-real• and •reference-to-real• be weakly balanced?
b) Is □(i | xx, nil, skip | go to error) :=: x□ an •identity-relation•?
c) Is □((p | l1) ; true . l1 : (i > 0 | l2) ; false . l2 : 1)□ a •closed-clause•?
d) How is □upb (1, 2.3, 4 i 5.6, x, xx, i)□ balanced?
e) Is □(p | nil | skip) := 3.14□ an •assignation•?

6.16 Program example

a) Describe the coercions involved in the elaboration of □(m +:= 1) > n□.
b) Describe the elaboration of □int c1 := am := abs am□.
c) What is the purpose of the •declaration• □ref int ai = a[i]□?

d) Why does a •skip• occur on line ¤110: skip end¤?
e) Can you eliminate the ¤go to 130¤ by using a recursive procedure at the position ¤110:¤?

7 United modes

7.1 United declarers

Although internal objects are always of one non-united mode, external objects such as •expressions• [R.6.0.1,a,b] may be of united mode, indicating that the mode of the value possessed is not known until elaboration (run time). To allow for this, it is necessary for the language to provide •declarers• which specify united modes. Examples of such •declarers• are □union(int, bool), union([ ]real, [ ]char), union(ref[ ]int, ref[ ]real), union(a,union(b, c), d)□.

The syntax of •united declarers• is not trivial but we may simplify it to the following:

```
  united declarer : union of symbol,
    open symbol, declarer list proper, close symbol.
  declarer list proper : declarer, comma symbol, declarer ;
    declarer list proper, comma symbol, declarer.
```

The syntax of the Report [R.7.1.1.cc,...,jj], however, is an intricate exercise in the use of metanotions. Its effect is to allow, syntactically, that unions may be both commutative and associative, and that the modes of the union may be treated in the sense of mathematical set theory. This means that the same united mode is specified by the •declarers• □union(a, b, c), union(a, c, b), union(a, union(b, c))□ and □union(union(c, a), union(c, b))□.

7.2 Assignations with united destination

Because •declarers• specifying united modes exist, the declaration of •variables• using such •declarers• is possible. Such a •declaration• might be □union(int, bool) ib□, whereupon the mode of □ib□ is •reference to union of integral and boolean mode•. An assignment may be made to such a •variable•,

```
               reference-to-union-of-integral-and-boolean-
                           mode-assignation
                                  |
           ┌----------------------┴-┬--------------------┐
           |                        |                    |
 reference-to-union-of-   becomes-  strong-union-of-
 integral-and-boolean-     symbol   integral-and-boolean-
     destination              |     :        source
           |                  |     :          | (1)
           |                  |     :        boolean-
           |                  |     :         base
          ─┴─                ─┘     :         ──┴─
          □ib                :=               true□
           :       (4)              : (2)      : (2)
          o(3)     ┌┌-------┐       :......┌---┐
        o o-->──┤|        |=======<========|   |
          o        └└-------┘              └---┘
```

Fig 7.2

but the •assignation• □ib := true□ is syntactically possible only because of the uniting coercion to which the •base•, □true□, resulting from its strong position as a •source•, is subjected (see figure 7.2 at 1). The •assignation• □ib := 1□ is also valid. In both these assignments the internal object assigned does not change under coercion, and the object □true□ possesses the same value whether it is considered, a priori, as a •base•, or, a posteriori, as a •source• (see the figure at 2). Note that □ib□ possesses a name (see figure at 3), whose mode is •reference to union of integral and boolean mode•, but that this name may refer to a value which is either of mode •integral• or of mode •boolean•, since values are not of united mode (i.e., a mode which begins with •union of•). Also, the mode of the value referred to by such a •variable• as □ib□, can be determined, in general, only at the time of elaboration of the •program• (not at "compile time"). These considerations lead one to suspect that the use of united modes implies storage allocation or run time organization methods which must be more elaborate than those required when such modes are not used (see the figure at 4). A certain price must therefore be paid for the use of united modes, but in some situations they are essential (see[R.11.11]); moreover, ALGOL 68 is designed to minimize those places in a •program• where a run time check of the mode of a value is necessary. Such a check is unnecessary for the •assignations• □ib := true□ and □ib := 1□. These checks are known as •conformity-relations•. Before passing to these we examine two further •assignations•.

In the range of the •declaration• □int n, bool p□ one might be tempted to consider the objects □n := ib□ and □p := ib□ in the hope that the assignment would take place, if possible. However neither of these two is an •assignation•, for in both cases, though the mode of the destination begins with •reference-to•, it is not followed by the mode of the •source•. In particular, there is no deuniting coercion. Thus we must rule them out as not belonging to ALGOL 68.

7.3 Conformity relations

•Conformity-relations•, like •assignations•, •identity-relations• and •casts•, are •confrontations•. Examples of •conformity-relations• are: □i ::= ir, real :: x of q□ and □a and b ::= i + 2 * x□. The syntax of •conformity-relations• might be written

conformity relation : tertiary, conformity relator, tertiary.
conformity relator :
conforms to and becomes symbol ; conforms to symbol.

This syntax makes the •conformity-relation• appear to be symmetrical, but this is not the case as an examination of the strict syntax of the Report [R.8.3.2.1] will reveal. There one may see that the •tertiary• on the left is soft, whilst that on the right is not of any sort and therefore cannot be coerced. Moreover, the mode of the left •tertiary• must begin with •reference-to•. We may recall that the •destination• of an •assignation•, i.e., the □x□ in □x := 3.14□, is soft, so that there is some similarity between •assignations• and •conformity-relations•. This is intentional, for the elaboration of a •conformity-relation• often results in an assignment. The right •unit• of an •assignation•, e.g., □3.14□ in □x := 3.14□, however, is strong. Thus the right •unit• of an •assignation• is strongly coerced but the right •tertiary• of a •conformity-relation• is not coerced.

We may now ask what the difference is between □x := 3.14□ and □x ::= 3.14□. In the case of □x := 3.14□, an assignment is made. In the case of □x ::= 3.14□, an assignment is also made but not before checking that such an assignment is possible. Another difference is that the value of □x := 3.14□, after its elaboration, is the name possessed by □x□, but the value of □x ::= 3.14□ is a truth value, viz., ▪true▪.

Now consider □x := 1□ and □x ::= 1□. In the case of □x := 1□ an assignment of the real value, •1.0•, is made to □x□ after the widening of □1□ to a value of mode •real•, but □x ::= 1□ delivers the value ▪false▪ and no assignment takes place. Note that the □1□ in □x ::= 1□ is not coerced and in particular cannot be widened to •real•. The reader may now protest that any simple minded compiler could determine, at compile time, that the value of □x ::= 3.14□ is ▪true▪ and that the value of □x ::= 1□ is ▪false▪, thus the information yielded is trivial. We agree. However, the possibility of using united modes makes the •conformity-relation• an essential tool, as we shall soon discover.

We have mentioned that the right •tertiary•, e.g., the □1□ in □x ::= 1□ is not coerced. Therefore we may ask what will happen with □x ::= y□ and □x ::= i□. The semantics of the •conformity-relation• [R.8.3.2.2] now comes to the rescue. It tells us that, instead of returning the value ▪false▪ immediately, the right •tertiary•, e.g., the □y□ in □x ::= y□ is dereferenced as often as is necessary or possible. Thus □x ::= y□ will deliver ▪true▪ and □x ::= i□ will deliver ▪false▪ and in arriving at this, both the □y□ and the □i□ are dereferenced once.

```
                    boolean-conformity-relation.............
                                  |                        :
         ┌------------------------+----------------------┐  ▪true▪
         |                        |                      |   (2)
  soft-reference-to-         conformity-               real
    real-tertiary              relator               tertiary
         | (1)                    |                      |
    reference-to-                 |                    real-
      real-base                   |                 denotation
         ┴                       ─┴─                    ─┴──
         □x                      ::=                   3.14□
         :                                               :
         o                 ┌-------┐                 ┌---┴---┐
         o o--->---┤       |==<==(assignment)==<==|       |
         o                 └-------┘                 └-------┘
```

Fig.7.3

The only difference between the •conformity-relations• □x ::= 3.14□ and □x :: 3.14□ is that no assignment occurs in □x :: 3.14□ despite the fact that the value yielded by □x :: 3.14□ is ▪true▪. A skeletal parse of the •conformity-relation• □x ::= 3.14□ is shown in figure 7.3, where the only coercion involved (it does nothing) is shown at 1 and the value possessed by the •conformity-relation• at 2.

We see therefore that the •conformity-relation• is a way of finding out whether an assignment is or is not possible. Without united modes, this would be of no value, since this information is known at compile time. It is only when united modes are used that the •conformity-relation• is useful. Thus the examples given above are merely for the purpose of illustrating the fundamentals of the •conformity-relation• and have no value in practical programming.

7.4 Conformity and unions

Suppose now that we are in the reach of the •declaration• □union(int, char) ic□. Then the value of the •clause• □(int i; ic := "a" ; i :: ic)□ is ▪false▪ and the value of the •clause• □(int i ; ic := 1 ; i :: ic)□ is ▪true▪. Note that, without following the logic of the •program•, these values cannot be determined at compile time. How can one use these things? The reader who is irked by trivialities is advised to turn to the Report [R.11.1, 10.5.2.1.b, 10.5.2.2.a, 10.5.3.1.b, 10.5.3.2.b, 10.5.4.2.b] where there are many examples of •conformity-relations• in action. For those not so brave, consider the following problem.

We wish to write a •procedure•, say □translate□, which will accept either an integer or a character as its only parameter and will deliver either a character or an integer which is the environmental equivalent [R.10.1.j,k]. Thus suppose that in a given environment the integral equivalent of ▪a▪ is ▪193▪, the •call• □translate("a")□ should then possess an integral value ▪193▪ and the •call• □translate(193)□ should possess the character value ▪a▪. Its declaration then might be

```
□proc translate = (union(int, char) a) union(int, char) :
      begin int i, char c ;
      if i ::= a then repr i # R.10.1.k #
      else c ::= a ; abs c # R.10.1.j # fi end□          .
```

In the body of this procedure the •condition•, □i ::= a□, determines whether the value delivered is □repr i□ or □abs c□. The value of the •conformity-relation• □c ::= a□ is voided, since one knows that, if control reaches it, the value will be ▪true▪; however, its presence is essential because the •operator• □abs□ is not defined for operands of united mode.

7.5 Conformity extensions

•Conformity-relations• occur in certain extensions, both for the convenience of the programmer and for the purpose of allowing more efficient implementation of certain constructions. Examples of these extensions occur in the Report [R.11.11.q,ah].

We begin by explaining them in a simple way.

The •conditional-clause•
□(a ::= u | 1 |: b ::= u | 2 |: c ::= u |: 3 | 0)□
can be written
□[* a, b, c ::= u *]□ .
Its effect then is to test several conformities in succession, delivering as an integral value the index of the one which succeeds. If all of them fail then the result ▪0▪ is delivered. This, in itself, is useful, but its main purpose is for use as the •unitary-clause• which follows the □case□ in a case clause [R.9.4.b,c]. In this particular situation the two enclosing symbols □[*□ and □*]□ may be omitted. A case clause might therefore be
□case a, b, c ::= u in f(a), g(b), h(c) out error exit esac□
and its interpretation is the following: if □a□ conforms to and becomes □u□, then the value is □f(a)□; otherwise, if □b□ conforms to and becomes □u□, then the value is □g(b)□; otherwise, if □c□ conforms to and becomes □u□, then the value is □h(c)□; otherwise the value is that of □error exit□. Note that if both □a ::= u□ and □b ::= u□ possess the value ▪true▪, then it is undefined whether the value is □f(a)□ or □g(b)□. Examples of the use of this extension are in the Report [R.11.11.q,ah]. We could perhaps write the procedure of section 7.4 as follows:

```
□proc translate = (union(int, char)a)union(int, char) :
     begin int i, char c ;
     case i, c ::= a in repr i, abs c esac
     end□
```

though little would be gained in this simple example.

The description of the extensions [R.9.4.e,f], however, is forbidding and it is perhaps worth while taking a little time to discover why it must appear in this way. Suppose we have the conformity case clause □(x, x ::= u | 9, 8 | error)□. It is clear that if it is interpreted as the equivalent of □(x ::= u | 9 |: x ::= u | 8 | error)□, then the value ▪8▪ can never be delivered. This is unfortunate, for the implementer of the language may find it convenient and more efficient to make the conformity test in an order different from that given. It therefore should be made impossible for the programmer to determine from the Report the order in which the conformity tests are made. This can be done by describing the extension by means of parallel processing. It is worth our while to examine this more closely.

According to the Report [R.9.4.e], the •clause• □[* x, x ::= u *]□, in the reach of □real x, union(int, real) u□, is equivalent to the following

```
□(int i, sema s = /1 ; union(int, real) k = u ;
     par(( x ::= k | down s ; i := 1 ; m ),
  ( x ::= k | down s ; i := 2 ; m)) ; 0 . m : i )□   .
```

The •declaration• □union(int, real) k = u□ ensures that the elaboration of □u□ occurs once only; its value is then held in □k□. The •declaration• □sema s = /1□, declares a semaphore □s□ [R.10.4] which will be used to control the elaboration of the two •clauses• in parallel. The semaphore is initialized to the value ▪1▪. The two clauses beginning with □x ::= k□, are, if this conformity is successful, followed by the •formula• □down s□ which drops the value of the semaphore to ▪0▪ and thus forms a barrier in the elaboration of whichever •clause• did not reach this action first. From this it is therefore not possible to predict whether the value ▪1▪ or ▪2▪ will be delivered. To the programmer, this is an unimportant matter, but the meticulous implementer will be pleased that there is no way in which he can be caught if he decides on one method of implementation rather than another.

The reader should now examine the description of the extensions in the Report [R.9.4.e,f,g] where he will see that it is necessary in this description to have □(S / 1)□ rather than □/1□ because the •operator• □/□ as a •monadic-operator• with an integral right •operand• could be redefined by the programmer. The letter □S□ stands for the •standard-prelude• and therefore returns to the original meaning of □/□ as a •monadic-operator• which accepts an integer as right •operand• and delivers an equivalent semaphore.

Review questions

7.1 United declarers

a) Is □union(int, bool) :=: union(bool, int)□ an •identity-relation•?
b) Is □union(int, bool) := bool□ an •assignation•?
c) What is the value of □union(int, union(bool, char)) :: union(bool, char, int)□?
d) Is □[1:n]union(char, int)□ a •declarer•?
e) Is □union(int, struct(int a))□ a •declarer•?

7.2 Assignations with united declarers

a) In the reach of □union(char, bool) cb□, is □cb := 1□ an •assignation•?
b) In the reach of □union(real, bool) rb□, is □rb := 1□ an •assignation•?
c) In the reach of □union(real, bool) rb□, what is the mode of the value referred to by the name possessed by □rb□?
d) Is □union(bits, bytes) :=: nil□ an •identity-relation•?
e) In the reach of □union(int, char) ic□, is □ic := ic + 1□ an •assignation•?

7.3 Conformity relations

a) In the reach of □union(real, char) rc□, what is the value of □rc :: rc□?
b) What is the value of □x ::= true□?
c) In the reach of □mode br = union(bool, real); union(int, br) ibr, br br□, what is the value of □ibr ::= br□?
d) In the reach of □union(bool, int) bi□, is □bi := i ::= 1□ an •assignation•?

e) Is □x ::= x ::= x□ a •conformity-relation•?

7.4 Conformity and unions

a) In the reach of □union(char, bool) cb□, is □x ::= cb□ a •conformity-relation•?
b) In the reach of □union([ ]real, real) r1r□, is □r1r ::= 3.14□ a •conformity-relation•?
c) Can □union([ ]int, [ ]ref int)□ be contained in a proper •program•?
d) In the reach of □union(int, real) ir□, can □ir := 1□ possess a name referring to a real value?
e) Declare a •procedure• which will accept an integer and deliver its square root, as an integer if it is integral and, otherwise, as a real value.

7.5 Conformity extensions

a) What is the value of □(x, i, b ::= 1 | 3, 4, 5, | 6)□?
b) What is the value of □(real, real, real :: 3.14 | 7, 8, 9 | 10)□?
c) Is □sema p = 1□ a •declaration•?
d) Is □case x, i, b :: u in f(x), g(i) out h esac□ a valid ALGOL 68 object?
e) In the reach of □union(char, int, bool) cib□ is □cib ::= skip□ a •conformity-relation•?
f) Is □x ::= go to k□ a •conformity-relation•?

8 Formulas and operators

8.1 Formulas

In section 3.11 •formulas• were discussed and the following simplified syntax was presented:

```
  formula : operand, dyadic operator, operand ;
    monadic operator, operand.
```

This is good enough as a first approximation but it does not help to explain that a •formula• such as

□x + y * z□

is elaborated in the order suggested by □x + (y * z)□. The question then is how the priority of the •operators• may be used to determine the order of elaboration. A closer approximation to the syntax of •formula• (still ignoring modes and coercion) is

```
  PRIORITY formula : PRIORITY operand,
    PRIORITY operator, PRIORITY plus one operand.
  PRIORITY operand :
    PRIORITY formula ; PRIORITY plus one operand.
  priority NINE plus one operand : monadic operand.
  monadic operand : monadic formula ; secondary.
  monadic formula : monadic operator, monadic operand.
```

[simplified from R.8.4.1.b,d,e,f,g]. Here the terminal productions of •PRIORITY• are [R.1.2.4.a,...,n] •priority-one•, •priority-one-plus-one•, •priority-one-plus-one-plus-one•, etc. Thus, •priority-NINE• has the meaning that one might expect. It is evident that the metanotion, •PRIORITY•, is being used here as a counter to ensure that the left •operand• must have priority not less than that of its associated •dyadic-operator• and the right •operand• must have priority greater than that of its associated •dyadic-operator•. We shall find it convenient to shorten the terminal productions of •PRIORITY•, in an obvious

```
                            p6-formula
                                 |
     ------------------------------------------------
     |                 |                            |
 p6-operand       p6-operator                  p7-operand
     |                 |                            |
     |                 |              ------------------------------
     |                 |              |             |              |
     |                 |         p7-operand    p7-operator    p8-operand
     |                 |              |             |              |
 secondary             |          secondary         |          secondary
     |                 |              |             |              |
    □x                 +              y             *             z□
```

Fig.8.1.a

way, to •p1, p2, p3, ... •. Using this shorthand notation, we obtain, from the first three rules above, the following nineteen rules:

```
  p1 formula : p1 operand, p1 operator, p2 operand.
  p1 operand : p1 formula ; p2 operand.
  p2 formula : p2 operand, p2 operator, p3 operand.
  p2 operand : p2 formula ; p3 operand.
     ...             ...             ...
  p9 formula : p9 operand, p9 operator, p10 operand.
```

p9 operand : p9 formula ; p10 operand.
p10 operand : monadic operand.
We may now present, in figure 8.1.a, a simplified parse of the •formula• □x + y * z□, remembering that □+□ is a •p6-operator• and □*□ is a •p7-operator•.

Because a •dyadic-operator• requires that its left •operand• be of the same priority (or higher) and that its right •operand• should be of higher priority, the •formula•

□x + y + z□

is elaborated as if it were □(x + y) + z□, for the only possible parse is that sketched in figure 8.1.b.

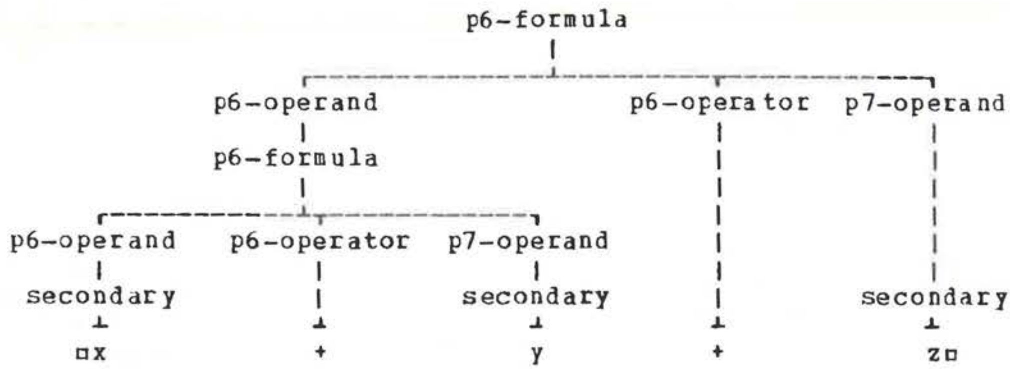


Fig.8.1.b

It is important to observe that, in a •formula• containing several •operators•, the •operands• of each •operator• are determined solely by the priorities of the •operators• and do not depend in any way upon the modes of the •operands•. Thus, assuming that the •operator• □d1□ has priority ▪1▪, □d2□ has priority ▪2▪ and so on, we know that the •formula•

□h d3 i d2 j d5 k d4 l d7 m d9 n□

must be elaborated in the order suggested by

□(h d3 i) d2 ((j d5 k) d4 (l d7 (m d9 n)))□ ,

without any knowledge of the modes of □h, i, j, k, l, m□ and □n□. The compiler writer appreciates the necessity for this mode independence and the programmer gains because of the resulting clarity in the meaning of •formulas•.

