AN
ALGOL 68 COMPANION

J. E. L. PECK

Revised Preliminary Edition
March 1972
Technical Report 71-01
An

ALGOL 68 COMPANION

J. E. L. Peck

Department of Computer Science
University of British Columbia
Vancouver

Revised Preliminary Edition
March 1972
CONTENTS

Introduction

1 Denotations.

1.1 Language levels. 1.2 Objects. 1.3 Names. 1.4 Variables. 1.5 Denotations. 1.6 Boolean denotations. 1.7 Integral denotations. 1.8 Real denotations. 1.9 Character denotations. 1.10 Modes. 1.11 String denotations. 1.12 Other denotations. 1.13 Program example.

2 Some fundamental concepts.

2.1 Declarers. 2.2 Generators. 2.3 Local generators. 2.4 The elaboration of a generator. 2.5 Identity declarations. 2.6 The syntax of identity declarations. 2.7 Formal parameters. 2.8 An extension. 2.9 An assignment. 2.10 The syntax of assignments. 2.11 References. 2.12 Dereferencing. 2.13 Initialized declarations. 2.14 Program example.

3 Unitary clauses.

3.1 Introduction. 3.2 Bases. 3.3 Identifiers. 3.4 Slices. 3.5 Multiple values. 3.6 Trimmers. 3.7 Calls. 3.8 Void cast packs. 3.9 Cohesions. 3.10 Selections. 3.11 Formulas. 3.12 Confrontations. 3.13 Identity relations. 3.14 Casts. 3.15 Program example.

4 Clauses.

4.1 Conditional clauses. 4.2 Simple extensions of the conditional clause. 4.3 Case clauses. 4.4 Repetitive statements. 4.5 Closed clauses. 4.6 Collateral phrases. 4.7 Serial clauses. 4.8 Program example.

5 Routine denotations and calls.

5.1 The parameter mechanism. 5.2 Routine denotations. 5.3 More on parameters. 5.4 The syntax of routine denotations. 5.5 What happened to the old call by name?. 5.6 Program example.

6 Coercion.

6.1 Fundamentals. 6.2 Classification of coercions. 6.3 Fitting. 6.4 Adjusting. 6.5 Adapting. 6.6 Syntactic position. 6.7 Coercends. 6.8 A significant example. 6.9 The syntactic machine.
This book is not intended to be a language ALGOL 68. The Report on the ALGOL 68 language is the document and it must be referred to in order to get the necessary precision in meaning. Of their proper intent or excessive amount of close reading, the Report specifies that, in a proper context, the Report language is the one to which a meaning in a context beyond all possible doubt has been assigned. Likewise, the student of law does not have to read the Report in order to understand statutes. Likewise, the student of programming must not read the Report in order to understand the language. The Report is not intended for general use, but rather for the student of programming language. Our intention is to provide a transition to the ideas presented in the Report. Since it is assumed that the student has used the Report, our purpose will have been fulfilled if the student can understand the language. This approach means that the reader should refer to the Report in order to understand the concepts presented in this book. We shall use only the terminology that is available in the Report and usually hyphenated, not unduly disturb the
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 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., \texttt{begin\ print("algol\_68")\ end\}, 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., \texttt{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 edition 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{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

ALGOL 68 internal objects (values) are represented thus

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

and paranotions and modes (syntactic parts) are represented thus

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

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

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

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

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

may be replaced by

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

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

```
\begin{verbatim}
\begin{verbatim}
\end{verbatim}
\end{verbatim}
```

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{verbatim}
begin real x; x := 3.14 end
\end{verbatim}
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
\begin{verbatim}
myprogram: (print { real lengths > 1 "multiple" | "single" ) ,
"precision_environment"
\end{verbatim}
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 \[R.2.1\] 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{verbatim}
begin real x; x := 3.14 end
\end{verbatim}
(which happens to be written in the extended language), is a representation of the following sequence of symbols
\begin{verbatim}
\end{verbatim}
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 which deals with "internal" objects and marks which the program possesses, like a collection of computer and the sequence of symbol arrangements of bits on a.

There is an importance to the computer and by the compiler writer means of a diagram along

The reader should quotes some standard reference effort will be made, Report as possible manner the reader may about the language by

The use of a diagram indicates that we are described by the [R.1.1.6.c]. If the
1.2 Objects

ALGOL 68 is described in terms of an hypothetical computer which deals with two kinds of "objects". 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 and an internal object is an arrangement of bits within the computer. For example, when the programmer writes \texttt{\textasciitilde{3}.\textasciitilde{14}c}, he makes, from four symbols, an external object, which is a denotation. 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, \texttt{\textasciitilde{3}.\textasciitilde{14}c}, which is a sequence of symbols, 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" an internal object. Thus, the external object, the denotation \texttt{\textasciitilde{3}.\textasciitilde{14}c}, possesses an internal object which is a collection of bits within the computer. We shall speak of the internal object as a "real value". 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

\begin{verbatim}
\texttt{\textasciitilde{3}.\textasciitilde{14}c}
\end{verbatim}

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. 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 notations \texttt{\textasciitilde{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 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 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 \( \pi 3.141 \), but also with "variables" [R.6.0.1.e] such as \( \nu x \). In ALGOL 68, as in many other languages, if a programmer wishes to consider \( \nu x \) as a variable, he writes a "declaration" [R.7.4.1], e.g., \( \nu x \colon \text{real} \). 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 \( \nu x \). In figure 1.4 the relationship of possession

\[
\begin{align*}
(1) &: o\quad \overrightarrow{------} \quad [ ] \quad (1) \\
(2) &: o\quad \overrightarrow{------} \quad [ ] \quad (2)
\end{align*}
\]

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 except for "nil" [R.2.2.2.1].
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.2.2.4.7] 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, \texttt{true}, 12, 5.67, and \texttt{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", \texttt{true}, and the "false-symbol", \texttt{false}. At the risk of tedious repetition, but for further emphasis, we observe that the external object \texttt{true} possesses an internal object, which is the boolean value \texttt{true},

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig1.6}
\caption{Fig.1.6}
\end{figure}

a value of mode "boolean" (see figure 1.6). Of course, a similar statement applies to \texttt{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.5.1.1.1.a]

\begin{equation}
\text{boolean denotation : true symbol ; false symbol.}
\end{equation}

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, \texttt{n3u} or \texttt{u0m} or \texttt{m000123m}, 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

\begin{equation}
\text{integral denotation : digit token sequence.}
\end{equation}

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 n00123 is 123, which is equal to that possessed
by n123n.