8.2 Priority declarations

•Priority-declarations• were mentioned, in passing, in section 3.11. An example of a •priority-declaration• is

□priority + = 6□ ,

which is indeed one of the •declarations• in the •standard-prelude• [R.10.2.0.a]. A parse of this particular •declaration• is shown in figure 8.2, where •6-token• is used here as shorthand for •one-plus-one-plus-one-plus-one-plus-one-plus-one-token•.

The syntax of •priority-declaration• is
•priority-declaration : priority symbol,

priority NUMBER indication, equals symbol, NUMBER token.• , [R.7.3.1.a], where we may observe that the metanotion •NUMBER• [R.1.2.4.f] is used as a counter to ensure that the value of the

```
                              priority-declaration
                                       |
          ---------------------------------------------------------------
          |                 |                    |                      |
  priority-symbol     p6-indication        equals-symbol             6-token
  ---------------           |                    |                      |
     □priority              +                    =                      6□
```

Fig.8.2

•token• on the right is the priority of the •dyadic-indication• on the left.

The first two •dyadic-indications• [R.4.2.1.d] used in section 8.1 above might have been declared in

□priority d1 = 1, priority d2 = 2□

but all of them might be declared more compactly by using an extension [R.9.2.c] which allows elision of □priority□s, as in

□priority d1 = 1, d2 = 2, d3 = 3, d4 = 4,
d5 = 5, d6 = 6, d7 = 7, d8 = 8, d9 = 9□ .

Observe that the programmer may choose his own •dyadic-indications•, like □d1□ and □d2□ and is not constrained to use only those which appear in the Report. The particular representations permitted will be determined by the implementation, but it is expected that most implementations will permit representations like □d1□ and □d2□ together with such characters as □?□ and □!□, if available, and which are not already used as representations of some symbols [R.1.1.5.b].

8.3 Operation declarations

Among the well known programming languages •priority-declarations• may be unique to ALGOL 68. Certainly •operation-declarations• are rare. The latter exist, perhaps in a more primitive form, in APL where all priorities are the same.

A simplified syntax of •operation-declaration• is

operation declaration :
  caption, equals symbol, actual parameter.
caption : operation symbol, virtual plan, operator.

[R.7.5.1.a,b], but the strict syntax uses the metanotion •PRAM• to convey information about the number of and the modes of the •parameters• and the metanotion •ADIC• to convey information about the priority of the •operator• and whether it is monadic or dyadic.

An example of an •operation-declaration• (in the strict language) is

□op (real, real) real max =
((real a, real b) real : (a > b | a | b))□

and a simple parse is shown in figure 8.3. In the extended language it may be written

□op max = (real a, b) real : (a > b | a | b)□ ,
for if the •actual-parameter• is a •routine-dentation•, then the •plan• may be elided and the •routine-denctation• may be

```
                          operation-declaration
                                    |
               r--------------------+-------------------,
            caption           equals-               actual-
               |              symbol                parameter
  r------------+---------,       |                      |
operation-  virtual-  oper-|                            |
  symbol       plan    ator |                            |
    |            |       |  |                            |
   _|_  ---------|----- -|- |  -------------------------|----------------
  □op  (real, real)real max = (real a,b)real : ( a > b | a | b)□
```

Fig.8.3

unpacked [R.9.2.e,d]. Before going further we should remember that this •declaration• can only occur in the reach of a •priority-declaration• like □priority max = 7□.

In the reach of the •declarations• given above, we may have a •formula• like □x max y + 3.14□. Since the priority of the standard •operator• □+□ is six, we should expect this •formula• to be elaborated in the order suggested by □(x max y) + 3.14□. If the •priority-declaration• had been □priority max = 5□ instead, then the •formula• would be elaborated as if it were □x max (y + 3.14)□.

The •actual-parameter• need not necessarily be a •routine-denotation•. For example,

□op (string, int) int si = string int□

is an •operation-declaration• in which the •actual-parameter• is an •identifier•. The •operator• □si□ is then made to possess the same routine as that possessed by □string int□ [R.10.5.2.2.c]. In the reach of this •declaration• the •formula• □"+123" si 10□ will possess the same value as that possessed by the •call• □string int("+123", 10)□. Observe that

□op si = string int□

is not an •operation-declaration• because □string int□ is not a •routine-denotation• so the •plan• □(string, int) int□ cannot be elided.

It is not necessary that an •operation• should deliver a value, but if it does not, then a •formula• containing such an •operator• cannot be used as an •operand•. Thus one loses some of the advantages of •operators•, except perhaps for the benefit of compactness of expression.

An example is

□op interchange = (ref real a, b) :
(a :≠: b | real t = a ; a := b ; b := t)□ ,

whose •operator•, □interchange□, could be used in the •formula• □x interchange y□. The same effect would be obtained by means of the •identity-declaration•

```
            □proc interchange = (ref real a, b) :
        ( a :≠: b | real t = a ; a := b : b := t)□                ,
```

whose •identifier• could then be used in the •call• □interchange(x, y)□. One might observe that the •actual-parameter• is the same •routine-denotation• in both •declarations• above.

•Operation-declarations• may therefore allow a compactness of algorithms since •formulas• using •operators• of several priorities may be built to do any job we may require. A •formula• like

□x max y max 0.1□

is sometimes a more pleasing expression of thought than a nesting of •calls• like

□max(max(x, y), 0.1)□ ,

although LISP lovers may not agree.

8.4 Elaboration of operation declarations

An •operation-declaration• causes its •operator• to possess that routine which is possessed by its •actual-parameter• [R.7.5.2]. In the elaboration of

□op max = (real a, b) real : (a > b | a | b)□ ,

the •operator• □max□ is made to possess the routine

•(real a = skip, real b = skip ; real : (a > b | a | b))• .

This is, of course, already the value possessed by the •routine-denotation• which is the •actual-parameter• on the right. The elaboration of an •operation-declaration• is thus similar to that of the •identity-declaration•, particularly that in which the •actual-parameter• possesses a routine with one or two •parameters•.

8.5 Dyadic indications and operators

Although the same occurrence of an external object may be a representation of both a •dyadic-indication• and an •operator•, the identification of the object, as it plays each role, is a distinct process. An example may help to illustrate this. In the •closed-clause•

```
□( priority max = 7 ;
            ¢1¢
    op max = (real a, b) real : ( a > b | a | b ) ;
       ¢2¢
    x := x max y + 3.14 )
           ¢3¢□
```

there are three occurrences of the object □max□. The first occurrence is the defining occurrence of a •dyadic-indication• [R.4.2.1.e, 4.2.2.a]; the second occurrence is an applied occurrence of □max□ as a •dyadic-indication• and its defining occurrence as an •operator• [R.4.3.1.b, 4.3.2.a]; the third occurrence of □max□ is an applied occurrence of a •dyadic-indication• and an applied occurrence of an •operator•. Thus, in each of the last two occurrences, the object □max□ represents two notions, both of which are involved in the identification process. Since an applied occurence must always identify a defining occurrence [R.4.4.1.b], the last occurrence of □max□

identifies two defining occurences, i.e., the first as a •dyadic-indication• and the second as an •operator•. In figure 8.5 we sketch the parse of each of the three occurrences of □max□ and indicate by "<===" how the identification occurs.

```
                                 operation-
                                 declaration          formula
         priority-                    |                  |
         declaration            r-----+--T--,       r----+-----,
              |                 |     |             |    |
      r-------+------T-,        |  operator  <====  operator
      |       |                 |     |                  |
      |     dyadic              |   dyadic             dyadic
      |   indication   <====    | indication  <====  indication
      |       |                 |     |                  |
    --+------ -+-               +-  --+                 -+-
  □priority max□               □op  max□           □x max y□
```

Fig.8.5

It is thus helpful to remember that an object like □max□, except in a •priority-declaration•, must be considered first as a •dyadic-indication• (carrying the information about priority) and second as an •operator• (possessing an operation - a routine). As a •dyadic-indication• it may identify only one defining occurrence [R.4.2.2, 4.4.2.b], but as an •operator• it may, at different applied occurrences, identify more than one defining occurrence [R.4.3.2]. One need only consider the •formulas• □3.14 + 4.25□ and □123 + 456□ to realise that the standard •operator• □+□, in the first •formula•, must be that which adds two real values [R.10.2.3.i] and in the second it is that which adds two integral values [R.10.2.4.i]. This "overloading" of •operators• (i.e., allowing them to have more than one meaning) has been traditional both in mathematics and in programming languages, so that it should not be difficult for us to remember that in ALGOL 68 any •operator• may have a meaning which depends upon the modes of its •operands•. Moreover, the programmer now has the power to overload operators at will.

8.6 Identification of dyadic indications

The identification of •dyadic-indications•, like that of •identifiers•, is a simple process. For each applied occurrence one must search in the current •range• for a defining occurrence. If it is not found, then one searches in the next outer •range• [R.4.2.2.b]. The process is then repeated. If a •particular-program• contains no •priority-declarations•, then the defining occurrence of any •dyadic-indications• will be found in the •standard-prelude• (or perhaps a •library-prelude•). Since •dyadic-indications•, again like •identifiers•, are subject to protection [R.6.0.2.d, 6.1.2.a], i.e., to systematic replacement in a •closed-clause• in order to avoid confusion with the same object used elsewhere, it follows that the occurrence of, say

□priority + = 1□

in some •range• will mean that all operations possessed by the •operator• □+□, in the next outer •range•, will become inaccessible. A small example may help to make this point clear. In the object

```
          □( priority max = 7 ;
                          ¢1¢
             op max = (real a, b) real : ( a > b | a | b ) ;
                ¢2¢
             x := 1.23 max y ;
                       ¢3¢
             ( priority max = 5 ;
                          ¢4¢
               x := 2.34 max y )
                         ¢5¢
          )□
```

the fifth occurrence of □max□ identifies the fourth occurrence. Moreover, due to protection of the inner •closed-clause•, both of these occurrences are systematically changed into some other •indicant• which is not used elsewhere. Consequently, the last occurrence of □max□ is that of an •operator• with no defining occurrence. Because of a context condition [R.4.4.1.b], this could not be contained in a proper •program•. This means that the changing of priorities of the standard •operators• cannot be undertaken lightly. Perhaps it is just as well.

8.7 Identification of operators

The identification of •operators• is not as simple. It is not sufficient for the •symbol• to match that which occurs in an •operation-declaration• since, as we have said before, one same •dyadic-indication•, when considered as an •operator• may, at different occurrences, identify more than one defining occurrence. The additional requirements to be satisfied are as follows. The mode of the left •operand• must be firmly coerceable to the mode of the first •formal-parameter• in the •operation-declaration• and the mode of the right •operand• must be firmly coerceable to the mode of the second •formal-parameter•; otherwise, the search for a defining occurrence proceeds to the other •operation-declarations• in the same •range•, or, as before, in successive outer •ranges•. We shall illustrate this with a simple example.

```
     □¢1¢     ( priority o = 8 ;
      ¢2¢       op o = (real a, b) real : 3.14 ;
      ¢3¢       ( op o = (real a, int b) real : 3.15 ;
      ¢4¢         ( op o = (bool a, b) real : 3.16 ;
      ¢5¢           2.3 o x)))□
```

The question to be answered here is, which defining occurrence is identified by the •operator• □o□ in the •formula• □2.3 o x□ in line 5. One first searches the •range• in which that •formula• occurs. There is an •operation-declaration•, on line 4 in this •range•, using the same •dyadic-indication• □o□. This is the first requirement. However, since the mode of the •operand• □2.3□ cannot be firmly coerced to •boolean•, this attempted identification of •operators• fails and we must search in the next outer •range•. This next outer •range• also contains an •operation-declaration•, in line 3, but again the identification fails since the mode of □x□ cannot be firmly coerced to •integral•. (Note that it is sufficient to have the failure occur in only one •operand•.) We must now search in the next outer •range•, which contains yet another •operation-declaration•, in line 2, using the same •dyadic-indication•. This time the identification succeeds since the mode of both □2.3□ and □x□ can be firmly coerced to •real•. The value yielded by the •formula• is therefore ■3.14■.

8.8 Elaboration of formulas

In section 5.1 we discussed the elaboration of a •call•. The elaboration of a •formula• is similar. As an example, consider the •clause•

```
□¢1¢     ( priority max = 7 ;
 ¢2¢       op max = (real a, b) real :
 ¢3¢          ( a > b | a | b ) ;
 ¢4¢       x := 3.14 max y )□
```

Here the •operator• □max□, in line 2, possesses the routine ■(real a = skip, real b = skip ; real : (a > b | a | b))■ . The elaboration of the •formula•, in line 4, then has the following effect. In a copy of the routine possessed by □max□, the two □skip□s are replaced by the •operands• of the •formula•. The resulting object

□(real a = 3.14, real b = y ; real : (a > b | a | b))□ ,

which is a •closed-clause•, replaces the •formula• and is elaborated. Its value is then the value of the •formula•. There is therefore nothing new to tell about the elaboration of •formulas•.

Since it seems that each operation in a •formula• involves a sequence of actions like those in the elaboration of a •call•, it may be thought that the execution of ALGOL 68 programs will be necessarily slow. This need not be the case, for the implementer will undoubtedly produce in-line code for the translation of a •formula• like □x + y□ (perhaps only one machine instruction). Provided that the effect is the same, he is free to produce any machine instructions for doing the job (see the note after 10.b Step 12 in the Report).

8.9 Monadic operators

The most significant fact concerning •monadic-operators• is that they are always of priority ten. There are no •priority-declarations• for •monadic-operators•. Because of this, monadic operations are always performed first. This is a simple rule and is easy to remember. It means that the value of □-1 ** 2□ is ■1■ and not ■-1■, contrary to its meaning in ALGOL 60 and in FORTRAN. The reason for making this choice has been explained earlier in section 3.11.

Because of the syntax

```
monadic formula : monadic operator ; monadic operand.
monadic operand : monadic formula ; secondary.
```

[R.8.4.1.f,g], the elaboration of a •formula• containing a sequence of •monadic-operators• proceeds from right to left.

Thus the •formula•

□bin round - x□

is elaborated in the order suggested by □bin (round (- x))□. A sketch of the parse of this •formula• is shown in figure 8.9.

```
                    formula
                       |
          r------------+------------,
      operator                  operand
          |                        |
          |          r-------------+-------------,
          |      operator                    operand
          |          |                          |
          |          |             r------------+-----------,
          |          |         operator              operand
        --+--     ---+---          +                    +
        □bin      round            -                    x□
```

Fig.8.9

The identification of •monadic-operators• proceeds as for the •dyadic-operators•, the only difference being that there is only one •operand• which must be checked against the only •formal-parameter• in the monadic •operation-declaration•. As for •dyadic-operators•, the mode of the •operand• must be firmly coerceable to that of the •formal-parameter•. An example is

```
□¢1¢      ( op m = (bool a) int : ( a | 100 | 0 ) ;
 ¢2¢        ( op m = (int a) int : 200 ;
 ¢3¢          m true ))□
```

in which the •operator• □m□ , in line 3, identifies the •operator• in line 1, since the value possessed by □true□ cannot be firmly coerced to a value of mode •integral•. The value of the •formula• □m true□ is therefore ■100■.

8.10 Related modes

Two modes are "related" if each of them can be firmly coerced from one same mode [R4.4.3.b]. An example is the pair of modes specified by □ref real□ and □proc real□. These are related because both can be firmly coerced from the mode specified by □ref real□. (We shall find it convenient here to shorten the phrase "the mode specified by □m□" to "the mode □m□", or even to "□m□".) Thus □ref real□ may be coerced to □ref real□, by the empty coercion, and to □proc real□, by dereferencing and then proceduring. One reason for defining this relationship between modes is to exclude some dubious unions from proper •programs• [R.4.4.3.d]. Consider, for example, the •declaration•

□union(proc real, ref real) pr := x□ .

Since □x□ is in a strong position it may be subjected to dereferencing, proceduring and then uniting, whereupon the assignment can occur. On the other hand the assignment can also occur with an immediate uniting of □x□. There is thus an ambiguity. For this reason, unions of related modes are excluded from proper •programs•.

Another reason, which has to do with •operators•, may become clear by examining the following:

```
¤(op m = (proc real) int : 0 ;
  op m = (ref real) int : 1 ;
  x := 3.14 ; i := m x)¤                  .
```

What is the value assigned to ¤i¤? Is it •0• or •1•? Since ¤x¤ may be firmly coerced both to the mode ¤ref real¤ and to the mode ¤proc real¤, it is clear that there are two defining occurrences of the •operator• ¤m¤ in the same range. This possibility must also be excluded from proper •programs• [R.4.4.3.d].

A first attempt to achieve this exclusion might be by forbidding the occurrence of two •operation-declarations•, in the same •range•, if their corresponding •operands• are of related modes. However, this is not enough as the following example shows:

```
¤( op + = ([ ]ref real a, b) real : 0.0 ;
   op + = ([ ]real a, b) real : 1.0 ;
   x1 := (x, y) + (y, x) )¤              .
```

In this example the modes ¤[ ]real¤ and ¤[ ]ref real¤ are not related, nevertheless we have two defining occurrences of the same operator ¤+¤, as used in the •formula• in the last line. It is for this reason that the concept of "loosely related" is developed in the Report. For most programmers and most implementers, this concept is sufficient to exclude multiple definitions of •operators•. It has been shown that there are certain pathological cases which can still slip through into proper •programs•. For a discussion of these the reader is referred to a paper by Wössner and the discussion following it [W]. A new wording of the context condition [R.4.4.3.b] is thus likely to appear in the revised Report.

8.11 Peano curves

In the following example we assume that there is a plotting device and a •library-prelude• (for plotting) containing •declarations• of the •identifiers• ¤x, y, plot¤ and ¤move¤. Both ¤x¤ and ¤y¤ are •real-variables•, the two coordinates of the plot pen. The •procedure• ¤plot¤ first lowers the pen and then plots a straight line from its current position to the position whose coordinates are ¤(x, y)¤. The •procedure• ¤move¤ first raises the pen and then moves it to the position ¤(x, y)¤.

In mathematics it is known that a uniformly convergent sequence of continuous curves (e.g., polygonal lines) will converge to a continuous curve. The particular example we have in mind is a sequence which defines a continuous curve passing through every point of a square. It helps in proving that the points of a square are in one-to-one correspondence with the points of a line interval. These are known as the Peano curves. The plotting of the approximants is an interesting exercise (provided that one has plenty of computing money) and the resulting figures are aesthetically pleasing.