1.8 Real denotations

There are two kinds of real-denotation [R.5.1.2]. Some
examples are: n3.14, .000123, 123.45e6, 5e-16, 4.7591012n. 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 ;
boolean point numeral ;

variable-point-numerals have an optional integral-part, like
n123n, followed by a mandatory fractional-part, like n.14n or
n.000123n. 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 n123.0,
3.456, .12335n and n.00023n but not n3n. 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.

An exponential denotation: n123n or n123n.
Examples of floating-point-number written in scientific
denotations: n3.14, .000123, 123.45e6, 5e-16, 4.7591012n.

Examples of floating-point-number written for computer
input hardware to vary the number of digits printed, for
example, possess four-digit accuracy. Thus both n123n and
1.9 Character denotations

Some characters like n123n or n123n.
Examples of floating-point-number written in
scientific denotations: n3.14, .000123, 123.45e6, 5e-16, 4.7591012n.

Examples of floating-point-number written for computer
input hardware to vary the number of digits printed, for
example, possess four-digit accuracy. Thus both n123n and

To review the character denotations: n123n or n123n.

Examples of floating-point-number written in scientific
denotations: n3.14, .000123, 123.45e6, 5e-16, 4.7591012n.

Examples of floating-point-number written for computer
input hardware to vary the number of digits printed, for
example, possess four-digit accuracy. Thus both n123n and

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

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
an identifier, 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 \(123n\) or \(123.45n\), followed by an *exponent-part*, like \(e+2\), \(e-16\) or \(e^{105n}\). Its syntax is in the rule

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

Examples of *floating-point-numerals* are therefore, \(n1\), \(2.3e-4n\) and \(n.3e26n\) but not \(n3.14n\). The *denotation* \(n.3e26n\), for example, possesses a real value, usually associated with the number written in physics textbooks as \(3\times10^{26}\). 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 \(n123n\) and \(n123.45n\) are acceptable *stagnant-parts*. The *exponent-part* is described in the rules

*exponent part : times ten to the power choice, power of ten. 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 \(a_{10}\), 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 \(e-5\), \(e4\), \(e+56n\) and \(e^{102n}\).

To review the above, we give some more examples of *real-denotations*: \(n123.4, .56789, 464.6e-53n\) and \(n987^{102n}\). Note that \(n123n\) 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 \(n123n\) permitted, ambiguities would arise. Also, \(ne15n\), 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\text{rd}\) and \("\text{mmm}\). 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* \("\text{mmm}\) 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

![Figure 1.10](image)

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""n. Observe that in the strict language, the representation of the •space-symbol• is •n [R.3.1.1.b]. The only feature in the above syntax, which we have

```plaintext
"abc"
```

... :

```plaintext
----------:
| a | b | c |
```

![Figure 1.11](image)

1.12 Other denotations

This discussion of 68, but it is useful to inspect the parts of the language denotations like [R.5.5].

1.13 Program example

Though we are not helpful to inspect the ideas. The following standard input file arithmetic mean of the

```plaintext
\begin{verbatim}
real s := 0
ss := 0
x := the current arithmetic mean of the
\end{verbatim}
```

Points of releva...
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\textquotesingle \textquotesingle\textquotesingle n is not a "string-denotation", but that n\textquotesingle \textquotesingle\textquotesingle n is. Since we have already seen that n\textquotesingle \textquotesingle\textquotesingle n 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 n\textquotesingle \textquotesingle\textquotesingle n 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 0n [R.5.1.0.1.b], "long-real-denotations" like nlong .ln, "bits-denotations" like n10ln [R.5.2.1], "routine-denotations" like n((\texttt{real} a,b) real : (a > b | a | b ))n [R.5.4] and "format-denotations" like c$16x37d$n [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{verbatim}
\begn real s := 0 \texttt{for the sum of the values},
     ss := 0 \texttt{for the sum of squares},
     x \texttt{the current value};
\lst n := 0 \texttt{for a count of the number of values};
\texttt{while} \logical file ended(standin) \texttt{do}
\begin{verbatim}
\{ get(standin, x) \texttt{ R.10.5.2.2.bf};
      s += x; ss += x ** 2; n += 1 \texttt{ R.10.2.11.d,e};
\}
\texttt{put(standout, \texttt{ R.10.5.2.1.bf} \"count = \n, \"mean = \n, \"\text{standard deviation} = \n, sqrt((ss - s ** 2 / n) / n) \texttt{ R.10.3.bf})}
\endv
\end{verbatim}

end
\end{verbatim}

Points of relevance to this chapter are that there are four "variables" ns, ss, x_n and n_n, some of which are initialized with the value zero. Also, the "integral-denotation" n0n occurs three times and the "integral-denotation" n1n, 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.1.1.2.6] 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, 9.2.9]?

1.2 Objects
a) What kind of object is possessed by the denotation n3.14 [R.2.2.2.2.4]?
b) What object may possess a real value?
c) Is n3.14 an internal object or an external object?
d) Does ntrue possess true or does true possess ntrue?

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.4.4.2.4] of nreal x, can the name possessed by nre refer to an integral value?
b) May nreal x, y, z be a declaration [R.9.2.1.2]?

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.2.2.4.1.1]?

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 n-123n an integral-denotation [R.5.1.1.1.1]?
c) Can a sequence of one thousand digits be an integral-denotation?
d) Does every integral-denotation possess a value [R.5.1.1.2.2]?

1.8 Real denotations
a) Can two different
b) Is n.1.a a real-
c) Is n12n a real-
d) Is n12e-4n a real-
e) Is n-12e4n a real-

1.9 Character den
a) Is n"" a char-
b) Does every character-

1.10 Modes
a) How many different
b) How many different

1.11 String den
a) Is n"""" a strin-
b) Is n"""" a string-
c) What is the mode
denotation?

1.12 Other den
a) Are the values same?
b) What is the mode?
c) What is the mode?

1.13 Program ex
a) What is the mode
b) What are the mode
c) Does the example
d) How many integers
e) Does the example
1.8 Real denotations
a) Can two different *real-denotations* possess equal values?
b) Is $1.0$ a *real-denotation*?
c) Is $12.0$ a *real-denotation*?
d) Is $12e-4$ a *real-denotation*?
e) Is $-12e-4$ a *real-denotation*?

1.9 Character denotations
a) Is "a" 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"" a *string-denotation*?
b) Is ""n"" 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 $\text{long} \ 0n$ and $\text{long} \ \text{long} \ 0n$ the same?
b) What is the mode of the value possessed by $\text{long} \ 01n$ [R.5.2]?
c) What is the mode of the value possessed by $\text{long} \ \text{long} \ 0n$?

1.13 Program example
a) What is the mode of the value possessed by "count~x~z"?
b) What are the modes of $\text{nn}n$ and $\text{nn}n$?
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

- og\[\]i for •integral•
- o\[\] for •real•
- o\[\]i for •boolean•
- o\[\]c for •character•
- o\[\]f for •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• o\[\]i. A •declarer• specifying the mode •row-of-real• is o\[\]i or if actual bounds are required, then say, of 1:10\[\]i. The mode of a real vector variable is •reference to row of real• and this mode is specified by a declarer like o\[\]i or o\[\]i[1:10]. We see, therefore, that other •declarers• may be built from the primitives by using the symbols o\[\]i for •reference to• and o\[\] for •row of•.

Other possible prefixes are o\[\]\[\]i for •reference to• and o\[\]\[\]i for •row of•. 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:

- o\[\]i real, [1:10 flex]chat, proc [real]real, [1:10]format, proc, struct [real re, is], union [real, int, bool]i.

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•, olocu, followed by a •declarer•, e.g., oloc into. A •global-generator• is an optional •heap-symbol•, oheap, followed by a •declarer•, e.g., oheap real or oheap. 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:

\[
\text{local generator : local symbol, actual declarer.}
\]
but the strict syntactic rule [R.8.5.1.1.b], in common with many other rules, containing a feature which the reader should now observe. The rule is

- reference to \texttt{MODE} local generator:
  local symbol, actual \texttt{MODE} declarer.

The feature to be noticed is the occurrence of the "metanotion" \texttt{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.7.1]. 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 \texttt{MODE} by a mode (one of the terminal productions [R.1.1.3.2] of \texttt{MODE} like \texttt{integral} or \texttt{real}), then we obtain a rule of the strict language. For example, if we replace \texttt{MODE} by \texttt{real}, we obtain the production rule

- reference to \texttt{real} local generator:
  local symbol, actual \texttt{real} declarer.

If we replace it by \texttt{boolean}, we obtain the rule

- reference to \texttt{boolean} local generator:
  local symbol, actual \texttt{boolean} declarer.

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

2.4 The elaboration of a generator

The "elaboration" of a \texttt{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 \texttt{generator} [R.8.5.1.2]. A \texttt{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 language, that we will attempt to make it clear by means of a diagram. We may picture the elaboration of the \texttt{generator} \texttt{iloc realm}, 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•, mloc real, thus creates a name which refers to some real value. The external object, mloc 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.6.3.1.2.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•.

before the elaboration of the •identity-declaration•, as in figure 2.5.a. After the elaboration of the declaration, int m = 4096n, 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 n4096n. 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 := 0n. In fact, m := 0n would be just as improper as n4096 := 0n. The •identifier• m is thus a •constant• [R.6.0.1.d].

The possession of a variable, is unlike the data area. It is essential that the reference may not be unlocked. Thus the name in order to refer and which can

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

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

[1.4.0.1.8]. The first step in the elaboration of this •identity-declaration• is the elaboration of its •actual-parameter•, which is nloc real. We have seen, in 2.4, that this will make mloc 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• oxo possesses the same value as that possessed by nloc realn. The result, in pictorial form, is shown in figure 2.5.d. Here, because oxo 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.

\[
\text{ref real } x = \text{loc realn} \\
\begin{array}{c}
\vdots \\
0 \\
0 \\
0 \\
0
\end{array} \\
\begin{array}{c}
\vdots \\
0 \\
0 \\
0
\end{array}
\]

(Fig.2.5.c) 

\[
\text{ref real } x = \text{loc realn} \\
\begin{array}{c}
\vdots \\
0 \\
0 \\
0 \\
0
\end{array} \\
\begin{array}{c}
\vdots \\
0 \\
0 \\
0
\end{array}
\]

(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 nint m = 4096c and nref real x = loc realn.

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 metanotic 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• \( \texttt{eref real \( x = \text{loc real} \) \( x \) n} \) (viz.), the mode of the •generator• \( \text{loc real} \) is •reference to real• and that of the •formal-parameter• \( \text{eref real \( x \) n} \) is also •reference to real•. It follows from the rule on •formal-parameters• [R.5.4.1.e], that \( x \) n is then a •reference-to-real•.

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 metanotic •MODE• appears three times. By substitution we obtain the rule applicable to the •formal-parameter• \( \text{eref real \( x \) n, viz.} \)

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

The •formal-reference-to-real-declarer• is \( \text{eref real} \) and the •reference-to-real-mode-identifier• is \( x \) n [R.4.2.2].

2.8 An extension

The object \( \text{eref real \( x = \text{loc real} \) \( x \) n} \) 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

\( \text{eref real \( x \) n} \)

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

\[
\begin{align*}
(1) &= \text{eref real \( x = \text{loc real} \) \( x \) n} \\
\text{xxxxx} &= \text{xxxxx} \quad (2)
\end{align*}
\]

Fig.2.8

effect of this temptation to call in parts which are unde
the •identifier• is that in the exten
•declarer\( \text{eref real} \) •actual-declarer\( \text{eref real} \) remains. This is of multiple values.

Another extension e.g., \( \text{eref real \( x \) real} \)

In the examples \( y, \text{int \( i \), \( j \), \( n \), \( \{1:1\} \) \}
unless contradicted \( \text{ref real} \) have the mode •reference to integr
to real•. It remains. This is of multiple values.

2.9 An assignment

We have seen before which to unlock the secret when an assignment i

\( \text{eref real \( x = \text{loc real} \) \( x \) n} \) (in the reach of \( \text{eref real} \) assignment [R.8.3.1].

\( \text{eref real} \) \( x \) n, a •source\( \text{eref real} \) •becomes-symbol•, \( \text{eref real} \) •destination• are "collaterally" [R.8]

obtain the values

\[
\begin{align*}
(1) &= \text{eref real} \\
\text{xxxxx} &= \text{xxxxx} \quad (2)
\end{align*}
\]

\( \text{eref real} \) is the

\( \text{eref real} \) to the name possesses precisely, the name

(new instance) of the

an •assignment•, a
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 *'s in figure 2.8, and then move the *identifier* in the manner indicated (provided that the following symbol is \( n,n',n \) or \( n=m \)). It is important to note that in the extended *declaration* \( \text{real } x \), the *formal-declarer* \( \text{real } x \) (see figure 2.8 at 1) is omitted, but the *actual-declarer* \( \text{real } x \) (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., \( \text{real } x \), \( \text{real } y \) may be written \( \text{real } x \), \( \text{real } y \) \[R.9.2.c\].