Suppose that one begins with a square of side ¤d¤. The first approximant (n = 0) is a single point at the centre of the square. To obtain the second approximant (n = 1), one divides the original square into four squares each of side □d / 2□. The solution for the case n = 0 is then applied to each of the four small squares. The four plots so obtained are then joined

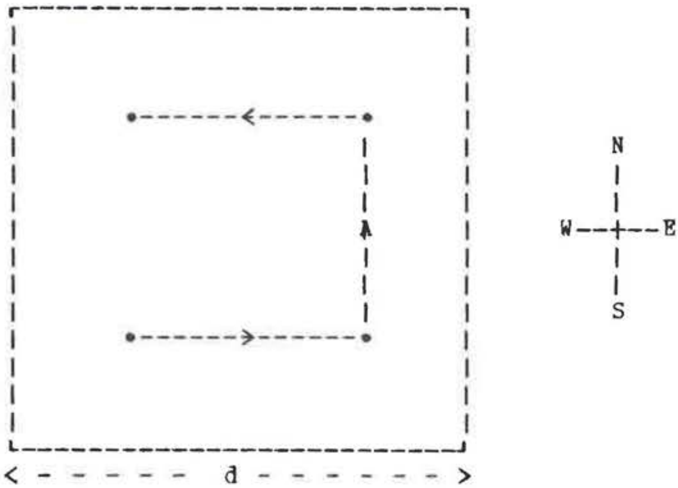


Fig.8.11.a

by three lines of length □d / 2 ** 1□ in the directions first E, then N and then W. The resulting plot is shown in figure 8.11.a. The process is recursive, but perhaps we should follow it one more step. The next approximant (n = 2) is shown in figure 8.11.b, in which the method is to apply the solution for the

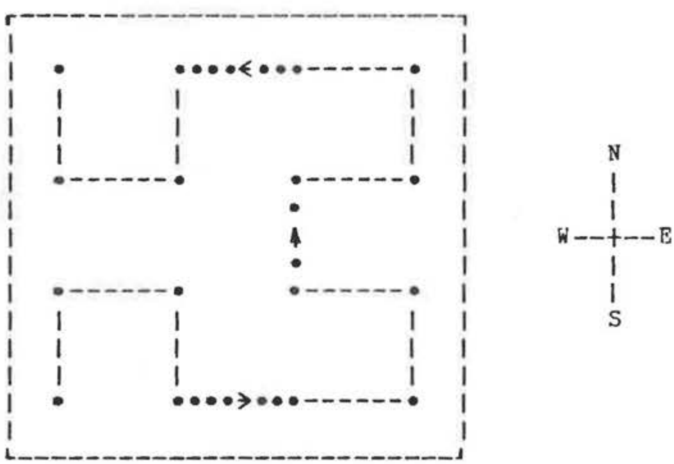
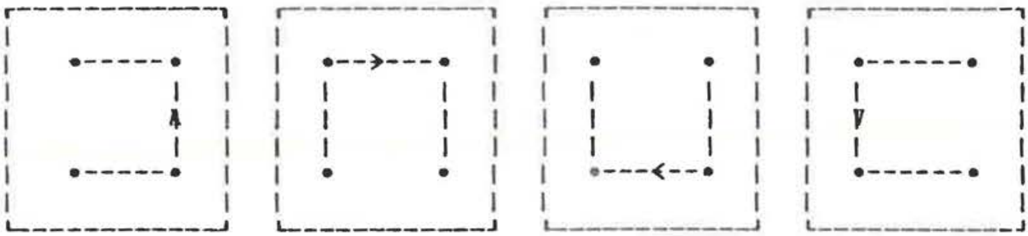


Fig.8.11.b

case n = 1 to the four quarters, but scaled down and re-oriented. These four plots are again joined by straight lines of length □d / 2 ** 2□ and in the same directions as before, i.e., first E, then N and then W.

To plot these approximants we consider some orientations of the case n = 1. A moment of thought will convince us that we need only four orientations and these are shown in figure 8.11.c, together with a pair of truth values (the first related to rotation about the NE diagonal and the second related to rotation about the NW diagonal) and the direction of the second



(true,true) N (false,true) E (true,false) W (false,false) S

Fig.8.11.c

of the three straight lines, either of which will determine one of the four orientations. In the reach of □bool p, q□, the •formula• □p * q□ plots an approximant with the orientation □(p, q)□. and the •formula• □p + q□ plots a straight line of the required length and with orientation □(p, q)□.

The program[1] to plot an approximant follows. It first reads the length □d□ of the side of the square and the degree □n□ of the approximant. The first step is to calculate the length of the line segments required and then to move the pen to the starting position. The plot is then driven by the •formula• □true * true□.

```
□begin ¢Peano curve approximant¢
op + = (bool p, q) : ¢this plots a straight line of length d¢
   (( p = q | y | x ) +:= ( q | d | -d ) ; plot ) ;
op * = (bool p, q) : ¢a recursive operation¢
     ( n > 0
     | n -:= 1 ; ¬p * q ; ¬p + q ; p * q ; p + q ; p * q ;
       p + ¬q ; p * ¬q ; n +:= 1
     ) ;
real d ¢the side of the square¢,
int n ¢the degree of the approximant¢ ;
start here : read((d, n)) ;
d /:= 2 ** n ¢length of connecting segments¢ ;
x := y := d / 2 ; move ¢to the starting point¢ ;
¢now plot it¢ (true * true)
end□
```

-----
(1) From an algorithm of A. van Wijngaarden.

8.12 Chinese rings

The next example is a solution to the puzzle of the Chinese rings. The puzzle may be stated as follows. There are □n□ rings with an elongated D shaped rod passing through them; the rings are attached, by wires through the D shaped rod, to a plate; this is done in such a manner that, if the first □m - 2□ rings have been removed, then the □m□th ring may be removed (or replaced) but not the □m-1□th ring. The problem is to remove all the rings. The solution is by induction [1]. Removal of rings 1 and 2 is done in the order "remove 2, remove 1". Assuming that we know how to remove (and therefore to replace) less than □m□ rings, then all □m□ rings are removed as follows: "remove m-2 rings, remove ring m, replace m-2 rings, remove m-1 rings".

In the following program[2] the •formula• □k down i□ removes □k - i□ rings. The •formula• □k up i□ replaces □k - i□ rings. The •formula• □n down 0□ then drives the algorithm by removing all the □n□ rings.

```
□begin
op down = (int a1, b) :
   ( int a := a1 ;
     (( a -:= b) > 0
     | a down 2 ; print(("remove", a)) ; a up 2 ; a down 1 )) ;
op up = (int a1, b) :
   ( int a := a1 ;
     (( a -:= b ) > 0
     | a up 1 ; a down 2 ; print(("replace", a)) ; a up 2 )) ;
int n ;
start here : read(n) ; n down 0
end□
```

Review questions

8.1 Formulas

a) Is □x := y□ a •formula•?
b) Is □x +:= y□ a •formula•
c) What is the order of elaboration of
   □x + - y - - - abs i over 2□?
d) How many priority levels are there for •dyadic-operators•?
e) Is □x :=: y□ a •formula•?
f) What is the value of □7 - 3 - 2□?

8.2 Priority declarations

------

(1) D.O.Shklarsky, N.N.Chentzov, I.M.Yaglom, The USSR Olympiad Problem Book, Freeman & Co. 1962, pp 80-84.

(2) This algorithm is due to Sharon Dyck and in its final form to W.L.van der Poel.

a) Is ¤priority :=: = 1¤ a •priority-declaration•?
b) Is ¤priority +:= = 0¤ a •priority-declaration•?
c) Is ¤priority m = 10¤ a •priority-declaration•?
d) Is ¤priority ? = 5¤ a •priority-declaration•?
e) Is ¤priority ?, ! = 6¤ a •priority-declaration•?

8.3 Operation declarations

a) Is ¤op :=: = (ref real a, b) : a = b¤ an •operation-declaration•?
b) Is ¤op t = (: true)¤ an •operation-declaration•?
c) Is ¤op * = (real a) real : exp(a)¤ an •operation-declaration•?
d) Is ¤op or = (ref real x, y) ref real : (random > .5 | x | y)¤ an •operation-declaration•?
e) Declare an •operator• ¤create¤ so that ¤f create n¤ has the same value as ¤create(f, n)¤ [R.10.5.1.2.c].

8.4 Elaboration of operation declarations

a) What is the value possessed by ¤o¤ in the reach of ¤op o = (real a) int : round a¤?
b) Is ¤op (real) real o = random¤ an •operation-declaration•?
c) What is the value of the •formula• ¤"+123" si ("+1000" si 2)¤ using the declaration of ¤si¤ as in 8.3?
d) Is ¤op or = (proc bool a, b) bool : (a | true | b)¤ an •operation-declaration•?
e) Is ¤op (real, real) real a = +¤ an •operation-declaration•?

8.5 Dyadic indications and operators

a) How many defining occurrences may be identified by an applied occurrence of a •dyadic-indication•?
b) How many operator defining occurrences of ¤+¤ are in the •standard-prelude•?
c) How many •priority-declarations• are in the •standard-prelude•?
d) Where is the •priority-declaration• for the •operator• ¤?¤ in line 3 of 10.5.3.i in the Report?
e) Is ¤::=¤ a •dyadic-indication•?

8.6 Identification of dyadic indications

a) Is ¤priority + = 8, + = 9¤ a •priority-declaration•?
b) Can a proper •program• contain ¤(priority abs = 9 ; x := abs x)¤?
c) Why does the S occur in the description of the repetitive statement [R.9.2.a,b, 9.c]?
d) Are •dyadic-indications• subject to protection?
e) Are •operators• subject to protection?

8.7 Identification of operators

a) In line 11.11.y of the Report, the •formula• ¤value of ec - 1¤ occurs. Where is the defining occurrence of its •operator•?

b) In line 11.11.at of the Report, the •formula• □f - one□ occurs. Where is the defining occurrence of its •operator•?
c) In line 11.11.l of the Report, the •formula• □a = zero□ occurs. Where is the defining occurrence of its •operator•?
d) Where is the defining occurrence of the •operator• □or□ in the •formula• □101 or bin 6□?
e) Where is the defining occurrence of the •operator• □<□ in the •formula• □"a" < (string :)□?

8.8 Elaboration of formulas

a) What is the value possessed by □t□ in □op t = (real a) bool : a > 0□?
b) What •closed-clause• is elaborated as a result of the elaboration of the •formula• □t x□ in the reach of the •declaration• above?

8.9 Monadic operators

a) What is the value of □2 + - - + - 3□?
b) Is □x :=: y□ a •formula•?
c) Is □x +:= real : random□ a •formula•?
d) Is □real + real□ a •formula•?
e) What is the value of □-1 i 2 = -1 i -2□?

8.10 Related modes

a) Are the modes □proc int□ and □real□ related?
b) Are the modes □ref ref int□ and □ref proc int□ related?
c) Are the modes □proc union(int, real)□ and □union(proc int, bool)□ related?
d) Can the •declarer• □union(proc real, proc)□ be contained in a proper •program•?
e) Can □(op - = (union(bool, ref char) a) int : 2 ; op - = (union(ref int, char) a) : 3 ; -(char := "a"))□ be contained in a proper •program•?

8.11 Peano curves

a) What would the •formula• □false + false□ accomplish?
b) Write this algorithm using four mutually recursive procedures.
c) Translate the algorithm into FORTRAN.

8.12 Chinese rings

a) What is printed by □2 down 0□?
b) What is printed by □3 down 0□?
c) What is the purpose of the •declaration• □int a := a1□?
d) What is printed by □6 down 2□?
e) Rewrite this algorithm without using •operation-declarations•.

9 The grammar

9.1 The syntactic elements

The grammar of ALGOL 68 is written using both "small-" and "large syntactic marks" (the lower and upper case letters of the alphabet) [R.1.1.2.a]. Thus, •base• consists of four small syntactic marks and •MODE• consists of four large syntactic marks. A sequence of zero or more small syntactic marks is a "protonotion" [R.1.1.2.b]. For example, •base• is a protonotion and so is •streets-that-flow-like-a-tedious-argument•, though the latter will not be found in the ALGOL 68 grammar. (The presence of hyphens within protonotions may be ignored.)

The syntax of ALGOL 68 is a set of "production rules of the strict language" ("production rules", for short). A production rule is a protonotion followed by a colon followed by a list of protonotions separated by commas and followed by a point. A "notion" is a protonotion for which there is a production rule, i.e., it lies to the left of the colon in some production rule. For example, •integral denotation• is a notion because of the existence of the production rule

•integral denotation : digit token sequence.•

[R.5.1.1.1.a], but •base• is not, for there is no production rule for it [R.8.6.0.1.a].

Any protonotion ending with •symbol•, e.g., •begin-symbol•, is a "symbol".

A "direct production" of a notion is the part between the colon and the point in a production rule for that notion. Thus, •digit-token-sequence• (see above) is a direct production of •integral-denotation• and •insertion-option, radix, letter-r• is a direct production of •radix-mould• [R.5.5.2.b]. The direct production of a notion is therefore a list of protonotions (the "members") separated by commas [R.1.1.2.b].

A direct production of a notion is also a "production" of that notion. If in a production of a given notion, some notion ("productive member") is replaced by one of its productions, then the result is also a production of the given notion. This replacement process may be repeated as often as we please and, in parsing, normally continues until all the notions have been replaced and the result is a list of symbols. Then we have a "terminal production" of the given notion. For example,

•digit one symbol, digit two symbol•

is a terminal production of the notion •integral-denotation•.

9.2 Two levels

The syntax of ALGOL 68 is a set of production rules for notions (the production rules of the strict language) as described in section 9.1 above. Only a few of the actual production rules are explicitly given in the Report. The number of production rules is infinite and the rule

•integral denotation : digit token sequence.•

[R.5.1.1.1.a] is one of them. The others may be obtained, when required, from a two level grammar which we shall now describe. A typical production rule of the strict language is

•reference to real assignation :
  reference to real destination, becomes symbol, real source.•

It is obtained from the rule in the Report

•reference to MODE assignation :
  reference to MODE destination, becomes symbol, MODE source.•

[R.8.3.1.1.a], by replacing the metanotion •MODE• consistently by one of its terminal productions, viz., •real•. The rules of the Report are called simply "rules" without further qualification. We shall be speaking of several different sets of rules, so it is perhaps just as well to use the word "hyper-rule" for the rules (such as the one just given) found in Chapters 2 up to 8 of the Report, especially if there may be some doubt about which set of rules we are referring to. A hyper-rule thus differs from a production rule of the strict language in that it may contain zero or more metanotions and zero or more semicolons. A production rule of the strict language contains no metanotions and no semicolons.

Another set of rules is the "metarules". These are found in Chapter 1 of the Report. A typical metarule is

•FORESE : ADIC formula ; cohesion ; base.•

[R.1.2.4.c]. A metarule may be distinguished from other rules by the fact that it has one "metanotion" (a sequence of large syntactic marks) to the left of the colon and zero or more semicolons to the right. However this is not sufficient to recognize one, for

•DIGIT : DIGIT symbol.•

[R.3.0.3.d] is a hyper-rule, not a metarule. From the metarules we may derive the production rules of the metalanguage in a rather simple way.

Thus, in summary, the ALGOL 68 grammar consists of two sets of rules

(i) the metarules (in Chapter 1) and
(ii) the hyper-rules (in Chapters 2 up to 8).

The production rules for the strict language are derived from both the metarules and the hyper-rules by a process which we shall explain, by example, in section 9.5.

9.3 The metarules

A typical metarule is

•FORESE : ADIC formula ; cohesion ; base.•

[R.1.2.4.c]. It provides three production rules for the metalanguage, which are

•FORESE : ADIC formula.• ,
•FORESE : cohesion.• ,

and

•FORESE : base.• .

Thus a production rule of the metalanguage contains no semicolons. The two direct productions •cohesion• and •base• are terminal (in the metalanguage), but the direct production •ADIC formula• may be produced further by using the metarule for

•ADIC• [R.1.2.4.d]. The terminal productions of metanoticns are always protonotions.

The words used for the metanotions are usually chosen in such a way that they help to convey a meaning. Coined words, such as •FORESE• are often mnemonic. Thus, •FORESE• is made up from

<u>for</u>mula coh<u>e</u>sion ba<u>se</u>

and FEAT from

<u>f</u>irm w<u>ea</u>k sof<u>t</u> .

The reader will find many others, similarly coined and usually the mnemonic is glaringly apparent. It is useful to remember that every metanotion ending with •ETY• always has •EMPTY• as one of its (not necessarily direct) productions.

The metanotion •ALPHA• is of interest because it has all the letters of the alphabet (small syntactic marks [R.1.1.2.a]) as direct productions. If more are required (perhaps in languages other than English), then it is permitted to add them (see 1.1.4 Step 2 in the Report).

Another metarule of significance is

•EMPTY : .•

[R.1.2.1.i], from which we see that the metanotion •EMPTY•, if it appears in one of the hyper-rules, or in those derived from them, may be consistently deleted.

Two metarules to watch are

•CLOSED : closed ; collateral ; conditional.•

[R.1.2.3.r] and

•LIST : list ; sequence.•

[R.1.2.5.h], where a distinction must be made between the metanotion, which appears on the left of the rule, and the first production of each, which is a protonotion. In speech this distinction will be lost.

Another interesting metarule is

•NOTION : ALPHA ; NOTION, ALPHA.•

[R.1.2.5.f]. Roughly speaking, anything is a terminal production of •NOTION•. More precisely, any sequence of small syntactic marks (the letters of the alphabet as used in the syntax) is a terminal production of •NOTION•. This is so because the productions of •ALPHA• are the small syntactic marks. This fact is used heavily in the rules of section 3.0.1 of the Report.

One might also wonder about the metarules

•LMODE : MODE.•

and

•RMODE : MODE.•

[R.1.2.2.j,k]. The mystery may be resolved by examining the rule for •formulas• [R.8.4.1.b], where the mode of the left •operand•, that of the right •operand• and that of the result delivered by the operation all appear in the same hyper-rule. These modes may be different, so it would not do to use the metanotion •MODE• for all three of them. Other instances of this same phenomenon are suggested by the metarule

•LOSETY : LMOODSETY.•
[R.1.2.2.o], which is used in the hyper-rule for •united-declarers• [R.7.1.1.ee,ff], and by
•ROWWSETY : ROWSETY.•
[R.1.2.2.d] used in the hyper-rule for •slices• [R.8.6.1.1.a], where •ROWWSETY• counts the number of •row-of•s not involved in the •indexer• and •ROWSETY• counts the number of •trimscripts• which are •trimmers•.

The two rules
•LFIELDSETY : FIELDS and ; EMPTY.•
and
•RFIELDSETY : and FIELDS ; EMPTY.•
[R.1.2.2.q,r] are another pair which play a similar role in the rule for •selections• [R.8.5.2.1.a].

There are two metarules in which the only direct production of the metanotion is a protonotion. They are
•COMPLEX : structured with real field letter r letter e and real field letter i letter m•
[R.1.2.2.s] and
•LENGTH : letter l letter o letter n letter g.•
[R.1.2.2.v]. This means that the presence of one of these metanotions in some hyper-rule is merely for the convenience of shortening the rule and plays no other grammatical role.

9.4 The hyper-rules

A good introduction to the hyper-rules is to be found in section 3.0.1 of the Report, where are collected together several rules which should be mastered early, for they are used extensively elsewhere. A typical example is
•NOTION option : NOTION ; EMPTY.•
[R.3.0.1.b]. The first step in deriving production rules of the strict language, from the hyper-rules, is to make two new rules as follows:
•NOTION option : NOTION.•
and
•NOTION option : EMPTY.•
As a next step we may replace each metanotion consistently by one of its terminal productions. For example, we might substitute •integral-part• for •NOTION• and nothing at all for •EMPTY•. This will now give us two production rules of the strict language. They are
•integral part option : integral part.•
and
•integral part option : .•

Note that •integral-part-option• means what the words suggest. i.e., either the presence or absence of an •integral-part•. This is used with good effect in the rule
•variable point numeral :
integral part option, fractional part.•
[R.5.1.2.1.b]. Examples are ¤3.45¤ and ¤.45¤. Many of the notions in ALGOL 68 are similarly chosen so that the words (protonotions) used give some suggestion of the semantic

elaboration.

The pair of hyper-rules
•NOTION pack : open symbol, NOTION, close symbol.•
and
•NOTION package : begin symbol, NOTION, end symbol.•
[R.3.0.1.h,i] are also used in several places elsewhere. Thus, if ¤x¤ is a certain •n•, then ¤(x)¤ is an •n-pack• and ¤begin x end¤ is an •n-package•.

The hyper-rule
•NOTION LIST proper : NOTION, LIST separator, NOTION LIST.•
[R.3.0.1.g] ensures that at least two •NOTION•s will appear in the production. It is used, for example, in the rule for •collateral-declarations• [R.6.2.1.a]
•collateral declaration : unitary declaration list proper•
meaning that, for example, ¤real x, int i¤ is a •collateral-declaration• but ¤real x¤ is not.

The hyper-rules
•NOTION LIST :
chain of NOTIONs separated by LIST separators.•
and
•chain of NOTIONs separated by SEPARATORs : NOTION ;
NOTION, SEPARATOR,
chain of NOTIONs separated by SEPARATORs.•
[R.3.0.1.d,c] are used to describe such objects as
¤123¤ ,
which is a •chain-of-digit-tokens-separated-by-EMPTYs•,
¤1, 2, 3¤ ,
which is a •chain-of-strong-integral-units-separated-by-comma-symbols•, and
¤1 ; 2 ; 3¤ ,
which is a •chain-of-strong-integral-units-separated-by-go-on-symbols•. These are used principally in the rules for •serial-clauses• [R.6.1.1], but in other places also.

9.5 A simple language

We shall now use this kind of grammar to describe an interesting but trivial language. By this small example we shall be able to see the complete grammar in a few lines. There are only three •symbols•, two hyper-rules and two metarules. Thus it will be easier to get an overall view of how the grammar works.

The language we choose is that in which the only sentences (or programs) are
¤xyz¤, ¤xxyyzz¤, ¤xxxyyyzzz¤ ... .
Perhaps we could say that the following would cause an ALGOL 68 computer to print sentences of this language until it runs out of time or memory space.

```
¤begin string a, b, c ;
do print((a +:= "x") + (b +:= "y") + (c +:= "z"))
end¤ .
```

The reason that this language is of interest is that it is known [H] that it cannot be described by a context-free grammar such as that used for the syntax of ALGOL 60.

The three symbols of the language and their representations are

symbol	representation
•letter x symbol•	□x□
•letter y symbol•	□y□
•letter z symbol•	□z□ .

This corresponds to the whole of section 3.1.1 of the Report. The three hyper-rules are

(i) •sentence :
NUMBER letter x, NUMBER letter y, NUMBER letter z.• ,
(ii) •NUMBER plus one LETTER : NUMBER LETTER, one LETTER.• ,
(iii) •one LETTER : LETTER symbol.• .

These three rules correspond to all the hyper-rules found in Chapters 2 up to and including 8 of the Report. Rule (i) expresses the requirement that the number of occurrences of each of the different letters should be the same. Rule (ii) will be used to interpret this number, i.e., actually to count them out one by one. Rule (iii) is almost the same as the hyper-rules 3.0.2.b and 3.0.3.d of the Report. Rule (ii) might be compared with 7.1.1.q of the Report, where the multiplicity of a •rower• is being counted. Rule (iii) is present in order to satisfy the requirement of ALGOL 68 that only protonotions ending in •symbol• are terminal productions of the grammar. Without this requirement we could describe the language with two hyper-rules instead of three.

The two metarules are
(I) •LETTER : letter x ; letter y ; letter z.• ,
(II) •NUMBER : one ; NUMBER plus one.• .

These two metarules correspond to the metarules found in section 1.2 of the Report. The first metarule, (I), is there so that we may be able, with one word, to speak of any one of the letters. It is similar to the metarule 1.2.1.t of the Report for the metanotion •ALPHA•. We could do without metarule (I), but then we should need seven hyper-rules instead of three. Metarule (II) is essential. In it, •NUMBER• is used as a counter. The terminal productions of the metanotion •NUMBER• are •one•, •one-plus-one•, •one-plus-one-plus-one• and so on. The metarule is somewhat similar to the metarule of the Report for the metanotion •ROWS• [R.1.2.2.b].

We shall now go through, in detail, the process of finding some of the production rules of the strict language, as defined by the above grammar. This process is described in sections 1.1.4 and 1.1.5 of the Report. Since there are infinitely many production rules of the strict language (even for the minilanguage above), we cannot give them all here.

If we substitute the first terminal production of •NUMBER•, viz., •one•, for that metanotion, in hyper-rule (i), it yields a new rule
(a) •sentence : one letter x, one letter y, one letter z.• .
The direct production of •sentence• in this new rule is not terminal, since it contains a notion which does not end with

•symbol•. To remedy this we use hyper-rule (iii) and, replacing •LETTER• by each one of its terminal productions in turn, we obtain

(b) •one letter x : letter x symbol.• ,

(c) •one letter y : letter y symbol.• ,

and

(d) •one letter z : letter z symbol.• .

The rules (a), (b), (c) and (d) are each production rules of the strict language. If now, in the right hand side of (a), we make use of the productions in (b), (c) and (d), then we obtain that

•letter x symbol, letter y symbol, letter z symbol•

is a terminal production of the notion •sentence•. This means that we may speak of □xyz□ as a •sentence• in the representation language.

We now take another terminal production of •NUMBER•, viz., •one-plus-one•, and substitute that in the hyper-rule (i). It yields

(e) •sentence : one plus one letter x,
one plus one letter y, one plus one letter z.• .

Also, in (ii), we replace •NUMBER• by •one•. (Note that this is the first use of hyper-rule (ii).) This gives

(f) •one plus one letter x : one letter x, one letter x.• ,

(g) •one plus one letter y : one letter y, one letter y.•

and

(h) •one plus one letter z : one letter z, one letter z.• .

Now, combining production rules (e), (f), (g) and (h) with production rules (b), (c) and (d) obtained above, we have that the object

•letter x symbol, letter x symbol, letter y symbol,
letter y symbol, letter z symbol, letter z symbol•

is also a terminal production of •sentence•. In the

```
                           sentence
                              |
       ┌----------------------+----------------------┐
  one-plus-one-          one-plus-one-          one-plus-one-
    letter-x               letter-y               letter-z
       |                      |                      |
  ┌----+-----┐           ┌----+-----┐           ┌----+-----┐
 one-       one-        one-       one-        one-       one-
letter-x  letter-x    letter-y  letter-y    letter-z  letter-z
  |          |          |          |          |          |
letter-x- letter-x-   letter-y- letter-y-   letter-z- letter-z-
 symbol    symbol      symbol    symbol      symbol    symbol
  |          |          |          |          |          |
 □x          x          y          y          z          z□
```

Fig.9.5

representation language we may therefore now say that

□xxyyzz□

is a •sentence• of the strict language. A sketch of the parse of this •sentence• is shown in figure 9.5. Perhaps we have now done enough of this to suggest that it is easy to show that □xxxyyyzzz□ is a •sentence•. A crucial new rule in this process

is
  •one plus one plus one LETTER :
    one plus one LETTER, one LETTER.• ;
moreover, the process for finding more •sentences• of the language should be clear.

It will also be obvious that the same language might be described more concisely by the grammar

```
  (I)   L : x ; y ; z.              (i)   s : N x, N y, N z.
  (II)  N : ; N p.                  (ii)  N p L : N L, L.
                                    (iii) L : L symbol.
```

and if we drop the requirement that every terminal must end with •symbol• by agreeing that •x, y• and •z• are already terminals, then even more concisely by

```
  (I)   L : x ; y ; z.              (i)   s : N x, N y, N z.
  (II)  N : ; N p.                  (ii)  N p L : N L, L.       .
```

For the student of formal grammars this is more natural, for he is by nature an algebraist who is dedicated to the cult of concise expression. In a description of a practical programming language we can afford to be more verbose so that even those who are not algebraists can read the rules and think that they understand them.

9.6 How to read the grammar

How do we really use a grammar such as the one we are considering? How do we read it? Is it necessary always to perform, in our minds, the replacement of the metanotions by their terminal productions before we can understand what the hyper-rules say? The answer to this is probably that we should have the experience of making these detailed substitutions at least once. With this experience we may then proceed as does the mathematician who finds that it is unnecessary to prove a theorem every time that he uses its result. His method is normally to check through the proof of the theorem at least once and then to remember its hypothesis and its conclusion.

For us, the metalanguage plays the role of a body of theorems and the results we need to remember are the shape of the terminal productions of the metanotions. For example, in the grammar of the minilanguage given in the last section, we need only remember that the terminal productions of •LETTER• are •letter-x-symbol•, •letter-y-symbol• and •letter-z-symbol• and that the terminal productions of •NUMBER• are •one•, •one-plus-one•, •one-plus-one-plus-one• and so on. With this information at hand, the complete language may be comprehended merely by reading the three hyper-rules

```
 (i)   •sentence :
        NUMBER letter x, NUMBER letter y, NUMBER letter z.•     ,
 (ii)  •NUMBER plus one LETTER : NUMBER LETTER, one LETTER.•    ,
 (iii) •one LETTER : LETTER symbol.•                            .
```

The same method of comprehension applies to ALGOL 68. The metarules should be well studied first and the shape of the terminal productions (at least of the commonly used ones) should be known. With this knowledge we can then read the hyper-rules and comprehend their meaning.

The most important metanotion in ALGOL 68 is •MODE•. For this reason its terminal productions should be well known before trying to read the hyper-rules. A chart is sometimes a helpful aid in understanding the metalanguage, though others may prefer to rely upon the alphabetic listing of the metarules which comes as a loose page with the Report. If you have not already done

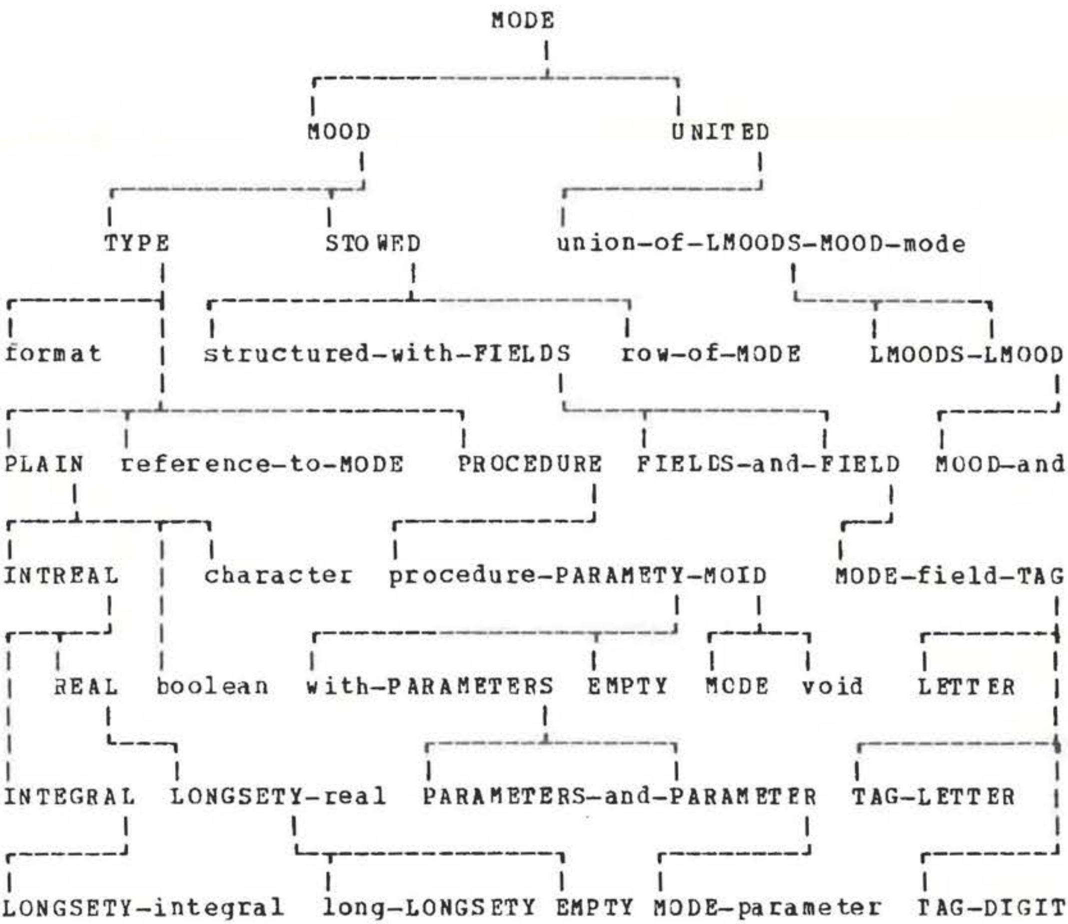


Fig.9.6

so, it is a good idea to take this loose page and arrange it so that it is attached to your copy as a fold-out page in such a way that it may be in view no matter what page of the Report you have open. For those who like charts, we reproduce, in figure 9.6, an abbreviated syntactic chart for the metanotion •MODE•, in which •LETTER• and •DIGIT• are the only metanotions not produced. Whichever method you prefer, ("people who like this sort of thing will find that this is the sort of thing they like") a careful study of the metalanguage is essential to the comprehension of the hyper-rules and thus of the grammar of the language.

9.7 The indicators

A "hypernotion" [R.1.3] is a sequence of metanotions and/or protonotions, e.g., •MODE field TAG•. A hyper-rule (in the sense used in section 9.2 above) is therefore a hypernotion followed by a colon, followed by zero or more hypernotions separated by semicolons and/or commas and followed by a point; e.g.,

```
 •strong COERCEND : COERCEND ;
    strongly ADAPTED to COERCEND.•
```

[R.8.2.0.1.d]. If, in a given hypernotion, one or more of its metanotions is consistently replaced by a production of that

```
                strongly-ADAPTED-to-COERCEND
                    ---+----    ----+---
                       |            +------+
                       |          MOID    FORM
                       |            |       |
                       |          MODE      |
                       |            |       |
                   ADJUSTED       MOOD      |
                       |            |       |
                       |          TYPE    FORESE
                       |            |       |
                       |          PLAIN     |
                       |            |       |
                       |         INTREAL    |
                  -----+-------    -+---  --+
              strongly-deprocedured-to-real-base
              ---+--                +---  --+-
                 |                 INTREAL  |
                 |                  |       |
                 |                 PLAIN    |
                 |                  |       |
                 |                 TYPE   FORESE
                 |                  |       |
                 |                 MOOD     |
                 |                  |       |
                 |                  MODE    |
               --+--               -+---  --+-
              STIRMly-deprocedured-to-MOID-FORM
```

Fig.9.7

metanotion, then we have another hyper-notion, or perhaps a protonotion. Let us call this an "offshoot" of the given hypernotion; e.g., •strongly deprocedured to real base• is a terminal offshoot of •strongly ADAPTED to COERCEND•, and •INTREAL base• is an offshoot of •MODE base•. In order to read the grammar easily, we frequently need to know whether two given hypernotions have a common offshoot. For example,

•strongly ADAPTED to COERCEND•

and

•STIRMly deprocedured to MOID FORM•

have at least one common offshoot, say

•strongly deprocedured to real base• .

That this is so can be seen by examining figure 9.7, where the steps in obtaining this offshoot are shown. In fact, examination of this same figure shows that there are infinitely many common terminal offshoots of these two hypernotions. They are all offshoots of a "maximal common offshoot", the hypernotion

•strongly deprocedured to MOID FORM• .

It is the existence of some maximal common offshoot, rather than that of any particular common terminal offshoot which becomes the point of focus when looking at two such hypernotions. Note that because of the requirement of consistent replacement, some offshoots may be too restrictive to be useful, e.g., the offshoot •procedure-with-MODE-parameter-and-MODE-parameter-MODE-PRIORITY-operator• of the hypernotion •procedure-with-LMODE-parameter-and-RMODE-parameter-MOID-PRIORITY-operator• [R.4.3.1.b].

In the process of parsing, given some hypernotion to the right of the colon in a hyper-rule, we need to know how to find a hyper-rule whose hypernotion to the left of the colon has a common offshoot with the given one. To help us in this search there are "indicators" [R.1.3]. The example considered above will actually occur in reading the Report. Consider the two hyper-rules [R.8.2.0.1.d]

```
  •strong COERCEND : COERCEND ;
     strongly ADAPTED to COERCEND {822a}.•
and [R.8.2.2.1.a]
  •STIRMly deprocedured to MOID FORM{820d} :
     procedure MOID FORM ;
     STIRMly FITTED to procedure MOID FORM.•              .
```

We have copied these two hyper-rules from the Report, together with two of the indicators, "822a" and "820d". In order to conserve space within the hyper-rules of the Report, the indicators have been compressed, according to obvious conventions [R.1.3]. If we expand them again, i.e., 822a becomes 8.2.2.1.a and 820d becomes 8.2.0.1.d, then we see that the hypernotion on the right of the hyper-rule 8.2.0.1.d points to the hyper-rule 8.2.2.1.a and the hypernotion on the left of hyper-rule 8.2.2.1.a points to hyper-rule 8.2.0.1.d. We are thus aided, in both directions, in finding hypernotions with common offshoots.

The indicators are clustered rather thickly in the hyper-rules concerning coercion, in section 8.2 of the Report. Perhaps this is evidence that it is in this section that the power of the two-level grammar is being used to its fullest. A similar, or perhaps greater, clustering of indicators might have been found in section 3.0.1 of the Report, dealing with chains, lists, sequences and options, but these have not been included in the Report since their great number would have rendered their presence of little value. Instead, the indicators have bypassed this section, which the reader is therefore advised to become familiar with at an early stage.

Sometimes a hyphen, "-", appears after a set of indicators for a hypernotion. This tells us that there is at least one offshoot of the given hypernotion which is a "dead end", i.e., it is not an offshoot of any hypernotion (on the other side of the colon) in any hyper-rule. An example of this occurs in the hyper-rule for strong coercion quoted above [R.8.2.0.1.d]. In this case it is there because, e.g.,

•strongly-widened-to-procedure-real-base•

is a dead end. It is not an offshoot of any hypernotion on the left of any hyper-rule [R.8.2.5.1]; in fact, it is not a •notion•.

Review questions

9.1 The syntactic elements

a) Is •MODE base• a protonotion?
b) Is •all-mimsy-were-the-borogroves• a protonotion?
c) Is •cast• a notion?
d) Is •MABEL identifier• a notion [R.4.4.1.b]?
e) Is •long-integral-denotation• a notion?

9.2 The metarules

a) How many production rules of the strict language are there for ALGOL 68?
b) How many production rules of the strict language are listed explicitly in section 6.1.1 of the Report?
c) How many production rules of the strict language can be derived from 7.1.1.s?
d) How many production rules of the strict language can be derived from 6.1.1.d?
e) What are the terminal productions of •VICTAL•?

9.3 The metarules

a) Is •LETTER : LETTER symbol.• a metarule?
b) How many production rules of the metalanguage can be derived from 1.2.1.r of the Report?
c) Is •NONSTOWED : TYPE ; UNITED.• a production rule of the metalanguage?
d) Are the terminal productions of •NONPROC• also terminal productions of •MODE•?
e) Is •FIELD• a production of •MODE•?

9.4 The hyper-rules

a) Is •PARAMETER : MODE parameter.• a hyper-rule?
b) Is •digit-token• a production of •digit-token-sequence-proper•?
c) Is ¤( )¤ a •strong-closed-[m]-clause•, where [m] represents some mode?
d) What production of •LFIELDSETY• would be used in parsing ¤im <u>of</u> z¤?
e) What production of •LMODE• is used in parsing ¤x + y¤?

9.5 A simple language

a) Define, by means of a two-level grammar, the language whose sentences are printed by

```
□begin string a, b := "y", c ;
 do print((a +:= "x") + (b +:= "y") + (c +:= "zz"))
 end□.
```

b) Define, by means of a two-level grammar, the language whose sentences are printed by

```
□begin string a, b, c ;
do (print(a+b+c) ; (a +:= "x", b +:= "y", c +:= "z"))
end□.
```

c) Rewrite the grammar of the language considered in 9.5 using two metarules and two hyper-rules and yet requiring that terminals end in •symbol•.

9.6 How to read the grammar

a) Is •real-format• a terminal production of •MODE•?
b) Is •reference-to-procedure-row-of-character• a terminal production of •MODE•?
c) Is •long-structured-with-real-field-letter-l• a terminal production of •MODE•?
d) Is •procedure• a terminal production of •MODE•?
e) Is •procedure-with-real-parameter-real• a terminal production of •NONPROC• [R.1.2.2.h]?

9.7 The indicators

a) Why is there a dead end in •MOID FORM• in 8.2.3.1.a of the Report?
b) What is a maximal common offshoot of •virtual NONSTOWED declarer• and •VICTAL MODE declarer• [R.7.1.1.a,n]?
c) What is a maximal common offshoot of •firmly ADJUSTED to COERCEND• and •STIRMly dereferenced to MODE FORM• [R.8.2.2.1].?
d) What is a maximal common offshoot of •STIRMly rowed to MOID FORM• and •strongly rowed to REFETY row of MODE FORM• [R.8.2.6.1]?
e) What is a maximal common offshoot of •SORTly ADAPTED to COERCEND• and •STIRMly united to MOID FORM• [R.8.2.0.1, 8.2.3.1]?

10 Mode declarations

10.1 Syntax

A typical •mode-declaration• is

□mode compl = struct(real re, real im)□

which, by virtue of extensions [R.9.2.b,c], may be written more concisely as

□struct compl = (real re, im)□ .

This •mode-declaration• is, in fact, one of the •declarations• of the •standard-prelude• [R.10.2.7.a], which means that the programmer may assume that he is within its reach (unless he has made a similar •declaration• himself). A simplified parse is

```
                         mode-declaration
                                |
     ┌──────────────────┬───────┴──┬─────────────────────┐
     |                  |          |                     |
 mode-symbol  mode-indication  equals-symbol  actual-declarer
     |                  |          |                     |
   ──┴──              ──┴──        ┴       ──────────────┴──────────
   □mode              compl        =       struct(real re, real im)□
```

Fig.10.1

shown in figure 10.1. The hyper-rule for a •mode-declaration• is

•mode declaration : mode symbol, MODE mode indication,
equals symbol, actual MODE declarer.•

[R.7.2.1.a]. The two occurrences of •MODE• here ensure that the mode of the •actual-declarer• on the right is then enveloped by the •mode-indication• on the left.