In the examples which follow, the *declarations* \( \text{real } x \), \( \text{real } y \), \( \text{int } i \), \( j \), \( n \), \([1:10]\) \( \text{real } x_1 \), \( y_1 \) will always be assumed. Thus, unless contradicted by another *declaration*, \( x_1 \) and \( y_1 \) will have the mode *reference to real*, \( u_1 \), \( j_1 \) and \( u_1 \) the mode *reference to integral* and \( u_1 \) and \( y_1 \) 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

\[ nx := 3.14n \]

(in the reach of the *declaration* \( \text{real } x_n \)), 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 \( nx \), a *source*, which is \( 3.14n \), 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

\[
\begin{array}{l}
\text{reference-to-real-assignation} \\
\quad \text{reference-to-real-destination becomes-symbol real-source} \\
\quad \text{nx } := 3.14n \\
\quad : (3) \\
\quad : (1) \\
\quad : 0 \rightarrow \text{-} \\
\quad \text{0 
\end{array}
\]

\[
\begin{array}{l}
\text{Fig.2.9} \\
\text{assignation}\text{ is the assignment of the value possessed by } 3.14n \\
to the name possessed by } nx \text{ (see figure 2.9 at 2). More precisely, the name possessed by } nx \text{ is made to refer to a copy (new instance) of the value possessed by } 3.14n \text{ [R.8.3.1.2.c,d]. An assignation, after its elaboration, possesses a value and}
\end{array}
\]
the value possessed is that of its destination, which is a
name (see figure at 3).

2.10 The syntax of assignments

We should now examine the syntax of assignments, in
particular, the rule

*reference to MODE assignment : becomes symbol, MODE source.*

[8.8.3.1.1.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 assignment :
reference to real destination,
becomes symbol( real source.*

The important point to notice about this rule, which is the rule
governing the object ox := 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 assignment 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

\begin{verbatim}
  int m = 4096 ; m := 4095n
\end{verbatim}

and decide that m := 4095n cannot be an assignment, 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 [8.6.0.1.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 [8.8.6.2.2] 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 n2.3 + 4.5n and
\(\alpha + \nu\). 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

\(\text{(1)}\) Except for the languages LISP, SNOBOL and TRAC.
elementary. We pursue this point a little further in our next paragraph.

2.12 Dereferencing

If \( mx := 3.1415 \) is an \textit{assignation}, then surely \( mx := y0 \) (in the reach of the declaration \( \text{real} \ y0 \)) must be also. However, the mode of \( mx \) and that of \( y0 \) is \textit{reference to real}, while an \textit{assignation} requires that the mode of the \textit{destination} should be \textit{reference to} followed by the mode of the \textit{source}. This means that the mode of \( y0 \) should be \textit{real}. It would seem then, that this object does not fit immediately into the syntax of \textit{assignations}. However, it is an \textit{assignation}. Diagrammatically, the situation is shown in figure 2.12. The first step is the elaboration of the \textit{source} and the \textit{destination} collaterally [R.6.2.2.a] (figure 2.12 at 1,2,3 and 4). However, the \textit{source}, in this object, requires an extra step in its elaboration. Since \( y0 \) 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

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 \( y0 \), as a \textit{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 \textit{assignation} is a name of mode \textit{reference to real} (see the figure at 6).

The syntactic analysis of the \textit{assignation}, \( mx := y0 \), 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 \( y0 \), which is of a priori mode \textit{reference to real}, can be considered, a posteriori, of mode \textit{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 ox := y is indeed an •assignment•
even though the a priori node of ox is not •real•.

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

2.13 Initialized declarations

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

actual parameter : unit ; ...
unit : unitary clause .
unitary clause : ... ; confrontation ... ...

Since mloc real := 3.14n is an •assignment•, this means that
mloc real x = loc real := 3.14n is an •identity-declaration•.
But we have seen that the object mref real x = loc real x
may be written mreal x [R.9.2.a]. This means that mreal x := 3.14n is
also an •identity-declaration• with the same meaning as that of
mref real x = loc real x := 3.14n. This meaning should now be
evident once it is realized that the •assignment•, 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,
the number of times per year that the interest is converted,
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{verbatim}
begin long real p $\text{the principal}$,
   r $\text{the interest rate per unit per year}$,
   mp $\text{the constant monthly payment}$,
   int t $\text{the number of times per year that the interest is}
   converted}$,
   y $\text{the number of years}$ :

start here : read((p, r, t, mp, y)) ;

The output from

REPAYMENT SCHEDULE
INTEREST RATE PER MONTHLY PAYMENT
MONTH AMOUNT
1  906.62
2  812.63
3  718.07
4  622.76
5  526.89
6  430.38
7  333.23
8  235.43
9  136.99
10 37.90
11 0.00

\end{verbatim}
outf(standout,
   "$1"repayment_schedule_of_a_loan_of_"9zd.2d,
   "$interest_rate_per_unit_"d.4d,
   "$converted_"2zd"_times_per_year",
   "$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(long(t / 12) * longln(long 1.0 + r / long 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, "$1 2x8a, 3(12a)$,
      ("month", "amount", "principal", "interest")) ;
   format(standout, "$1 4zd, 3(7zd.2d)$)
   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 outf(standout, (j, 0.0, p, interest))
      else (regular monthly payment, p := ap - mp ;
      outf(standout, (j, p, mp-interest, interest)) ;
      go to again
   enda

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 oreal ref a •declarer•?
b) Is oref ref ref ref real a •declarer•?
c) Write down a •declarer• specifying the mode •reference to reference to row of character•.
d) Is of format o real a •declarer•?
e) Is oref format o ref real a •declarer•?
f) Is oreal proc o 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.8.1.7.5.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• o loc real, after elaboration, possess a real value?
b) Does the •generator• o 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.7.2.3.5.a]?
d) Write a •declaration• of mnp as a •reference-to-row-of-procedure-real-mode-identifier•.

2.7 Formal parameters

a) Is \texttt{real} \texttt{mn} a •formal-parameter•?
b) Is \texttt{proc} \texttt{real} \texttt{gmn} a •formal-parameter•?
c) Is \texttt{loc} \texttt{real} a •formal-parameter•?
d) Is \texttt{int} \texttt{ln} a •formal-parameter•?

2.8 An extension

a) Write the •declaration• \texttt{ref real xxn} in the strict language.
b) Write the •declaration• \texttt{real x, ym} in the strict language.
c) Write the •declaration• \texttt{real x, y := 3.14n} in the strict language.
d) Write \texttt{ref ref real xx = loc Ref real + 3.14n} in the extended language [R.9.2.a].

2.9 An assignation

a) Is \texttt{x := 3.4n} an •assignation•?
b) Does an •assignation•, after elaboration, possess a value?
c) Can an •assignation•, after elaboration, possess a real value?
d) Is \texttt{n(x := 3.14) := 3.15n} an •assignation•?

2.10 Syntax of assignations

a) Is \texttt{loc real := 2.3n} an •assignation•?
b) Is \texttt{loc ref real := xmn} an •assignation•?
c) Is \texttt{loc ref real := 3.14n} an •assignation•?
d) What is the •source• in the •assignation• \texttt{ox := y + 2n}?
e) What is the mode of the •assignation• \texttt{uxx := xmn} (in the reach of \texttt{ref real xx, real xmn})?
f) In the reach of \texttt{bool t = true}, is \texttt{ct := false} an •assignation•?

2.12 Dereferencing

a) What is the essential difference between the elaboration of \texttt{mx := ym and nx := 3.14n}? 
b) Is any dereferencing necessary in the •assignation• \texttt{uxx := xmn} in the reach of \texttt{ref real xx, real xmn}?

2.13 Initialized declarations

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

2.14 Program example
3 Unitary clauses

3.1 Introduction

The *unitary* blocks of the language as the statement of *unitary-clauses* as the statement or *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-clauses* as the statement of *unitary-cla
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, \( nx := y, x + y, \) re of \( z, 123n \) and \( n( x := 1 ; y := 2 )n \). *Unitary-clauses* are classified further into *confrontations, formulas, cohesions, bases* and other objects like *closed-clauses*. Thus, \( nx := yn \) is a *confrontation*, \( nx + yn \) is a *formula*, re of \( zn \) is a *cohesion*, \( c123n \) is a *base* and \( n( x := 1 ; y := 2 )n \) 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.

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
\[ na \text{ or } b := c \text{ of } d \text{ of } e[f] - gn^{(1)} \]
where the modes of \( na, b, c, d, e, fn \) and \( gpn \) are unknown. In fact the order is as if we wrote

\[-------
^{(1)} \text{Note that the operator } \text{norm may be declared in such a way that it delivers a name.} \]
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 parses.

**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 \( pi \), \( i23 \), \( a[1] \), \( \sin(X)u \) and \( u: \) random \( j \). A simplified syntax for **base** is

\[
\text{base : mode-identifier ; denotation ; slice ; call ; void cast pack.}
\]

but the strict syntax of the report should be studied [R.6.0.0.1]. **Identifiers** are as in other programming languages, e.g., \( \text{random} \) and \( x14283c \). **Denotations** we have met before in section 1.5, e.g., \( w_{75m} \) is an **integral-denotation**, \( m_{3.1n} \) is a **real-denotation**, \( m_{\text{false}} \) is a **boolean-denotation**, \( m_{\text{"abc"}} \) is a **character-denotation** and \( m_{\text{"abcdefg"}} \) is a **string-denotation**. Thus we are already familiar with several objects which are **bases**. The objects \( \text{ox1[i]} c \) and \( \text{ox2[4;e]} j \) are **slices**, \( \text{msin(x)}u \) is a **call** and \( u: \) random \( j \) 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 **coercion** [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

\[
\text{identifier : letter ; identifier , letter ; identifier , digit.}
\]

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, \( \text{malgo1 68} \), \( a, a_{7649} \), \( \text{random} \), \( st \) pierre de chartreusse (note that spaces are of no significance within **identifiers**) [R.4.1.6.a].

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 **assignment** \( nx := y + 3c \), the **mode-identifier** \( nx \), supposedly in the reach of the **declaration** \( \text{real } nx \), possesses a name which refers to some real value. The value possessed is, in fact, the same **identifier** at the occurrence indicated at 2. In the occurrence of \( oxn \) possess the first occurrence of \( \text{same instance of a} \), consult the report description of the **identifiers** is made.

### 3.4 Slices

We continue our **denotations**, but we go on to **slices**. In

\[
x1, [1:m,1:n] \text{real } x2, \quad \text{ax}[i], \quad x2[i,1] \quad \text{A simplified syntax of slice} : \text{primary , su indexer , trimscrip} : \text{trimscript} : \text{trimscript} : \text{trimscript}
\]

but the strict syntax is more than the skeleton.

The most important **field-selector** is **primary**. Also notice a **primary**. Following symbol, represented **bus-symbol**, represents the above examples, \( \text{ak, bkn} \). An **indexer** is also represented by the objects \( \text{min} \) and **trimmer**. A **subscript**

In order to account for limited character set written **nx1(i)** [R.9...
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 oxo 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 oxo in
\[
\text{real } x : x := y + 3n
\]
is shown pictorially in the figure 3.3,

\[
\begin{align*}
\text{Fig.3.3}
\end{align*}
\]

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 oxo possesses a copy of the name possessed by the first occurrence of oxo. 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*. In the reach of the *declarations* u[1:n]real x1, [1:m,1:n]real x2n, the following are examples of *slices*.

- \(u_1[i], x_2[i,j], x_2[i,:], x_1[2:n], x_2[i,0], x_2[i]\)

A simplified syntax of *slice* is

\[
\text{slice} : \text{primary, sub symbol, indexer, bus symbol.}
\]

- \(\text{indexer} : \text{trimscript, \text{indexer, comma symbol, trimscript.}}\)
- \(\text{trimscript} : \text{trimmer, subscript.}\)

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 \(u_1[i] n\) in \(u_1[i,j]n\), 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 \(n, o\), then an *indexer* and finally a *bus-symbol*, represented by \(j, n\). Thus all of the following, in the above examples, are *indexers*: \(n, j, n, j, n\). An *indexer* is one or more *trimscripts*, separated by *comma-symbols*. A *trimscript* is a *trimmer* or a *subscript*. The objects \(n, j, n\) are *subscripts* and \(n, j, n\) are *trimmers*. A *subscript* is an *integral-tertiary*.