It is perhaps worth while to look at the hyper-rule

•MODE mode indication : mode standard ; indicant.•

[R.4.2.1.b] and to realise that the programmer may choose his own •indicant• more or less at will [R.1.1.5.b]. He is, however, subjected to the restrictions of his installation. It is expected that most implementations will permit such •indicants• as □abc□ and □m12□, i.e., objects which look like identifiers but are in bold face (or underlined). Objects which are •mode-standards• are □string, sema, file, compl, bits, bytes, long bytes, long long bits, long long long compl□, etc. This means that one may write

□mode file = int□

or

□mode long compl = compl□ ,

each of which is legitimate but unpleasant for the human reader.

10.2 Development

One purpose of the •mode-declaration• is to introduce a shorthand whereby the programmer may save himself trouble. If he uses some complicated •declarer•, then he may avoid writing it out in full each time that he uses it. A simple example might be a numerical analyst, working with vectors and matrices, who may wish to use the convenience of the •declaration•

□mode v = [1:n] real,
mode m = [1:n, 1:n] real□ .

In the reach of this •declaration•, he may now use these •mode-indications• as •declarers• by declaring a vector variable with □v x1□ or a matrix variable with □m x2□. It should be carefully noted that the value of □n□ which occurs in the •bounds• of these multiple variables is that which is possessed by □n□ at the time of elaboration of the •declaration• □v x1, m x2□ and not that possessed at the time of elaboration of the •mode-declaration•. An example may help to make this clear. In the reach of □int n□, the elaboration of

□n := 5 ; mode v = [1:n] real ;
n := 3 ; v x1 ; print(upb x1)□

should print the value ▪3▪ and not the value ▪5▪. This means that the •declaration• □v x1□ acts as though the □v□ were replaced by □[1:n] real□. This process is known as "developing" the •declarer• [R.7.1.2.c]. An important consequence is that, in the reach of the •declaration•

□mode v = [1:n] real,
realvec = [1:n] real□ ,

the •mode-indications• □v□ and □realvec□, when used as •declarers•, both specify the same mode. The actual •symbol• (•indicant•) chosen therefore has no influence on the mode. Observe that the same principle applies to •identity-declarations•, for

□ref int name1 = i, name2 = i□

means that both □name1□ and □name2□ possess (different instances of) the same name. In the reach of the •declaration• □mode r = [1:2]real, s = [1:3]real□, the •indicants• □r□ and □s□ also specify the same mode, when used as •declarers•; however, values of such modes may run into trouble when assigned, for then the bounds are checked [R.8.3.1.2.c Step 3].

The examples we have given are simple. However, a •mode-declaration• may be used for introducing a •mode-indication• which, when used as a •declarer•, will specify a mode which contains a reference to itself. In fact, this will normally occur in a list processing application. For such a mode, the compiler must be able to make some checks to determine whether storage space for a value of that mode is indeed possible. It is therefore not surprising that the process of developing a mode should have some rather natural restrictions.

10.3 Infinite modes

What we call here "infinite modes" are those hinted at in the last paragraph. An infinite mode will arise from the •declaration•

□struct link = (int val, ref link next)□ .

In its reach, the elaboration of

□link a := (1, link := (2, link := (3, nil)))□

will generate values linked together as shown in figure 10.3. In such a linked list, the value of the last name is ▪nil▪. If we try to write the mode specified by □link□, using small syntactic marks, it will be

•structured-with-real-field-letter-v-

letter-a-letter-l-and-reference-to-
[link]-letter-n-letter-e-letter-x-letter-t• ,
where [link] represents the same mode which we are trying to write. Since the mode contains itself, it is not unnatural to

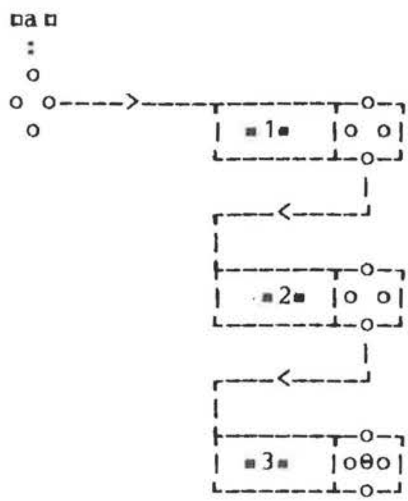


Fig.10.3

call it an infinite mode[1]. The programmer (and the compiler) however, always works with a finite formulation of that mode, so that this infiniteness need not bother him.

10.4 Shielding and showing

If we consider the mode specified by □m□, in the reach of
□mode m = struct(real v, m next)□ ,
we soon come to the conclusion that, unlike □link□ above, the field selected by □next□ contains, not a name, but a value of the same mode. Of course, this value in turn has such a field and so on ad infinitum. This is troublesome, for if we try to visualize how storage might be allocated for such a value, it is clear that it cannot be done in a computer whose storage is of finite size. It is therefore necessary to exclude such •mode-declarations• from proper •programs•. The exclusion rests upon the fact that, in this •mode-declaration•, its •actual-declarer•, □struct(real v, m next)□, "shows" [R.4.4.4.b] □m□, which is the •mode-indication• on the left. It is therefore illegal. However, in
□mode n = struct(real v, ref n next)□
the •actual-declarer• □struct(real v, ref n next)□ does not show □n□, so that this •declaration• may be contained in a proper •program•. Whether an •actual-declarer• shows a •mode-indication• rests upon whether that •mode-indication• is not "shielded" [R.4.4.4.a]. We must therefore know what is meant by

------
[1] Those who are bothered by these infinities should consult the work of C.Pair [Pa], L.Meertens [M], and W.Brown [B].

shielding a •mode-indication• before we can understand how certain •mode-declarations• can be excluded. Roughly speaking, a •mode-indication• contained in a given •declarer• is shielded if its presence in that position does not lead to difficulties in allocating computer storage for a value of the mode which that •declarer• specifies.

For the •mode-indication• □m□, examples of •declarers• in which that □m□ is shielded are

```
        □struct(int k, ref m n)□                              ,
        □ref struct(m n, char a)□                             ,
        □proc (m, int)□                                       ,
        □proc (real) m□
```

and

```
        □[1:(mode m = int ; m k ; read(k) ; k)] real□         .
```

Examples of •declarers• in which □m□ is not shielded are

```
        □m□                                                   ,
        □ref m□                                               ,
        □proc m□                                              ,
        □[1:n] m□                                             ,
```

and

```
        □union(int, m)                                        .
```

The precise definition of shielding is given in the Report [R.4.4.4.a], so we shall only paraphrase it here by saying that □m□ is shielded if there is both a □struct□ and a □ref□ to its left, or if it is in, or follows, a •parameters-pack•, or if it is essentially local to one of the bounds of the •declarer•.

As a first approximation, one may now say that a •mode-indication• which is not shielded is shown by the •declarer• containing it. We then exclude from proper •programs• all •mode-declarations• whose •mode-indication• is shown by its •actual-declarer•. This immediately excludes such undesirable objects as

```
              □mode a = a,
                    b = proc b,
                    c = ref c,
                    d = [1:n] d,
                    e = union(e, char)□                       .
```

However, examination of the •declaration•

```
                        □mode f = ref g,
                                  g = proc f□
```

reveals that we are still in trouble with the first approximation to the concept of showing. For, although □ref g□ does not explicitly show □f□, the elaboration of □ref g□ [R.7.1.2 Step 1] involves the development of □g□ and would give us the •declarer• □ref proc f□, which does indeed show □f□. It is therefore necessary to insist that we must develop all •mode-indications• which are not shielded in order to find the •mode-indications• which are shown by an •actual-declarer•. The definition of showing is carefully stated in the Report [R.4.4.4.b], so we shall not repeat it here. Perhaps the motivation given here for that careful statement is sufficient for its understanding.

10.5 Identification

Within a •serial-clause• containing a •mode-declaration•, •mode-indications• are subject to protection [R.6.0.2.d], in the same manner as are •identifiers• and •dyadic-indications•, in order that they may not become confused with the same •indication• used elsewhere. It is possible therefore to write

```
¤(mode r = real ; r x := 2 ;
   ¢└----<-----┘¢
     (mode r = int ; r x := 1
          ¢└----<----┘¢
      print(x))
  print(x))¤
```

whereupon the values printed should be •1• and •2.0•. The method of identification of the •mode-indications• is shown by "--<--".

Although this identification process is familiar (it works the same way for •identifiers•), there is one small point to be

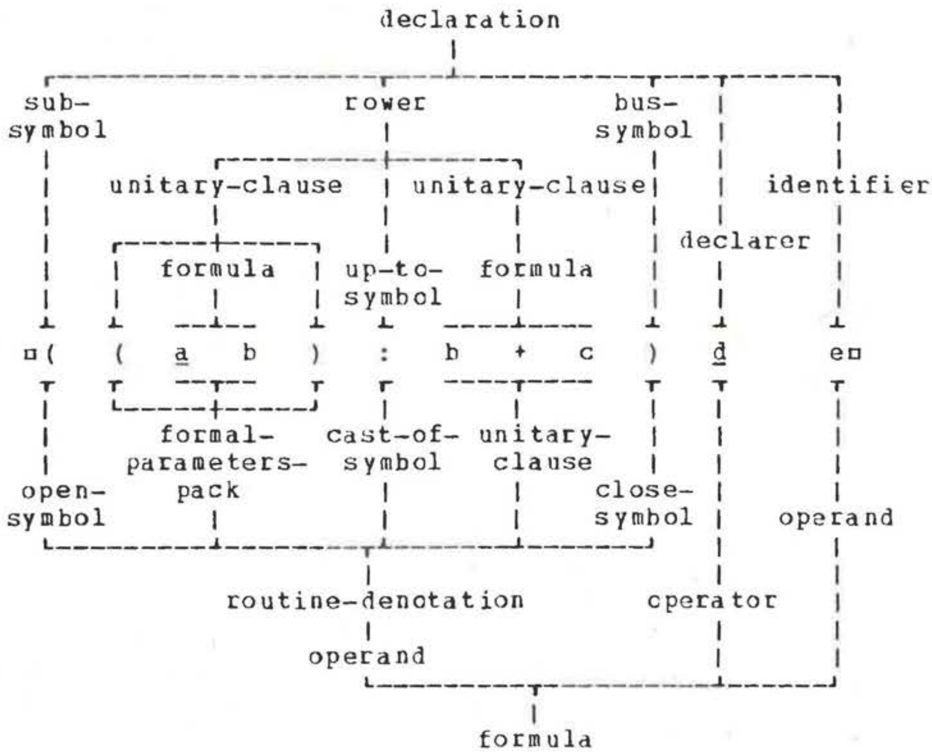


Fig.10.5

watched carefully. It is that no •indicant• may be used both as a •mode-indication• and as a •monadic-indication• [R.1.1.5.b]. The reason for this is best shown by the following example.

```
¤¢1¢      begin int b, c, e ; ¢ ... ¢
 ¢2¢        begin mode a = real ;
 ¢3¢        ((a b) : b + c) d e
 ¢4¢        ¢ ... ¢
```

```
¢5¢          end ;
¢6¢          op a = (int x) int : 1 + x ;
¢7¢          ¢ ... ¢
¢8¢          mode d = bool
¢9¢          ¢ ... ¢
¢10¢         end□
```

The problem here is whether □(a b) : b + c□ is a •row-of-rower• (remember that it is permitted to replace □[ ]□ by □( )□ [R.9.2.q]) and therefore □((a b) : b + c) d e□ is a •declaration•, or whether □((a b) : b + c)□ is a •routine-denotation• and therefore □((a b) : b + c) d e□ is a •formula•. These two possibilities are sketched in figure 10.5. If it were such that □a□ could be used as a •mode-indication• in line 2, and again as a •monadic-indication•, in line 6, then confusion would reign, for the matter can only be resolved when we meet the •declaration• of □d□ in line 8. If we now make it illegal to use □a□ both as a •monadic-indication• and as a •mode-indication•, then this unhappy situation does not arise. For those interested in compilation problems, this example shows why it is necessary to identify all •mode-indications• before a detailed parse of the •program• is made, for the identification of the second occurrence of □b□ on line 3 depends upon the information discovered in line 6.

10.6 Equivalence of mode indications

In the •mode-declaration•

```
□mode a = ref real,
      b = ref real□
```

it is rather obvious that both □a□ and □b□, when used as •declarers•, specify the same mode. However, since a •mode-declaration• has the possibility of depending on other •mode-declarations•, or on itself, one may make several •mode-declarations• like

```
□struct a = (ref a left, ref a right),
        b = (ref b left, ref struct
                  (ref b left, ref b right) right),
        c = (ref d left, ref e right),
        d = (ref e left, ref c right),
        e = (ref c left, ref d right)□                ,
```

in which it is not immediately clear whether the modes specified by □a, b, c, d□ and □e□ are all different or perhaps whether some of them are the same. In fact, a close examination reveals that each of them specifies exactly the same mode. Each is merely a different way of thinking about the same kind of data structure. It might be thought that, because the human reader (presumably) has trouble in deciding that the five •mode-indications• are equivalent, it would also be difficult and expensive for the compiler. But this turns out not to be the case[1]. Thus, in large programs, perhaps written by several persons, each person may describe the basic data structure in his own way. If these are indeed the same, then the compiler will quickly find out about it.

------

[1] See the papers of Koster [Ko], Goos [G] and Zosel [Z].

10.7 Binary trees(1)

We shall now consider some procedures for manipulating binary trees. These are data structures of the shape shown in figure 10.7.a. in which each "o" is called a "node" of the tree. At each node there are two branches a "left-" and a "right branch". In more detail, the value of each node is, as is shown in figure 10.7.b, a structured value with at least three fields. The first and last fields are references to the left and right branches, respectively, and the middle field contains some

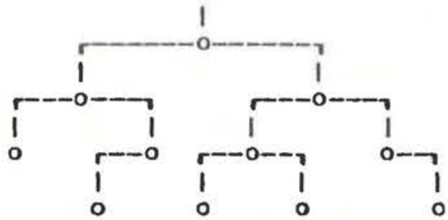


Fig.10.7.a

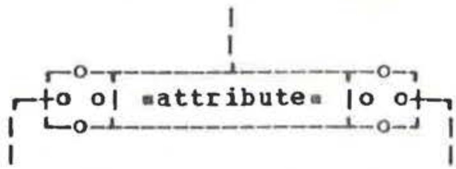


Fig.10.7.b

information, perhaps a string, which is an attribute of that particular node.

The necessary •mode-declaration• would be
□<u>struct</u> <u>node</u> = (<u>ref</u> <u>node</u> left, <u>string</u> val, <u>ref</u> <u>node</u> right)□ .
We may observe that the mode specified by □<u>node</u>□ is infinite, in the sense described in section 10.3 above.

A binary tree is used for many different purposes. For an illustration, we shall use it to store and retrieve character strings in alphabetic order.

10.8 Insertion in a binary tree

Suppose that we are given three strings "jim", "sam" and "bob", in that order, and that we wish to store these in a binary tree such as that discussed above. Storing the first string would result in the structure shown in figure 10.8.a. After the second and third strings have been stored, the

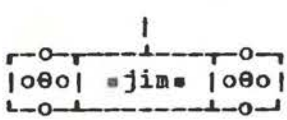


Fig.10.8.a

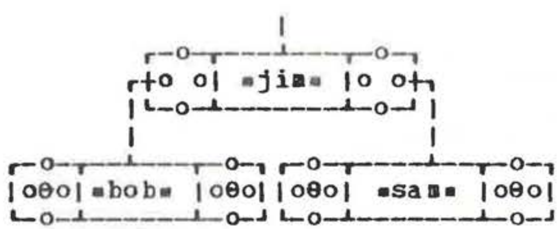


Fig.10.8.b

(1) For an authoritative discussion of binary trees, see Knuth [Kn] Section 2.3.1.

structure is that shown in figure 10.8.b. Note that the shape of the tree will depend upon the order in which the strings are encountered. Whichever string is stored first generates a node which becomes the "root" of the tree. The succeeding strings are then compared with those already present to determine whether to branch to the left or to the right.

A procedure to insert a given string □s□ into a tree whose root is referred to by □root□ is as follows.

```
  □proc insert = (string s, ref ref node root) :
     ( ref ref node n := root ;
       while (ref node : n) :≠: nil do
          n := ( s < val of n | left of n | right of n ) ;
        ( ref ref node : n ) := node := (nil, s, nil)
     )□                                                    .
```

Suppose that we start with an empty tree, i.e., the •declaration•

□ref node tree := nil□

and then elaborate the •call• □insert("jim", tree)□. The

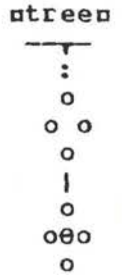


Fig.10.8.c

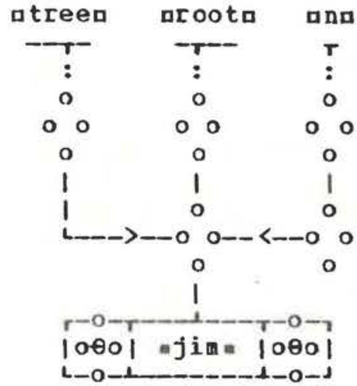


Fig.10.8.d

situation both before and after this •call• is shown in figures 10.8.c and d. Observe that the modes of both the •formal-parameter• □root□ and the •actual-parameter• □tree□ are the same, viz., that specified by □ref ref node□, so that no coercion occurs when the parameter is passed.

The •declaration• □ref ref node n := root□ implies that the mode of □n□ is that specified by □ref ref ref node□. Since □root□ is of mode specified by □ref ref node□, the initializing assignment to □n□ invokes no coercion. In the •assignation•

□(ref ref node : n) := node := (nil, s, nil)□ ,

the second occurrence of □node□ is a •global-generator• generating a name of mode □ref node□, to which is assigned the value of the •structure-display• □(nil, s, nil)□. Because the mode of □n□ is □ref ref ref node□, it must be dereferenced once before the new node is assigned. This is the reason for the •cast• □ref ref node : n□. This •cast• is necessary. In fact, □n := node□ is not an •assignation•, for there is one •reference-to-• too many on the left.

If now we elaborate the •call• □insert("sam", tree)□, we have what is shown in figure 10.8.e. Here we have effectively elaborated the assignation □n := right of n□ in going from figure 10.8.d to 10.8.e. In the •selection• □right of n□, □n□ has the a priori mode □ref ref ref node□, but being in a weak position, it is dereferenced (twice) to □ref node□. The a priori mode of □right of n□ is thus □ref ref node□, since the field

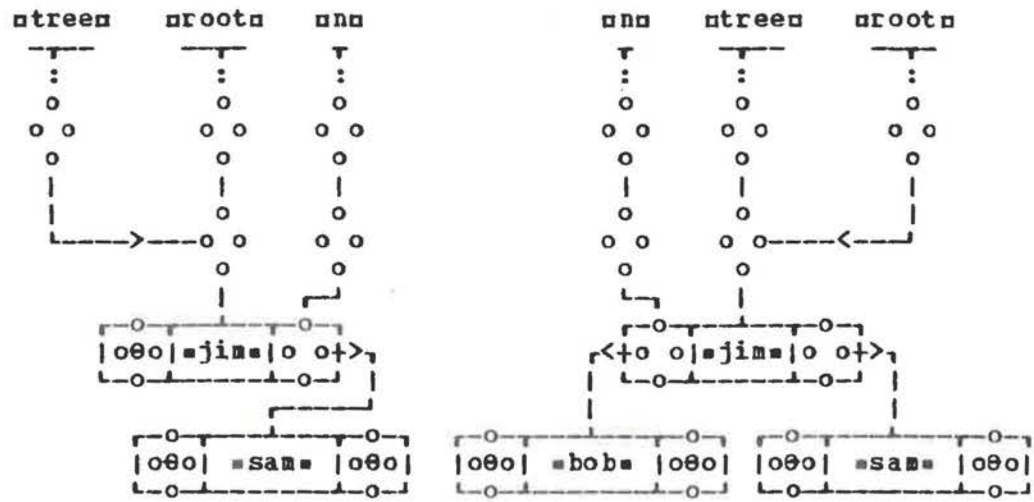


Fig.10.8.e   Fig.10.8.f

selected by □right of n□ is thus a name which refers to a name in a node. Since the mode of □n□ is □ref ref ref node□, the assignment now takes place without further coercion. This moves □n□ down the tree by one node. After elaboration of □insert("bob", tree)□, we would have what is shown in figure 10.8.f.

10.9 Tree searching

Another process in tree manipulation is the searching of a tree for a node which contains a given attribute. In the reach of the •declarations• of section 10.8, and of □ref node m := nil□, this would be accomplished by the following:

```
 □proc search = (string s, ref ref node root) bool :
   ( ref ref node n := root ;
     while (ref node : n) :≠: nil do
       if s = val of n
       then m := n ; go to done
       else n := ( s < val of n | left of n | right of n ) ;
       fi ; false .
     done : true
   )□
```

The value delivered by the •procedure• is ▪true▪ if the node with string □s□ is found; otherwise, it is ▪false▪. As a side effect, the node where the string occurs is assigned to the non-local •variable• □m□; otherwise, □m□ remains referring to ▪nil▪. Using the tree constructed in section 10.8, the result of elaboration of the •call• □search("sam", tree)□ would result in the situation pictured in figure 10.9.

The •variable• □m□ serves to remember where the node was found. In the •assignation• □m := n□, □n□ is dereferenced twice. Note also that in the •formula• □s = val <u>of</u> n□, first □n□ is

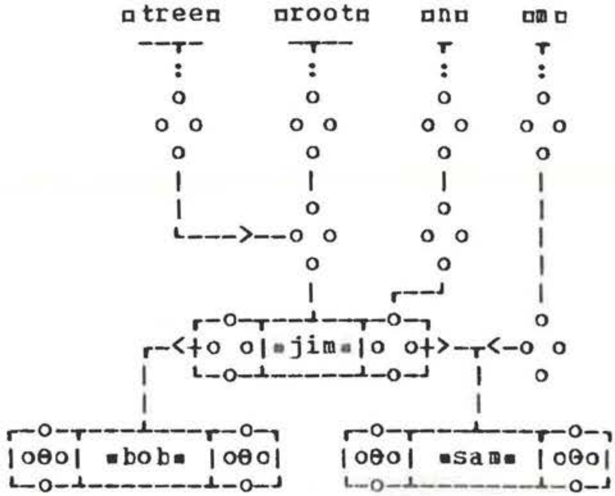


Fig.10.9

dereferenced twice, then □val <u>of</u> n□ is dereferenced once before the comparison of strings is made.

10.10 Searching and inserting

The two processes just described are often combined into one. Thus we may wish to search a binary tree for a given string, to insert it if it is not there, and, in any case, to return with a knowledge of its position. This would be the kind of action necessary if the tree were being used as a symbol table for a compiler. A procedure to accomplish this might be as follows.

```
  proc searchin = (string s, ref ref node root) ref ref node :
     ( ref ref node n := root ;
       while (ref ref node : n) :≠: nil do
         if s = val of root
         then go to done
         else n := ( s < val of n | left of n | right of n )
         fi ;
       (ref ref node : n) := node := (nil, s, nil) ;
       done : n
     )□
```

All the elements of this procedure have been seen already. It is therefore sufficient to remark that the value delivered by the procedure is that of the □n□ which follows the label □done :□, after this □n□ has been dereferenced once.

10.11 Tree walking

Another fundamental manipulation with binary trees is known as a "tree walk". This is a process of visiting each and every node of the tree. Usually some action is to be taken at each node, e.g., printing its string, or counting the node. A tree walk is called a "pre walk", "post walk" or "end walk" (see Knuth [Kn]) depending on whether the action is to be taken upon first reaching the node, or between examining its left and right branches, or upon leaving the node for the last time. For

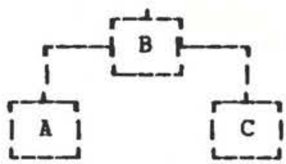


Fig.10.11

example, for the tree displayed in figure 10.11, a pre walk would perform action on the nodes in the order B A C, a post walk in the order A B C and an end walk in the order A C B.

We shall now write a procedure for printing the strings of the nodes, in alphabetic order, by doing a post walk over a binary tree. This is a typical problem in which recursion provides a neat solution, which is as follows: if the tree is empty, then do nothing; otherwise, using an induction hypothesis that we know how to walk a tree with the number of nodes less one, first walk the left branch, then print the string, then walk the right branch. The procedure is as follows.

```
□¢1¢          proc post walk = (ref node root) :
 ¢2¢            (root :≠: nil
 ¢3¢            | post walk(left of root) ;
 ¢4¢              print(val of root) ;
 ¢5¢              post walk(right of root)
 ¢6¢            )□
```

In lines 3 and 5, the •actual-parameters• □left of root□ and □right of root□ are dereferenced once. Note that an end walk is similar - merely interchange lines 4 and 5 (except for □;□). For the pre walk we interchange lines 3 and 4 (except for the □|□). For the tree discussed in section 10.8, the •call• □post walk(tree)□ should print its strings in alphabetic order. Note that the •actual-parameter• □tree□ is dereferenced once.

We may now make this procedure more useful by generalizing it to perform a given action at each node. The action is in the form of a •procedure• which is passed as a parameter.

```
  □proc post walk a = (ref node root, proc(ref node) action) :
   begin proc q = (ref node r) :
     ( r :≠: nil
     | q(left of r) ; action(r) ; q(right of r)) ;
   q(root)
   end□
```

10.12 A non recursive approach

The recursive solution to the tree walk problem, given in section 10.11 above, is simple to program and easy to understand. When proving the correctness of programs, this is an important consideration. However, by using recursion, a certain price must be paid for this convenience, because the run-time organization may need to build a stack to remember the nested •calls• and this stack will require storage the size of which is unknown. In certain situations the programmer may not wish to pay this price. For example, he may be writing a garbage collection routine which must work well just when the amount of free storage is at a minimum. For this reason other schemes of walking trees are exploited [SW]. We shall outline such a scheme here.

The basic principle is that the tree is broken apart at one node, some of the names are reversed and three variables are used to keep track of where the break occurs. As we move the break down the tree, the names are reversed to refer to where we came from. As we move up the tree, the names are restored to their former state. Also, when we move from the left branch to the right branch of a node, it is necessary to shift the reversed name from the left to the right. The extra storage required consists of three variables □p, q□ and □r□ of mode specified by □*ref* *ref* *node*□, and the existence of a boolean field in each node (or corresponding to each node) which remembers whether we have already moved across that node (i.e., whether the name which refers upward is on the right). The value of this field is initially •false•.

The •mode-declaration• given above is thus amended slightly to

□*struct* *node* =
(*ref* *node* left, *string* val, *bool* flag, *ref* *node* right)□ .

The situation at some moment in moving down the tree is

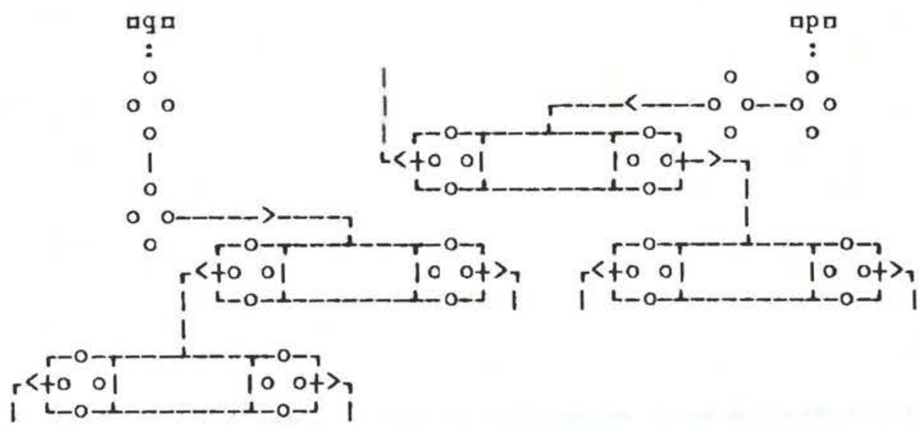


Fig.10.12.a

pictured in figure 10.12.a.

The steps in the process of moving down are

```
               □(      r := left of q ;
               left of q := p ;
                       p := q ;
                       q := r )□
```

after which the situation is as shown in figure 10.12.b. We need

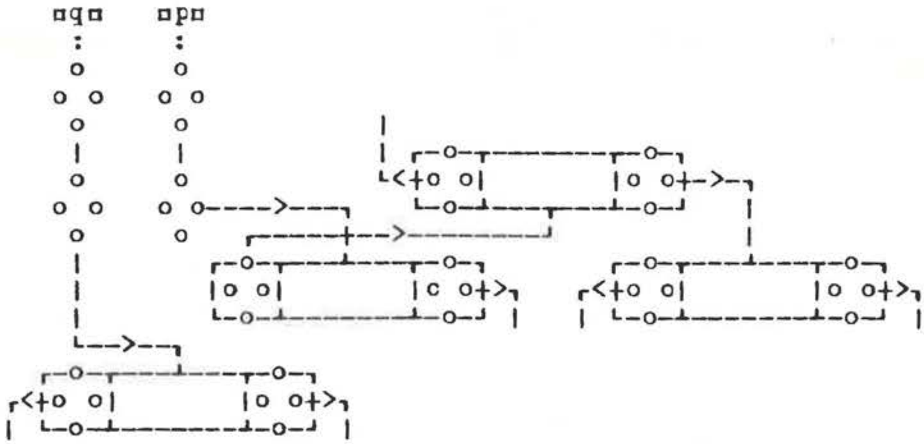


Fig.10.12.b

only add some way to stop this process. This is accomplished by the •condition•

□(ref node : q) :≠: nil□ .

One should also check that the process starts from the □root□ correctly and works properly when □(ref node : q) :=: nil□.

When the walk on the left branch is done we must move across the node. The situation before is as in figure 10.12.c

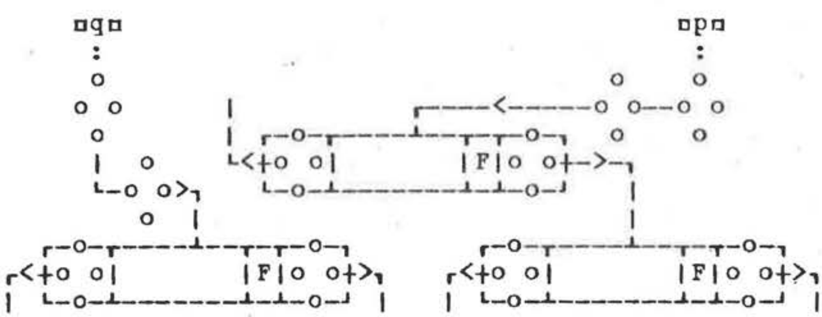


Fig.10.12.c

and the steps in the process are

```
                    □r := q ;
                     q := right of p ;
              right of p := left of p ;
               left of p := r□
```

.

The situation after elaboration of these statements is as in figure 10.12.d. Now we perform the action at this node and then remember that we have done so by

¤action(p) ;
tag of p := true¤ .

The process of moving up the tree is the opposite of moving down the tree except that we must check whether we are done,

¤(ref node : q) :=: root¤ ,

and whether we should change to moving across

¤¬ tag of p¤ .

Also, as we move up, the value of the flag field is restored to ▪false▪.

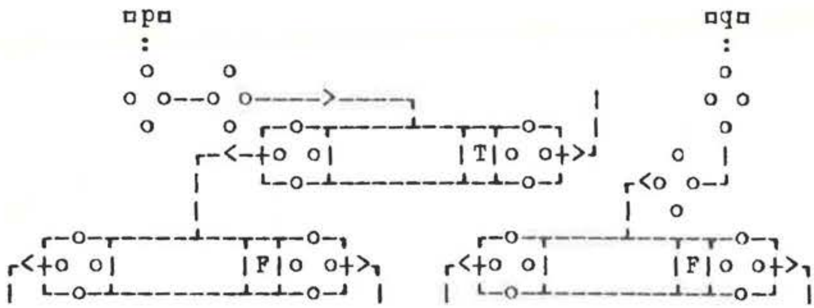


Fig.10.12.d

The complete algorithm is expressed as follows:

```
¤proc walk = (ref node root, proc(ref node) action) :
  begin ref node p := root, q := root, r ;
    if root :≠: nil
    then
    down : while (ref node : q) :≠: nil do
      (¢see figure 10.12.a¢
      r := left of q ; left of q := p ; p := q ;
      q := r ¢see figure 10.12.b¢ ) ;
    across : ¢see figure 10.12.c¢
    r := q ; q := right of p ; right of p := left of p ;
    left of p := r ; ¢see figure 10.12.d¢
    tag of p := true ; action(p) ;
      if (ref node : q) :≠: nil then down fi ;
    up : while (ref node : q) :≠: root do
      if tag of p
      then tag of p := false ; r := right of p ;
      right of p := q ; q := p ; p := r
      else across
      fi
    fi
  end ¢walk¢                                          .
```

Review questions

10.1 Syntax

a) Is □mode real = long int□ a •mode-declaration•?
b) Is □mode a = [1:n]real□ a •mode-declaration•?
c) Is □mode r = [ ]real□ a •mode-declaration•?
d) Is □union a = (b)□ a •mode-declaration•?
e) Is □struct u = (int q, real s)□ a •mode-declaration•?

10.2 Development

a) In the reach of □mode a = ref b ; mode b = [1:n] int, d = proc b□, develop the •declarer• □struct (a a, d d)□.
b) What is printed by □begin mode a = [1:2] int ; ref a v ; print(upb v) end□?
c) Develop the •declarer• □form□ in 11.11.t of the Report.
d) Develop the •declarer• □triple□ in 11.11.k or the Report.
e) Develop the •declarer• □book□ in 11.12.w of the Report.

10.3 Infinite modes

a) What are the two occurrences of □link□ on line 4 in section 10.3?
b) What are the three occurrences of □link□ on line 6 of section 10.3?
c) Is the mode specified by □a□, in the reach of □mode a = ref b, b = struct(a a)□, an infinite mode?
d) Build the list structure shown in figure 10.3 from top down.
e) Is □link a := (1, (2, (3, nil)))□ a •declaration•?

10.4 Shielding and showing

a) Is □m□ shielded in □[1:n]struct(m a, int b)□?
b) Is □m□ shown in □struct(ref a a, b b)□, in the reach of □mode a = [1:10]m, b = proc m□?
c) Can □mode m = ref proc m□ be contained in a proper •program•?
d) Can □mode m1 = ref m2, m2 = struct(m1 f)□ be contained in a proper •program•?
e) Can □mode m1 = union(m2, m3), m2= struct(ref m1 a, [1:n]m3 b), m3 = proc(m1)□ be contained in a proper •program•?

10.5 Identification

a) Is □(b : u) a v□ a •formula• or a •declaration•?

10.6 Equivalence of mode indications

a) In the reach of □mode a = [1:10] char□, are the modes specified by □a□ and □string□ equivalent?
b) Are the modes specified by □a□ and □b□, in the reach of □mode a = struct(ref a x), b = ref struct(b x)□, equivalent?
c) Simplify the •mode-declaration• □struct a = (int u, ref struct(int u, ref a v) v)□.
d) In the reach of □struct a = (ref b r, bool s), b = (bool s,

ref a r)¤, are the modes specified by ¤a¤ and ¤b¤ equivalent?

e) In the reach of ¤struct k = (ref l a, int b), l = (ref m a, int b), m = (ref k a, int b)¤, are the modes specified by ¤k, l¤ and ¤m¤ equivalent?

10.7 Binary trees

a) In the reach of ¤mode nood = ref struct(nood l, string val, nood r)¤, does ¤nood¤ specify an infinite mode?
b) Using at most three statements, in the reach of the •mode-declaration• for ¤node¤ of 10.7, construct the binary tree of figure 10.8.b.

10.8 Insertion in a binary tree

a) Write, as one •assignation•, the equivalent of ¤insert("ron", tree)¤, for the situation in figure 10.8.f.
b) For the tree as shown in figure 10.8.f, what is printed by ¤print(val of left of tree)¤?
c) For figure 10.8.f, what is the value of ¤(ref node : root) :=: n¤?
d) For figure 10.8.f, what is the value of ¤left of tree :=: n¤?
e) For figure 10.8.f, what is the value of ¤left of n :=: nil¤ and that of ¤left of n :=: (ref node : nil)¤?

10.9 Tree searching

a) Rewrite the •declaration• of ¤search¤ without using a •completer•.

10.11 Tree walking

a) Define a •procedure• ¤p1¤ such that ¤p1(tree)¤ will print the strings of a tree (see figure 10.11) in the form ((()A())B(()C())).
b) Define a •procedure• ¤p2¤ such that ¤p2(tree)¤ will print the strings of a tree (see figure 10.11) in the form (A,B,C).

10.12 A non recursive approach

a) Alter the algorithm of 10.12 from a post walk to a pre walk.

11 Easy transput

11.1 General remarks

The transput routines of ALGOL 68 are written in ALGOL 68 itself [R.10.5]. This means, in theory, that it is not necessary to explain any of them here. In order to understand what the transput routines do, we need only to act like a computer and to elaborate the routines of the Report. However, most of us prefer not to emulate a computer. For this reason, extensive pragmatic remarks are included in section 10.5 of the Report and some informal remarks on the simple routines, which would be used by a beginner, are appropriately the subject of this chapter.

The general philosophy is that no new language tricks are used. This means that what we have already learned about the language should be sufficient for the understanding of the transput routines. The transput does not depend upon exceptions or special cases.

11.2 Print and read

The two most useful routines for the beginner are

¤print¤

and

¤read¤ .

We have met them before in several examples in preceding chapters. The procedure ¤print¤ is used for unformatted output to the standard output file (probably a line printer) and the procedure ¤read¤ is used for unformatted input from the standard input file (probably a card reader). Examples of their use are

¤print(x)¤
¤print(("answer.=.", i))¤
¤print((new page, title))¤

and

¤read(x)¤
¤read((i, j))¤
¤read((x1, new line, y1))¤
¤read((a, space, b, space, c))¤ .

An important point to notice is that both ¤print¤ and ¤read¤ accept only one •actual-parameter•. Thus ¤read(x, y)¤ is incorrect. The mode of the •parameter• of ¤print¤ and ¤read¤ begins with •row-of-•. This means that ¤read((i, j))¤ or ¤print((i, j))¤ is acceptable since ¤(i, j)¤ is a •row-display•. Note that ¤print((x))¤ is as good as ¤print(x)¤, for ¤(x)¤ is a •closed-clause• whose value is ¤x¤ and ¤x¤ will be rowed to a multiple value, a row with one element.

Observe that, in addition to •variables• like ¤x¤ (and for ¤print¤, •constants• like ¤"answer.=."¤), the •units• of the •row-display• (or the single •parameter•) may be certain layout procedures like ¤space, backspace, new line¤ or ¤new page¤, to allow for a rudimentary control over the standard input and output files. Thus ¤print((new page, "page.10", new line, "name", space, "address"))¤, should result in the following output at the top of a new page.

PAGE 10
NAME ADDRESS

11.3 Transput types

In order to understand what values can be printed and read, we should examine the •mode-declarations• for the hidden •indicants• □outtype□ and □intype□ [R.10.5.0.1.b,e]. We call these "hidden" because, although they appear in the Report in the form □% outtype□ and □% intype□, they may not be used directly by the programmer. They are present only for the purpose of description of the transput routines. If one is used by a programmer, then it will be regarded as an •indicant• with no defining occurrence.

The declaration of □outtype□ may be paraphrased as follows: □outtype□ specifies a union of the modes □int, real, bool□ and □char□, together with prefixed □long□s where applicable, and all multiple and/or structured modes built from these. Examples are □[ ]int, string, compl□ and □[ ]struct(int n, [ ]bool b1)□. Note that values of each of these modes are constants.

If we consider a union of the same modes as for □outtype□, but each preceded by •reference to•, then we have the mode specified by □intype□. Examples are □ref int, ref char, ref[ ]int, ref string, ref compl□ and □ref[ ]struct(int n, [ ]bool b1)□.

Thus, □outtype□ is an appropriate union of those constants which we might expect to print and □intype□ is a union of the corresponding •variables•.

It is now perhaps convenient, for our discussion, to suppose that there is a •mode-declaration•

```
□mode printtype = union(outtype, proc(file)),
      readtype = union(intype, proc(file))□      ,
```

although such a •mode-declaration• does not exist in the •standard-prelude•. With this in mind, we may now say that the •parameter• of □print□ is of the mode specified by □[ ]printtype□ and that of □read□ is that specified by □[ ]readtype□. This means, in particular, that the □x□ in □print(x)□ will be subjected syntactically to the coercion of dereferencing to □real□, uniting to □printtype□ and then rowing to □[ ]printtype□, whereas in □print((x, y))□, the last coercion is not necessary since □(x, y)□ is already of mode •row of•. In □print(new page)□, the □new page□ is of a priori mode □proc(file)□ and it is united to □printtype□ and rowed to □[ ]printtype□. These particular coercions are of little concern to the programmer except perhaps that their understanding helps to prevent such errors as □print(x, y)□.