In order to accommodate those users whose computers have a limited character set, a *slice* like \(u_1[i,j]n\) may also be written \(u_1(i)n\) [R.9.2.d]. 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

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\hline
5 & 6 & 7 & 8 \\
\hline
9 & 10 & 11 & 12
\end{array}
\]

Figure 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•

\[
\begin{align*}
\text{int } n1 &: = (1, 2, 3); \\
\text{int } u1 &: = (1, 2, 3); \\
\end{align*}
\]

\[
\begin{align*}
\text{int } n1 &: = (1, 2, 3); \\
\text{int } u1 &: = (1, 2, 3)\text{.}
\end{align*}
\]

The object \((1, 2, 3)\) looks and acts like a •denotation• of a row of integers, but it is actually a •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 \(n1\) is a multiple •constant• and \(n1\) is multiple •variable•. The "D" in the figure, at 1, indicates that a "descriptor" [R.2.2.3.1.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 expect to obtain a •constant•, e.g., \(n1[2]\) 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 •assignment• but \(n1[2] := 4\) 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• \(n1[2]\) [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.

Observe now the use of the Report.

3.6 Trimmers

A programmer who is to choose certain subexternal object to pass for example, one may wish a matrix or even a sub by using a •trimmer•, a single element, illustrate the use of \(n[1:3]j\) \(n1 := (5, 7, 9)\), referring, at the moment •variable• referring to •9•; moreover, being •formula• referring •null• \(n1[2][1] \neq \text{fact, it will always made to } n1\text{. Another way to re-}

The effect of the range of values of the value \(\text{of } n1\), the new lower bound. This \(\text{new-lower-bound}\) is actually built into the examination of the synt the \(n1\), the new lower bound• or the •upper-bound• empty [R.8.6.1.1.f]. If empty, then the lower position, is the same as •sliced; if the •upper-bound
Observe now the use of the word "weak" in the rule 3.6.1.1.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:

\[
\text{int } n := (5, 7, 9) \text{.}
\]

The "slice" \[ n[2] \text{a} \] is a "variable" referring, at the moment, to \(7\), but the "slice" \[ n[2:3] \text{a} \] 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 \[ n[2:3][1] \text{a} \] is a "variable" referring to the same integral value \(7\) and the formula \[ n[2:3][1] = n[2] \text{a} \] possesses the value true. In fact, it will always be true no matter what assignments are made to \(n[2] \text{a}\). Another way of saying this is that the "identity relation" \(n[2:3][1] := n[2] \text{a} \) possesses the value true.

The effect of the "trimmer" \(\text{alo} \) is then to restrict the range of values of the subscript to run from the value of \(\text{olo}\) and to renumber, starting from \(1\). If the renumbering from \(1\) is not desired, then the "trimmer" should be written \(\text{alo} \text{abo}\), where the value of \(\text{abo}\) is to be taken as the new lower bound. This means that, e.g., \[ n[2:3][0] := n[2] \text{a} \] possesses the value true. We may think of this as the sense that if \(\text{abo}\) is omitted, then the default value of \(\text{abo}\) 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 \(\text{alo}\), the \(\text{uno}\) and the \(\text{abo}\) 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 "upper-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 "subscript" of
the *slice*, in that subscript position, will start from *m*. It is even possible for all three to be empty at the same time. Thus on1[1] := n1[1:3].p will possess the value *true*. Extension 9.2.f, in the Report, allows the *up-to-symbol* to be elided, under conditions specified in the above *identity-relation* might be written on1[1] := n1[1:3].p.

If the *declaration* of 1:a, l:n]real x2n is used as that of an n by n matrix, then x2[i][j] refers to the i-th row of the matrix, x2f[i][j], or even x2[i][j] := [R.9.2.f], to the j-th column and x2f[a;b,c;d] may refer to a certain submatrix, if the values of na, b, bc and d are appropriate. The rules for *trimmers* [R.9.6.1.1.f,g,h] should appear as x2f(i) := [R.9.2.f], to the j-th element, or even x2[i][j] := [R.9.2.f], to the j-th element. In particular, a *formula* is a *tertiary* but an *assignment* is not. So that x2[i] := i + 1, j of x2 is an acceptable *slice* but x2[i] := i + 1, j of x2 is not. The latter, to be acceptable, should appear as x2f[i := i + 1], j of x2.

3.7 Calls

A simplified syntax of a *call* is:

```
call : primary, open symbol, actual parameters, close symbol.
actual parameters : actual parameter.
actual parameter : primary, open symbol, cast c
```

goal: go in certain circumstances, so that the *actual parameters* will possess the value of the *primary* as that of the *call*. This possibility is present so that the *primary* may, if so wishes, take up in chapter 3.9 Coherions, where the *slice* may be used as a *call*. This possibility is present so that the *primary* may, if so wishes, take up the *call*.

Thus, in the *call* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position 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* of f(n; a, b)n, the nns might serve as a bound for the arrays na and nb, provided that a *go-on-symbol* was used in a similar position in the corresponding *routine-denotation* [R.5.4.1], where its use will force the elaboration of the *actual parameters* serially rather than collaterally.

3.8 Void cast packs

A simplified syntax of a *void cast pack* is:

```
void cast pack : open symbol, cast c.
```

The *void cast pack* is that its first constituent, e.g., using in six(x)n, must be a *primary*. Also notice that a *call* itself is a *primary*, so that na(b)(c)(d)n might well be a *call* in which the order of elaboration is that suggested by o(b)(c) = x2d. As we have already remarked, in section 3.4, in some programs it may not be possible to determine whether

```n(a)b u is a *slice* or not, but since the parameter is of no great harm that the object ruf x
*primary* and therefor
```
ma(b)n is a •slice• or a •call•, without knowing the mode of cac, 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(x)n is a •primary• and therefore if x < pi/2 then cos else sin fi(x)n is a •call•. It so happens that nbegin r := s + 2 ; sin end (x)n 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

\[ \text{void} : x := 2 * x + 1 \]

Its purpose is to void the mode of the •unit• contained therein in those situations where this is not done implicitly, such as in u; 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

\[ \text{uproc } \text{void } p = (\text{void} : x := 2 * x + 1) \]

where uproc is made to possess a routine, which contains an •assignation• but the •assignation• should not itself be elaborated until uproc is called. The object uproc void p = (x := 2 * x + 1)n 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, 61a, 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

\[ \text{void cast pack : open symbol, cast of symbol, unitary clause, close symbol.} \]

but the strict syntax is found in more than one place in the Report [R.8.3.4.1.a, 3.0.1.b, 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., •reald•, or a •selection•, e.g., •one of z•. The strict syntax is:

\[ \text{MODE cohesion : MODE generator ; MODE selection.} \]

[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 one of zn in the reach of the declaration struct(real re, im) zn. A simplified syntax of selection is

\[ \text{selection : field selector, of symbol, secondary.} \]

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 are of zn, the *field-selector* is zer.