11.4 Standard output format

We shall now examine what to expect of the appearance of •constants• on the standard output file □stand out□ as a result of a •call• of □print□. For this purpose, the mode specified by the hidden •indicant• □simplout□ [R.10.5.0.1.a] is relevant to our explanation. It is a union of the modes specified by □int, real, compl, bool, char□ and □string□ together with prefixed □long□s, if applicable. We shall be able to understand the output appearance then, if we consider the action of □print□ on values of each of these modes in turn.

We shall also need some assumptions about the environment, if we are to give illustrative examples. Therefore let us assume that, in our environment, □int width□ [R.10.5.1.3.h] is ▪5▪, □real width□ [R.10.5.1.3.i] is ▪7▪, □exp width□ [R.10.5.1.3.j] is ▪2▪ and □max char[stand out channel]□ (the line length) [R.10.5.1.1.m, 10.5.1.3.e] is ▪64▪ (the same as this text).

With these assumptions then, the result of the •call•

□print((newline, true, false, 1, 0, -1, 1.2,
0.0, -.0034, "a", "abc", 1i2))□

is

```
1 0     +1     +0     -1 +1.200000E +0 +0.000000E +0
-3.400000E -3 A ABC +1.000000E +0 I +2.000000E +0              .
```

The value -3.400000E -3 was printed on a new line because there was not enough room on the first line. Note that an integral value occupies 6 (□int width + 1□) print positions, a real constant 13 (□real width + exp width + 4□) print positions, a complex value 28 and a boolean or a character value 1 each. Also each of these is separated from the previous one by a space, unless we are at the beginning of a line.

Multiple values are also included in the united mode specified by □outtype□ and therefore multiple values may be printed. For example, in the reach of [1:3]int u1 = (1, 2, 3)□, the result of □print((u1, 4))□ is

```
    +1     +2     +3     +4                                        .
```

Also, in the reach of □[1:2, 1:2]int n2 = ((5, 6), (7, 8))□, the result of □print(n2)□ is

```
    +5     +6     +7     +8                                        .
```

Actually, the description of □print□ [R.10.5.2.1.a,b] indicates that each of the •units• of a •row-display• □(a, b, c, d)□ in □print((a, b, c, d))□ is first "straightened" (unravelled) [R.10.5.0.2.c] to a value of mode specified by □[ ]simplout□ and each of the elements of each of these straightened rows is then printed with the standard format discussed above. This means, for example, that the □n2□ in □print(n2)□, given above, is, within the •procedure• □print□, straightened from □outtype□ to □[ ]simplout□ [R.10.5.2.1.b, 10.5.0.2.a]. Thus, all multiple values and all structures (except for □compl□ and □string□, which are already in □simplout□) are straightened to □[ ]simplout□ before printing.

The exceptions for □string□ and □compl□ are that, although □string□ has the mode •row of character•, the result of □print("abcd")□ is ABCD and not A B C D, which would be the case if it were treated like other multiple values, and □print(1.2 i 3.4)□ gives

+1.200000E +0 I+3.400000E +0

rather than

+1.200000E +0 +3.400000E +0

which would be the case if it were treated in the same way as the other structured values.

One final point is that the appearance of the result of ¤print(x) ; print(y)¤ is exactly the same as that of ¤print((x, y))¤. In particular, each •call• of ¤print¤ does not start the output on a new line. A new line is started only when there is not enough room on the old line or when one of the layout procedures ¤new line¤ or ¤new page¤ is called.

11.5 Conversion to strings

For those who find that this standard format does not meet their needs, there are a few •procedures• which allow for some form of simple control over the appearance of the output, without resorting to the use of formats. These procedures convert integer or real values and their long variants to strings. They are ¤int string, real string, dec string¤ and the same preceded by ¤long¤s, if applicable [R.10.5.1.3.c,d,e]. Thus, if it is desired to print the integral value ▪25▪ using a width of three print positions, this can be done by

¤print(int string(25, 3, 10))¤ .

The second •parameter• of ¤int string¤ is the string length and the third is the radix. The •call•

¤print(int string(25, 3, 8))¤

would yield +31, because 25 = 3 * 8 + 1. For real values the value of ¤real string(3.14, 10, 3, 2)¤ is ▪+3.140E+00▪ and the value of ¤dec string(3.14, 10, 3)¤ is ▪+00003.140▪. In both •procedures•, the second •parameter• is the length, the third is the number of digits to the right of the point, and for ¤real string¤, the fourth •parameter• is the length of the exponent.

Notice that the value of ¤int string(25, 8, 10)¤ is ▪+0000025▪, so that those who require zero suppression must either accept what they get from ¤print(x)¤ or use formatted output. Another possibility is to do the zero suppresion oneself by defining a •procedure• like the hidden •procedure• ¤% sign supp zero¤ [R.10.5.2.1.q].

11.6 Standard input

The philosophy for unformatted input is that any reasonable representation of the value to be read is acceptable, that it may appear anywhere on the line and may be of any width. What is expected for each value depends upon the mode of the •variable• to which it is to be assigned. Remember that the mode of the •parameter• of ¤read¤ is ¤[ ]<u>readtype</u>¤, where ¤<u>readtype</u>¤ is ¤<u>union</u>(<u>inttype</u>, <u>proc</u>(<u>file</u>))¤. Thus, in ¤read((a, b, c))¤, the ¤a¤ is either a layout •procedure•, like ¤new line¤, or a •variable• (or perhaps a •clause• which delivers a name of the appropriate mode).

The modes we need to consider are those in the union specified by ¤<u>simplout</u>¤, each preceded by •reference to•, i.e., ¤<u>ref</u> <u>int</u>, <u>ref</u> <u>real</u>, <u>ref</u> <u>compl</u>, <u>ref</u> <u>bool</u>, <u>ref</u> <u>char</u>, <u>ref</u> <u>string</u>¤

and their long versions like □ref long real□ and so on. For convenience let us suppose that this union is specified by □simplin□. We shall need to consider each of these modes in turn.

In the reach of □int i, long int li□, the •call• □read((i, li))□ would be satisfied by two •integral-denotations• like

```
          3              -2
```

or

```
        +  304          -       0000005              .
```

The •procedure• □read□ looks for the first non blank character from the current position on the input file and interprets what it finds as a value of the required mode. It allows for the possibility that, in the case just cited, there will be two •integral-denotations• with zero or more blanks between the sign and the first digit, if a sign appears at all, but that no blanks may appear between the digits. Note that the same set of characters may be presented for □int□ as for □long int□ (a •long-symbol• is not used).

In the reach of □real x, long real lx□, the •call• □read((lx, x))□ would be satisfied by

```
          2               - 3.45
```

or by

```
    6.789           e + 2    .00003
```

or by

```
   123-4.56                                          .
```

Note that the values on the input file need not necessarily be separated by blanks or commas, although most people would naturally do this.

In the reach of □compl z, bool b□, the •call• □read((z, b))□ would be satisfied by

```
   3.456  e  -3  i  + 7.69  1
```

or by

```
 .000345i60                                          .
```

Observe that although □read□ will widen from □int□ to □real□, when necessary, there is here no widening from □int□ or □real□ to □compl□. If the •variable• to be assigned to is of mode □ref compl□, then it expects two values acceptable as □real□ and separated by a •plus-i-times-symbol•.

In the reach of □char c□, □read(c)□ merely reads the next character from the input file and assigns it to □c□ even if that character is a blank. In the reach of □[1:10]char c1□, □read(c1)□ will read exactly 10 characters, including blanks, and assign these to □c1□. If however, we have □[1:3 flex]char cf1□, then □read(cf1)□ reads characters until it finds the end of line or one of the characters which belongs to the string □term of stand in□ [R.10.5.1.mm], whereupon the preceding characters are taken to be those to be assigned to □cf1□. Whichever bound is flexible is then adjusted suitably. If both of them are flexible, e.g., in the reach of □[0 flex: 0 flex]char silly□, the •call• □read(silly)□ will result in a lower bound of ▪1▪ for □silly□. The programmer may specify the terminators as for example in □term of stand in := "?!"□, which changes the set of terminators to "?" or "!".

For multiple and structured •variables• in the union □intype□, the first step is to straighten to □[ ]simplin□, where □simplin□ is the union of modes discussed above. Thus, in the reach of □[1:3, 1:2]real x2, struct(int a, bool b) c□, the •call• □read((x2,c))□ would be satisfied by

```
   3 .1   .6   4  .2    .7         5 0                          .
```

11.7 String to numeric conversion

The •procedure• □read□ must of necessity convert character strings to integral or real values, and in doing so it makes use of three standard •procedures•, □string int, string dec□ and □string real□ [R.10.5.2.2.c,d,e]. These •procedures• are not hidden. The programmer may use them himself. The first •procedure•, □string int□, converts a given string to an integral value. It assumes that the first character of the string is a sign. Any character which is not a (hexadecimal) digit, e.g., a space, is treated as a 0. Thus the value of □string int("+␣␣23", 10)□ is ▪23▪ (the second parameter is the radix). The •procedure• □string dec□ converts a •variable-point-numeral•, e.g., □"+2.3450"□, to a real value and □string real□ converts a •floating-point-numeral•, e.g., □"+2.345e-2"□ to a real value. The value of □string dec("+2.345")□ is ▪2.345▪ and that of □string real("+2.3450e-1")□ is ▪.2345▪. These •procedures•, although available, are not likely to be useful for input since □read□ itself has all the flexibility needed. However, thay may well be used for internal manipulation of strings.

Another •procedure• which may be mentioned here is □char in string□ [R.10.5.1.2.n]. It has three •parameters•; the first is of mode •character•, the second of mode •reference to integral• and the third of mode •row of character•. The •procedure• delivers a boolean value which is ▪true▪ if the character, which is the first •parameter•, is found in the string, which is the third •parameter•, in which case its position is assigned to the •integer-variable•; otherwise, the value delivered is ▪false▪ and no assignment is made. The result of ▪char in string("+", i, "x␣+␣y")□ is therefore ▪true▪ and the value ▪3▪ is assigned to □i□.

11.8 Simple file enquiries

For any file, it is possible to make simple enquiries concerning the current position in the file. There are three •procedures•, □char number, line number□ and □page number□ [R.10.5.1.2.v,w,x], each yielding an integral value, the three coordinates of the □book□. In the case of the standard input file, the •calls• □char number(stand in), line number(stand in)□ and □page number(stand in)□ should each yield the value ▪1▪ after the •call• □read((c, back space))□, if this is the first call of □read□ and is in the reach of □char c□. Notice that these •procedures• deliver integral values and not names, so that they are for enquiry only and cannot be used to alter the position in the file.

There are also three •procedures• ▫line ended, page ended▫ and ▫file ended▫ [R.10.5.1.2.h,i,j], each of which delivers an appropriate boolean value, but a careful distinction must be made between ▫file ended▫, which tests whether the maximum capacity has been exceeded, and ▫logical file ended▫ [R.10.5.1.2.k], which tests whether the usable information in the file has been exhausted. In the case of the file ▫stand in▫, if it is a card reader, then ▫file ended(stand in)▫ is likely always to be •false•, but ▫logical file ended(stand in)▫ may become •true• each time we reach the end of the data for a particular job. The •call• ▫logical file ended(stand out)▫ will always yield •false•, because ▫get possible[stand out channel]▫ [R.10.5.1.1.j, 10.5.1.3.b] is likely to be •false•, i.e., ▫stand out▫ is not an input file. But ▫file ended(stand out)▫ may well become true when the page limit for a particular job is reached, or when the box of paper is exhausted.

11.9 Other files

It is worthwhile noticing now that ▫print(x)▫ is the same as ▫put(stand out, x)▫ and ▫read(x)▫ is the same as ▫get(stand in, x)▫; in fact, this is the way that ▫print▫ and ▫read▫ are defined [R.10.5.2.1.a, 10.5.2.2.a]. This means that if another file is available, say in the reach of the •declaration• ▫<u>file</u> f▫, then what we have said about unformatted transput on the standard files applies also to the file ▫f▫ by using, e.g., ▫put(f, x)▫ and ▫get(f, x)▫. Such files must be opened (and closed) by the programmer, but this is the subject matter of another chapter.

Another standard file which is always available, i.e., is opened automatically, is ▫stand back▫. This file may be used for saving intermediate results during the elaboration of a •program•. When the elaboration is completed, this information will be lost, since the file is locked [R.10.5.1.ii, 10.5.1.2.t] by the •standard-postlude•. The two relevant •procedures• here are ▫write bin▫ and ▫read bin▫. The mode of the •parameter• of ▫write bin▫ is ▫[ ]<u>outtype</u>▫, and that of ▫read bin▫ is ▫[ ]<u>intype</u>▫. For example, in the reach of ▫[1:n]<u>real</u> x1▫, if we want temporarily to save the values of a rather large array, this could be accomplished by the •call• ▫write bin(x1)▫. The array can then be recalled by ▫read bin(x1)▫. If another file, say ▫f▫, is available, the same could be done by ▫put bin(f, x1)▫ and ▫get bin(f, x1)▫, and if the file ▫f▫ is not locked then these two •calls• might appear in different •programs•.

Review questions

11.2 Print and read

a) Is ▫print(new page, new line)▫ a •call•?

b) Is ¤print(nil)¤ a •call•?
c) What is the result of ¤print(get possible[stand in channel])¤?
d) In the reach of ¤ref real xx := loc real := 3.14¤, what is the result of ¤print(xx)¤?
e) In the reach of ¤ref real xx := loc real := 3.14¤, what is the result of ¤print(ref real : xx)¤?

11.3 Transput types

a) What is the result of ¤print(for i by 2 to 10 do 3)¤?
b) Can ¤nil¤ be coerced to ¤[ ]printtype¤?
c) In the reach of ¤ref real xx¤, can ¤xx¤ be coerced to ¤[ ]readtype¤?
d) In the reach of ¤struct(ref c next, int n) s := (nil, 2)¤, what is the result of ¤print(s)¤?
e) In the reach of ¤format f¤, is ¤read(f)¤ a •call•?

11.4 Standard output format

In the following, assume the same environment as given in section 11.4.

a) What is the result of ¤print(("?", int width))¤?
b) What is the result of ¤print(("?", space, "abc"))?
c) In the reach of ¤ref real xx := loc real := 3.14¤, what coercions occur to ¤xx¤ in ¤print(("?", xx))¤ and what is printed?
d) How many real values can be printed on a line?
e) How many integral values can be printed on a line?
f) Is the result of ¤print(("a", "b", "c"))¤ ABC or A B C?

References

[B] W.Brown, The cross-referencing of a van Wijngaarden grammar, University of Calgary, 1969.

[G] G.Goos, Some problems in compiling ALGOL 68, ALGOL 68 Implementation, North-Holland, 1971, pp. 179-196.

[H] J.E.Hopcroft and J.D.Ullman, Formal Languages and their Relation to Automata, Addison Wesley, 1969.

[Kn] D.E.Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison Wesley, 1968.

[Ko] C.H.A.Koster, On infinite modes, Algol Bulletin, No. 30, Feb. 1969, pp. 86-89 (AB.30.3.3).

[M] L.Meertens, On the generation of ALGOL 68 programs involving infinite modes, ALGOL Bulletin, No 30, Feb. 1969, pp. 90-92 (AB30.3.4).

[N] P.Naur, Revised Report on the Algorithmic Language ALGOL 60, Comm. Assoc. Computing Machinery, 6(1963) pp. 1-77.

[Pa] C.Pair, Concerning the syntax of ALGOL 68, Algol Bulletin, AB 31.3.2, March 1970.

[P] PL/I Language Reference Manual, IBM Form C28-8201-2.

[R] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the Algorithmic Language ALGOL 68, Numerische Mathematik, 14(1969) pp. 79-218.

[SW] H.Schorr and W.M.Waite, An efficient machine independent procedure for garbage collection in various list structures, Comm. Assoc. Computing Machinery, Vol. 10 (1967), pp. 501-506.

[W] H. Wössner, On identification of operators in ALGOL 68, ALGOL 68 Implementation, North Holland, 1971, pp. 111-118.

[Z] Mary Zosel, Mode classification, Univ. of Washington, 1970.

Answers to review questions

1.1 a) It ends with •symbol•. b) Three, •label-symbol•, •cast-of-symbol• and •up-to-symbol•, unless one observes that the •label-symbol• is in italic, and the other two in normal type. c) Yes, e.g., ¤.¤, which represents a •point-symbol• and a •completion-symbol•. d) It is a representation of the •open-symbol•, but, by extension 9.2.g, it may be used in place of ¤[¤.

1.2 a) An internal object which is a real value. b) A •real-denotation• (amongst other things). c) It is an external object. d) ¤true¤ possesses ▪true▪.

1.3 a) No. b) Yes. c) No, it is an internal object. d) No, i.e., not at the same time, but in the course of time - yes. e) No.

1.4 a) No. b) Yes, a •collateral-declaration• [R.6.2.1.a].

1.5 a) There are four classes: integral values, real values, truth values and characters. b) Yes, the truth values. c) The mode.

1.6 a) The mark ":" is read as "may be a", ";" as "or a" and "," as "followed by a". b) Yes.

1.7 a) Yes, e.g., ¤123¤ and ¤000123¤. b) No, but it is a •formula•. c) Yes. d) No, not if this value would exceed ¤max int¤ [R.10.1.b].

1.8 a) Yes, e.g., possibly ¤2.34¤ and ¤23.4e-1¤. b) No. Oh, please no. c) No. d) Yes. e) No, but it is a •formula• [R.8.4].

1.9 a) No. b) Yes.

1.10 a) Infinitely many. b) As many as he likes, but always a finite number.

1.11 a) No, it is a •character-denotation•. b) Yes. c) •row of character•.

1.12 a) No [R.2.2.3.1.b]. b) •structured with row of boolean field letter aleph•. c) •format•.

1.13 a) •row of character•. b) •reference to real•, •reference to integral• c) No. d) Six. e) No.

2.1 a) No. b) Yes. c) ¤ref ref [ ]char¤. d) Yes. e) Yes. f) No. g) No, except for ▪nil▪. h) No, a •declarer• specifies a mode.

2.3 a) None. b) □loc char□. c) □loc bool□. d) No. e) No. f) No.

2.4 a) No, but it possesses a name referring to a real value. b) Yes. c) No. d) No. e) No. f) No, i.e., not at the same time, but in the course of time - yes.

2.5 a) Yes, but not the same instance [R.2.2.1]. b) No. c) No, but the value referred to by the name possessed by □x□ may be changed. d) □loc[1:3]proc real□.

2.6 a) No. b) Yes, in the extended language. c) •reference-to-reference-to-integral•. d) □[1:3]proc real p□.

2.7 a) Yes. b) Yes. c) No. d) No.

2.8 a) □ref ref real xx = loc ref real□. b) □ref real x = loc real, ref real y = loc real□. c) □ref real x = loc real, ref real y := loc real := 3.14□. d) It is not possible; moreover, if □+□ has its usual meaning, then this is not a •declaration•.

2.9 a) No. b) Yes. c) No. d) Yes, but a rather foolish one.

2.10 a) Yes. b) Yes. c) No. d) □y + 2□. e) •reference-to-reference-to-real•. f) No.

2.12 a) The □y□ is dereferenced and the □3.14□ is not. b) No.

2.13 a) the □n□ is an •integral-mode-identifier• but the □m□ is a •reference-to-integral-mode-identifier; i.e., □n□ is a •constant• and □m□ is a •variable•. c) No.

2.14 a) Four. b) Both □ap□ and □mp□ are dereferenced. c) It is equivalent to □j := j + 1□. d) Yes. □mi□. It's mode is •long-real•. e) •reference-to-long-real•.

3.1 a) No. b) Yes. c) □(a + (b of (c[d]))) - e□. d) An •expression• may possess a value but a statement cannot. e) Yes.

3.2 a) No. b) Five, •mode-identifier, denotation, slice, call• and •void-cast-pack•. c) □a[i], a, i, c, sin(x), sin, x, cos(x + pi/2), cos, x, pi, 2□. d) No. e) It could be either, depending on the mode of □a□ [R.9.2.g].

3.3 a) □l, ca, f□. b) •reference-to-real•. c) •row-of-row-of-integral•. d) Yes. e) No.

3.4 a) Yes. b) Yes, its mode is •reference-to-row-of-real•. c) Yes. d) Yes. e) □35, item of a, i + n * 2, i +:= 2□.

3.5 a) No. b) Yes. c) No. d) Yes. e) Yes.