An example of a declarer which specifies a structured mode is struct(real value, string name)n. 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* orn referring to a value of such a mode, this is done by using the selection-value of ro; the string field is obtained by the *field-selector* name of zn. Note the similarity with the slice ox!i]i], where an element is selected from the value of the *primary* according to the value of the *subscript* min. In the selection value of ro, an element is selected from the value of the *secondary* m, using the *field-selector* valuec. There is, however, one essential difference in that the value of the subscript, min, may vary dynamically, whereas the field-selector, valuec, 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 h) ib := (1, true)n and a struct(real r, char c) rc = (1.2, "k")n, ni of ibn is a variable and ni of ib := 2n is an acceptable *assiguation*; however, uc of rcm is a constant and uc of rc := "m"n is not permitted. The reader may wish to note that these effects are obtained, syntactically, through the use of the metanotion REFET 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 na of b[c]n must be the same as that of na of (b[c])n, for na of bn is not a primary. Also, a selection is itself a secondary so that na of b of c of dn may be a selection whose order of elaboration is suggested by na of (b of (c of d))n. Observe that if ndu is a

3.11 Formulas

A simplified formula: operator: monadic operator: operand: tertiary. Also, the strict syntax of the Report [R.8.6.1.1.a, 8.6.1.1.b,c,d].

The syntax of a tertiary. Also, the syntax shows that a formula deduce that the elaborated object is found the following in the order suggested, not discussed, to be in the secondary is operand-declaration.

A major new feature is the introduction of a *variable* then new in which monadic operator: operand: primary: tertiary. Also, the syntax shows that a formula deduce that the elaborated object is found the following in the order suggested, not discussed, to be in the secondary is operand-declaration.

A set of standard operators: operator: monadic operator: operand: tertiary. Also, the syntax shows that a formula deduce that the elaborated object is found the following in the order suggested, not discussed, to be in the secondary is operand-declaration.

\[
\begin{align*}
\text{DYADIC} \\
\text{?} & 2 & 3 & 4 \\
n := & o & (6) & = \\
+ := & + & = & + \\
* := & / & := \\
+ := & + & := \\
* := & * & := \\
\end{align*}
\]

A simplified formula: operator: monadic operator: operand: tertiary. Also, the strict syntax of the Report [R.8.6.1.1.a, 8.6.1.1.b,c,d].

The syntax of a tertiary. Also, the syntax shows that a formula deduce that the elaborated object is found the following in the order suggested, not discussed, to be in the secondary is operand-declaration.

A major new feature is the introduction of a *variable* then new in which monadic operator: operand: primary: tertiary. Also, the syntax shows that a formula deduce that the elaborated object is found the following in the order suggested, not discussed, to be in the secondary is operand-declaration.
•variable• then ma of b of c of d is also a •variable•.

3.11 Formulas

A simplified syntax of •formula• is

\[ \text{formula : operand, dyadic operator, operand ;} \]
\[ \text{monadic operator , operand.} \]

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 \( a - b \cdot c \), 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., \( \cdot \cdot \cdot \), 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

\[ \text{nop or = (real a, b)real : if a > b then a else b fin} \]

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• \( a \cdot b[i] + c \) is in the order suggested by \( c(a \cdot ((b[i])) + c) \). 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

\[
\begin{array}{ccccccccccc}
\text{DYADIC} & & & & & & & & & & \text{MONADIC} \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 (10) \\
\hline
\text{:=} & \text{of} & \text{=} & < & * & \text{up i} & \leftarrow & - & + \text{down up} \\
\text{+=} & \text{=} & \leq & + & \text{lwb} & \text{abs bin repr} \\
\text{*:} & \geq & + & : & \text{upb} & \text{lwb upb lws ups} \\
/=: & \geq & + & / & \text{lws} & \text{leng short} \\
\text{+:} & \text{elem ups} & \text{odd sign round} \\
\text{+:} & \text{re in cond} \\
\text{+:} & \text{bth} & \text{cth} \\
\end{array}
\]

Fig. 3.11
The reader •operators• have no i-times-symbol in four [R.3.1.1.c] available in this the TN print chain.

3.12 Confrontations

There are four strict rule

(1) Here it is more longer but correct

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

There is one standard •dyadic-operator• min or n/n of priority 9 (the programmer may create more if he wishes). The value of nx i y/n is a complex number with real part nx and imaginary part nyn [R.10.2.5.f]. In the standard •declarations• the result of the •dyadic-operator• n/n, •divided-by•, is real (or complex) and that of n/m is integral (integral division of two integral operands). The operator nelem delivers an element from bits or bytes, e.g., n2 elem 101n delivers •false •. Note that n2 elem b := truec is not an •assignation• [R.10.2.8.K, 10.2.9.c]. Manipulation of bits can be achieved with the operators xor, and, upo and unpto [R.10.2.8.d,e,h,m]. The value of on +: mn is mod n mn, i.e., the remainder obtained on dividing mn by mn [R.10.2.3.n]. Apart from the fact that mabs is an operator on real, integral and complex values, rather than a •call•, i.e., it is not mabs(x)n, 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 n+:=m, which we can explain by saying that the •formula• nx +=: 1m has the same effect as nx := x + 1m. Another •dyadic-operator• with priority •1• is n:=:m, which may be used with two •operands• of mode •row of character• [R.10.2.11.r,t]. After elaboration of the •formula• ns +=: tn, in the reach of ns := "abc", t := "def", we have ns = "abc" and nt = "abcdef". On the other hand, after the elaboration of the •formula• ns +=: "gn", we have ns = "abc"gn.