3.6 a) The same as that of □(2,3)□. b) It possesses the value ■true■ only when □x2[3,1] = x2[2,1]□. c) ■2■. d) ■2■. e)

No, because □i := 1□ is not a •tertiary• and therefore not a •lower-bound•.

3.7 a) Yes. b) No, it is a •deprocedured-coercend• [R.8.2.2.1.a]. c) No, but □cos((x > 0 | x | pi/2))□ is a •call•. d) When the mode of □a□ is •procedure with M1 parameter reference to M2• where •M1• and •M2• are terminal productions of MODE. e) When the mode of □a□ is •procedure-with-M1-parameter-procedure-with-M2-parameter-M3•, i.e., □a□ is a •procedure• with one •parameter• which delivers a •procedure• with one •parameter•, and the modes of □b□ and □c□ are •M1• and •M2• respectively.

3.8 a) Yes. b) No, □(: x)□ has no mode. c) Yes, provided that the mode, after soft coercion, of □x□ is •reference-to-procedure-void•. d) Yes. e) No [R.8.2.3.1], but □proc p := (: x := 3.14)□ is a •declaration•.

3.9 a) No. b) Yes. c) No. d) Yes. e) When the mode of □b□ is structured, has a field selected by □a□ whose mode is •reference-to-M1• where •M1• is the a posteriori mode of □c□, or when □b□ is a •variable• and will refer to structured values that have a field selected by □a□ whose mode is M1.

3.10 a) No. b) No, it is a •field-selector• [R.7.1.1.i]. c) □a of (b[c]), e of(g(x))□. d) No, □(a of b)□ is not a •field-selector•. e) Yes, it could be.

3.11 a) Yes. b) ▪false▪ (if the value of □bits width□ is ▪3▪). c) ▪-4▪. d) No, the left •operand• of the •operator• □+:=□, as declared in the •standard-prelude•, must possess a name. e) ▪false▪.

3.12 a) No. b) No, □i := i + 1□ is not a •tertiary•. c) No. d) No, □proc : (:random)□ is. e) It is an •assignation•.

3.13 a) ▪false▪. b) ▪true▪. c) ▪true▪. d) No, □3.14□ does not possess a name. e) Yes.

3.14 a) No. b) It looks like one, but □3.14□ cannot be strongly coerced to an integral value. c) An •identity-relation•. d) No, because □[1:1]real□ is not a •virtual-declarer•. e) No, □ref int : ii□ is not a •tertiary•.

3.15 a) None. b) Eleven. c) A •constant•. d) •real•. e) None.

4.1 a) The same as that of □3 i 0□. b) No. c) No. d) Yes. e) Yes. f) Yes.

4.2 a) ▪5▪. b) Some undefined integral value. c) ▪11▪. d) □if p then a elsf q thef r then b else c fi□. e) □(a |(b | c |(d | e | skip))| skip)□.

4.3 a) No. b) □if, case□ and □(□. c) ▪4▪. d) ▪2▪. e) No.

4.4 a) No. b) No. c) Yes, □e□ is elaborated infinitely often, or until a jump occurs to a •label-identifier• outside of it. d) Yes, zero times. e) Yes, zero times. f) The second and third occurrences of □i□ identify the first, but □i := 2 * i + 1□ is not an •assignation• since □i□ does not possess a name. g) The last three occurrences of □i□ identify the second occurrence, but the third and fourth occurrences identify the first occurrence.

4.5 a) Yes. b) No. c) Yes. d) No. e) No. f) No.

4.6 a) No. b) No. c) No. d) The same as that of □"abcde"□. e) Yes, e.g., if the order of elaboration happens to be □j +:= i ; i +:= j□.

4.7 a) Yes. b) Yes. c) No. d) Yes. e) □(x or y | 1) ; n := 2 ; s . l: n := 1 ; r□.

4.8 a) Seven. b) •reference-to-row-of-integral•. c) •reference-to-integral•. d) Four. e) None.

5.1 a) No, □real proc□ is not a •declarer•. b) No, □(real a)real□ is not a •virtual-plan• [R.7.1.1.x]. c) □proc real r2 = 2 * random□. d) □proc max = (real a, b) real : (a > b | a | b)□. e) □proc recip = (ref real a) : a := 1 / a□.

5.2 a) No, unless □*□ has been redeclared and possesses an operation which delivers a name. b) □ref[ ]real x1□. c) □(real a = x + 1, real b = y ; a * b)□. d) □(real a = skip ; real : a * a)□. e) ■(int n = skip, int m = skip ; ref[1:n]real a1 = skip ; real : (n < m | a1[n] | a1[m]))■.

5.3 a) The value is voided. b) ■4.6■, in the sense of numerical analysis. c) That of □y□. d) The object □p(x, y)□ is not a call, since □ref ref real a = x□ is not an •identity-declaration•. e) ■2.2■, in the sense of numerical analysis.

5.4 a) □proc p = (int a, proc ref int b) : b *:= 2 * a□, but in most applications □proc p = (int a, ref int b) : b *:= 2 * a□ would be sufficient. Note that since □b□ is passed by name in ALGOL 60, the side effects of □b := b * 2 * a□ occur twice but in □b *:= 2 * a□ they occur only once.

5.6 a) A •constant•. b) Because no assignment is made to □s□. c) Because □g□ is a •constant• and □grow□ requires a •variable• in its last •parameter•. d) It's value is irrelevant for it is used only in the •formula• □t or ¬t□. e) The same as that of □11100000□.

6.1 a) A priori mode, a posteriori mode and syntactic position. b) Strong, firm, weak and soft. c) Yes. d) No. e) Widening.

6.2 a) Eight. b) Dereferencing and widening. c)

Dereferencing and deproceduring. d) Rowing. e) Hipping.

6.3 a) Dereferencing (four times). b) Dereferencing (twice) c) Dereferencing, dereferencing and deproceduring. d) Dereferencing, deproceduring and dereferencing. e) 834a, 71b,c, 61e, 81a,b,c,d, 820d, 822a, 860a, 41b,c, 302b.

6.4 a) Deproceduring and uniting. b) No. c) A routine. d) No. e) No, □random□ is of a priori mode •procedure-real•, it cannot be procedured to •procedure-void• [R.8.2.3.1].

6.5 a) No. b) Hipping. c) Widening of □5□. d) Deproceduring and rowing. e) None, this is not a •cast• since rowing cannot be followed by uniting [R.8.2.4.1.b].

6.6 a) Dereferencing and deproceduring. b) Firm. c) Weak. d) Dereferencing of □rr1x□ twice (not thrice). e) Soft.

6.7 a) •Base, cohesion, formula, confrontation•. b) □b, a **of** b, x, 2, x := 2, x, y, 3, y + 3, x := y + 3□. c) Yes, but its elaboration is undefined since the dereferencing of a •nihil• is undefined [R.8.2.1.2 Step 2]. d) Yes, assuming the •declaration• □**ref** **real** xx□. e) No, hipping cannot occur in a soft position.

6.9 a) 834a, 71b, 421b,c, 61e, 81a,b,c,d, 820d, 825b,a, 821a, 860a, 41b, 302b. b) No, there is no deuniting coercion. c) 74a, 54e, 71b,w,aa,z; 41b, 302b; 74b, 61e, 81a, 820d, 823a, 830a, 834a, 71z; 61e, 81a, 820d, 828a, 830a, 831a,b, 81b,c,d, 820g, 860a, 41b, 302b; 831c, 61e, 81a,b,c,d, 820d, 825a, 860a, 511a, 303c,d. d) 61e, 81a,b,c,d, 820d, 828b, 822a, 860a, 41b,c 302b. e) No, hipping cannot occur in a firm position.

6.10 a) No. b) Yes. c) •real•. d) •real• or •procedure real• or •union of integral and real• or •union of integral and real and boolean• etc. e) No.

6.11 a) No. b) □px□ is softly deprocedured and □xx□ is strongly dereferenced. c) □px□ is softly deprocedured and □**go to** k□ is strongly hipped to •reference-to-real•. d) Yes. e) No.

6.12 a) □x1□ is weakly coerced, □2□ is strongly widened and then rowed to •row-of-real•. b) Yes, strongly-weakly to •real•. c) Yes. d) Yes. e) □random□ is strongly deprocedured and widened and □0 **i** 2□ is weakly coerced.

6.13 a) No. b) No. c) Yes, firmly-strongly. d) Yes. e) No.

6.15 a) Yes. b) Yes, the balanced mode is •reference-to-real•. c) No, it cannot be balanced. d) □4 **i** 5.6□ is firm, the others strong. e) No.

6.16 a) The object □m +:= 1□ is interpreted as □m := m + 1□ so □m□ is dereferenced once, □m +:= 1□ is dereferenced as the left operand of □>□. b) This is equivalent to □**ref** **int** c1 = **loc** **int** := am := **abs** am□. First □am□ is dereferenced to •integral• and the absolute value of this integer is found. It is assigned to □amm□. Then a name is created by □loc int□, the •assignation• □am := abs am□ is dereferenced and the integral value (referred to by □amm□) is assigned to this name. Finally □c1□ is made to possess the name. c) The identifier □ai□ is made to possess the same name as that possessed by □a[i]□. This happens for each repetition of the repetitive statement, in which there are five occurrences of □ai□, thus saving time on subscript calculation. d) This is the position of the statement number 30 in the FORTRAN program. It is redundant in ALGOL 68, but □l30: end□ is not permitted for there is no empty statement. e) ?

7.1 a) Yes, its value is ■false■ [R.7.1.2.c Step 8]. b) Yes, but rather useless. c) ■true■. d) Yes. e) Yes.

7.2 a) No, •integral• mode cannot be united to •union of character and boolean•. b) No, in R.8.2.4.1.a, •strong• goes to firm, so the □1□ cannot be widened. c) Either •real• or •boolean•. d) Yes, and its value is ■false■. e) Yes, provided that it is in the reach of a suitable declaration of the •operator• □+□.

7.3 a) ■true■. b) ■false■. c) ■true■. d) Yes. e) No, □x ::= x□ is not a •tertiary• [R.8.3.2.1.a].

7.4 a) Yes, its value is ■false■. b) Yes, its value is ■true■. c) Yes [R.4.4.3.c,d]. d) No. e) □proc sqirt = (int i)union(int, real) : (real x = sqrt(i) ; int j = round x ; (j * j = i | j | x))□.

7.5 a) ■4■. b) Either ■7■ or ■8■ or ■9■ [R.10.4.2]. c) No, it should be □sema p = /1□. d) Yes, surprisingly, and if the value of □u□ is of •boolean• mode, then the value of the expression is that of □h□. e) No, because a •skip• can only be hipped and must therefore be in a strong position. The right •tertiary• of a •conformity-relation• is of no sort [R.8.3.2.1.a]. f) No, a •jump• can only be hipped (see the answer to e).

8.1 a) No, it is a •confrontation•. b) Yes. c) □(x + (-y)) - ((-(-(abs i))) over 2)□. d) Nine. e) No, it is a •confrontation•. f) ■2■.

8.2 a) No, □:=:□ is not a •dyadic-indication•. It is a •identity-relator•. b) No, the •token• on the right must be > 0. c) No, the token must be < 10. d) Yes, if the implementation permits □?□ as a •dyadic-indicant•. e) No, perhaps the intention was □priority ? = 6, ! = 6□.

8.3 a) No, □:=:□ is not an •operator•. It is an •identity-relator•. b) No, the •actual-parameter• must possess a routine with one or two •parameters•. c) No, □*□ is not a •monadic-operator• [R.3.0.4.a, 4.2.1.f, 4.3.1.c]. Think about □x**2□. d) Yes. e) □op (ref file, int) create = create□.

8.4 a) ■(real a = skip ; int : round a)■. b) No, □random□

possesses a routine which has no •parameters•. c) ▪83▪. d) Yes. e) No, ▫+▫ is not an ▫actual-parameter▫.

8.5 a) One. b) 16 times a sufficient number [R.10.b Step 3, 10.2.3.i,j, 10.2.4.i,j, 10.2.5.a,b, 10.2.6.b, 10.2.7.j,k,p,q,r,s, 10.2.10.j,k,i]. c) 30, [R.10.5.2.2.b, 10.5.3.2.f, 10.2.0]. d) There is none since this is a •monadic-operator•. e) No, it is a •conformity-relator• [R.8.3.2.1.b].

8.6 a) Yes, but it cannot be contained in a proper program. b) Yes, because the second occurrence of ▫abs▫ is that of a •monadic-indication•. and does not identify the first. c) In order to reinstate the •dyadic-indications• and •operators• of the •standard-prelude•. They may have been re-declared. d) Yes [R.6.1.2.a, 6.0.2.d Step 1]. e) Yes [R.6.1.2.a, 6.0.2.d Step 2].

8.7 a) R.10.2.5.a. b) R.11.11.k. c) R.11.11.i d) R.10.2.8.d. e) R.10.2.10.i.

8.8 a) ▫(real a = skip ; bool : a > 0)▫. b) ▫(real a = x ; bool : a > 0)▫.

8.9 a) ▪-1▪. b) No, it is an •identity-relation•. c) No, a •cast• is not an •operand•. d) Yes. e) ▪false▪.

8.10 a) No. b) No. c) Yes, try coercing from ▫int▫ or from ▫proc int▫. d) Yes. e) No, there is a multiple definition of ▫-▫.

8.11 a) It draws a straight line of length ▫d▫ in the direction S. b) Try, ▫n, s, e, w▫. c) !

8.12 a) Remove 2, remove 1. b) Remove 1, remove 3, replace 1, remove 2, remove 1. c) The •formula• requires that ▫a▫ should be a •variable•. d) Remove 2, remove 1, remove 4, replace 1, replace 2, remove 1, remove 3, replace 1, remove 2, remove 1. e) Try ▫proc up▫ and ▫proc down▫.

9.1 a) No. b) Yes. c) No [R.8.3.4.1.a]. d) No. e) Yes [R.5.1.0.1.b].

9.2 a) Infinitely many. b) Six. c) Two. d) Two. e) •virtual, actual• and •formal•.

9.3 a) No [R.3.0.2.b]. b) Three. c) No, it is a metarule. d) Yes. e) No.

9.4 a) No [R.1.2.1.m]. b) No. c) Yes, •row-of-character•, say. d) •real-field-letter-r-letter-e-and• [R.8.5.2.1.a]. e) •real•.

9.5 a) (I) L : x ; y ; z. (II) N : ; Np. (i) s : Nx, yNy, NNz. (ii) NpL : NL, L. b) (I) L : x ; y ; z. (II) N : p ; Np. (i) s : Nx, Ny, Nz. (ii) NpL : NL, L. (iii) pL : . c) (I) L : x ; y ; z. (II) N : ; pN. (i) s : letter x symbol N, letter y symbol N, letter z symbol N. (ii) letter L symbol pN : letter L symbol, letter L symbol N.

9.6 a) No. b) Yes. c) No. d) No. e) Yes, •NONPROC• excludes only •procedure-MOID• or the same preceded by •reference-to• or •row-of•.

9.7 a) •void-cohesion• or •void-confrontation• [R.8.5.0.1]. b) •virtual NONSTOWED declarer•. c) •firmly dereferenced to MODE FORM• d) •strongly rowed to REFETY row of MODE FORM•. e) •STIRM ly united to MOID FORM•.

10.1 a) No, ¤<u>real</u>¤ is not a •mode-indication• [R.4.2.1.b, 1.1.5.b]. b) No, ¤a¤ is an •identifier•, not an •indicant•. c) No, [ ]<u>real</u> is not an •actual-declarer•. d) Perhaps, if ¤b¤ already specifies a united mode¤ [R.7.1.1.cc, 9.2.b]. e) Yes [R.9.2.b].

10.2 a) ¤<u>struct</u>(<u>ref</u> <u>b</u> a, <u>proc</u> <u>b</u> d)¤ b) This is undefined. In ¤<u>ref</u> <u>a</u> v¤ or ¤<u>ref</u> <u>ref</u> <u>a</u> v = <u>loc</u> <u>ref</u> <u>a</u>¤, the •generator• ¤<u>loc</u> <u>ref</u> <u>a</u>¤ contains ¤<u>a</u>¤ which is virtual and is therefore not developed [R.7.1.2.c]. c) ¤<u>union</u>(<u>ref</u> <u>const</u>, <u>ref</u> <u>var</u>, <u>ref</u> <u>triple</u>, <u>ref</u> <u>call</u>)¤. d) ¤<u>struct</u>(<u>union</u>(<u>ref</u> <u>const</u>, <u>ref</u> <u>var</u>, <u>ref</u> <u>triple</u>, <u>ref</u> <u>call</u>) left operand, <u>int</u> operator, <u>union</u>(<u>ref</u> <u>const</u>, <u>ref</u> <u>var</u>, <u>ref</u> <u>triple</u>, <u>ref</u> <u>call</u>) right operand)¤. e) ¤<u>struct</u>([1:0 <u>flex</u>] <u>char</u> title, <u>ref</u> <u>book</u> next)¤.

10.3 a) The first is its defining occurrence as a •mode-indication• and the second is an applied occurrence as a •virtual-declarer•. b) The first is a •declarer• and the other two are •global-generators•. c) Yes. d) ¤<u>link</u> a := (1, <u>nil</u>) ; next <u>of</u> a := <u>link</u> := (2, <u>nil</u>) ; next <u>of</u> next <u>of</u> a := <u>link</u> := (3, <u>nil</u>¤. e) No [R.6.2.1.f].

10.4 a) No. b) Yes. c) No. d) Yes. e) Yes.

10.5 a) If ¤<u>a</u>¤ is a •dyadic-indication•, then it is a •formula• and ¤<u>b</u> : u¤ is a •cast•; if ¤<u>a</u>¤ is a •mode-indication•, then it is a •declaration• and ¤<u>b</u> :u¤ is a •row-of-rower•.

10.6 a) Yes. b) No. c) ¤<u>struct</u> <u>a</u> = (<u>int</u> u, <u>ref</u> <u>a</u> v)¤. d) No. e) Yes.

10.7 a) Yes. b) ¤<u>node</u> tree := <u>node</u> := (<u>nil</u>, "bob", <u>nil</u>), "jim", <u>node</u> := (<u>nil</u>, "sam", <u>nil</u>))¤.

10.8 a) ¤left <u>of</u> right <u>of</u> tree := <u>node</u> := (<u>nil</u>, "ron", <u>nil</u>)¤. b) BOB. c) •false•. d) •true•. e) •false•, •true•.

10.9 a) In line 2, insert ¤<u>bool</u> b := <u>true</u>¤; lines 7 and 8 become ¤<u>fi</u> ; b := <u>false</u> ; done : b¤.

10.11 a) ¤<u>proc</u> p1 = (<u>ref</u> <u>node</u> root) : (print("(") ; (root :≠: <u>nil</u> | p1(left <u>of</u> root) ; print(val <u>of</u> root) ; p1(right <u>of</u> root)) ; print(")"))¤. b) ¤<u>proc</u> p2 = (<u>ref</u> <u>node</u> root) : (root :≠: <u>nil</u> |: left <u>of</u> root :=: (<u>ref</u> <u>node</u> : <u>nil</u>) <u>and</u> right <u>of</u> root :=: (<u>ref</u>

node : nil) | print(val of root) | print("(") ; p2(left of root) ; print(",") ; print(val of root) ; print(",") ; print(right of root) ; print(")"))¤.

10.12 a) Remove ¤action(p)¤ from line 12 and insert it in line 8.

11.2 a) No, ¤print¤ has only one parameter. b) No, ¤nil¤ can only be hipped, but since it must also be united, it is therefore in a firm position [R.8.2.4.1.b]. c) 1 [R.10.5.1.1.f, 10.5.0.2 Table 1]. d) +3.140000E +0. e) +3.140000E +0.

11.3 a) Undefined, since the repetitive statement is void and therefore cannot be coerced to ¤printtype¤. b) No [R.8.2.4.1.b]. c) Yes, dereference to ¤ref real¤, unite to ¤intype¤ and then row it. d) Undefined, since ¤s¤ cannot be coerced to ¤outtype¤. e) No, ¤format¤ cannot be coerced to ¤[ ]readtype¤.

11.4 a) ? +5. b) ? ABC. c) Twice dereferenced and then united to ¤printtype¤, ? +3.400000E +0. d) Four and 9 spaces left over. e) Nine and 2 spaces left over. f) A B C.

An

ALGOL 68 COMPANION

J.E.L.Peck

Revised Preliminary Edition

March 1972

This document may be ordered from

The Bookstore,
University of British Columbia,
Vancouver 8, B.C.,
Canada.

The price is $2.00 plus handling charges.

....

Please send me ...... copies of An ALGOL 68 COMPANION, and bill me.

Name ....

Address ....

....

....