The reader should be careful to note that several •operators• have more than one representation, e.g., the •plus-i-times-symbol• has three representations and the •up-symbol• four [R.3.1.1.c] (moreover, 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 n2*2 = 4n rather than the longer but correct statement n2*2 = 4n possesses the value •true•.
•MODE confrontation : MODE assignation ;
•MODE conformity relation ;
•MODE identity relation ; MODE cast.* [R.8.3.0.1.a]. The object \( ox := y + z \) is an •assignation•, or \( ox := in \) is a •conformity-relation•, or \( ox := ho \) is an •identity-relation• and \( ox := i \) 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• \( c(xx := y) := x \) is done in the order suggested by \( c(xx := y) := x \). Such an •assignation• might well be possible if the •operator• core had 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 \( c := : c \) and the •is-not-symbol•, represented by \( c 
\neq : c \). A simplified syntax of the •identity-relation• is

\[
\text{identity relation : tertiary, identity relator, tertiary.}
\]

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• \( \text{real } x, y \) has been made. The •identity-relation• \( nx := y \) then has the value •false•, despite the possibility that we may have elaborated the •assignations• \( nx := 3.14, y := 3.14 \). This is because the •declaration• \( \text{real } x \) (strictly \( \text{real } x := \text{loc } \text{real} \)) involves the elaboration of the •generator•, \( \text{loc } \text{real} \), which creates a name different from all other names [R.7.1.2.d, Step 8]. The same applies to \( \text{real } y \). Hence, the name possessed by \( nx \) is not the same as the name possessed by \( ny \). After the •declaration• \( \text{real } x := x \), the name possessed by \( nx \) is the same as the name possessed by \( nxn \), but a different instance of that name. Consequently, the value of the •identity-relation• \( nx := an \) will be •true• and will remain •true• no matter what assignments are made to \( nxn \) or to \( nxn \). Notice that an assignment to \( nxn \) is at the same time an assignment to \( nxn \).

Now suppose that the •declaration• \( \text{ref int } i, j, \text{int } n \) is elaborated followed by the •assignations• \( ni := i, jj := in \). The •identity-relation• \( ni := jjn \) possesses the value •false•, for a similar reason to that explained above, but the •identity-relation• \( nij := in \) 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 •tertiaries• of the •identifiers• begins with •reference-de-referencing• [R.8.2.1 at 3], whereupon the •true• (see the figure speaking, a coercion mode and the a posteriori is therefore absent. So the left or on the right alternatives in the •identifiers• [R.8.3.1.1.a]. The role of the •tertiaries•

In the case of \( \text{pj} j := \text{sn} \), the •strong•. This is a balancing of modes which

3.14 Casts

The object

is a trivial example of enough to illustrate followed by a •cast-of•. The purpose of a •cast-clause• into a value of

example given in tr; more easily from the •
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*, ujj (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

<table>
<thead>
<tr>
<th>strong-reference-to-integral-tertiary</th>
<th>soft-reference-to-integral-tertiary</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong-reference-to-integral-base.....</td>
<td>soft-reference-to-integral-base.....</td>
</tr>
<tr>
<td>(coercion) (3) :</td>
<td>(coercion) (3) :</td>
</tr>
<tr>
<td>reference-to-reference to-integral-base :</td>
<td>reference-to-(2) to-integral-base :</td>
</tr>
<tr>
<td>ujj :=</td>
<td>in</td>
</tr>
<tr>
<td>: :</td>
<td></td>
</tr>
<tr>
<td>o o &lt;--(identity) o o</td>
<td></td>
</tr>
<tr>
<td>o o</td>
<td></td>
</tr>
<tr>
<td>[ ] &lt;--</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.13

In the case of ujj :=: in, the pin is soft and the ujjic 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

\[
\text{real} : 2n
\]

is a trivial example of a *cast* [2.8.3.4.1.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* \(n2.0n\).
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, writing: "a"n is a multiple value, i.e., a row of characters with one element, and objects like (Ref Cell: next of cell): := niln 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., n := yu. 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., n(stand x) := yu, or as part of a routine-denotation [R.5.4.1.b], e.g., if get bin(stand back, x):= yu then bin(stand back, x) := yu. 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 ny := yu or nx := yu.

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

\[ \text{cast: virtual declarer, cast of symbol, unitary clause.} \]

[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:

\[
\text{int} \quad (\text{real a}) \text{link} : \text{if a then 1 else 0} \quad \text{fin} ~\text{[R.10.2.2.f]} \quad \text{in which the cast is link: if a then 1 else 0 fin.}
\]

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 : xx is a cast, but its value is the same as that of real : xx. Note that a cast which envelops a mode is not a primary or even a tertiary; consequently, real : real : xx := 3.14c is not an assignment. The effect perhaps intended could be obtained by writing n(Ref real : xx) := 3.14c.

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 upb calculates the real coefficients of a polynomial whose zeros are the elements of a given complex vector zzn. 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)n, then the polynomial will be ⅈ*2 - ⅈ - 1n. Thus, in the *range* of 1:3]comp z := (1, 0 ⅈ 1, 0 ⅈ -1)n, the value of the *call* np(w)n will be that of n1real : (1.0, -1.0, 1.0, -1.0)](30)n. The existence of a non-local *procedure*, errorn, is assumed, for use upon encountering invalid data.

npcc p = (ref 1:]comp z)[1real :
  #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 af
  while i ≤ upb z do
    begin compl zi := z[i]; a[i] := 0;
      if zi = 0 then a real zero#
      for k from i by -1 to 1 do
        a[k] := zi * a[k-1]
      else #a pair of complex zeros#
      if zi ≠ conj zi[i+:=1] then error fi ;
      real s = real zi ** 2 + im zi ** 2, t = 2 * real 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 af
  a end

From of 1real in, on the first line, to the final endn is the *cast* of a *routine-denotation* [R.5.4.1.b]. It begins with 1real : in to ensure that the value delivered by the routine is of mode *row of real*. Note the use of the *operator* upbn in the *declaration* of 0:upb z 1real an, which creates a vector *variable* with index running from 0 to the upper bound of zzn. The *declaration* of compl zi := z[i]n [R.10.2.7.a] indicates that, for each value of in in the iterative statement, zi is a constant. This avoids repeated calculation of nz[i]n later. Observe that, in the *formula* nzi ≠ conj zi[i+:=1]n, the *formula* nzi += in is elaborated first. The value of the *variable* nzi is thus incremented by 1. The value of this *formula* is the name possessed by zi[i+:=1]n, which is the same as the name possessed by nzi. It is then dereferenced. The object nz[i+:=1]n is a *slice* whose value is the next zero of the polynomial sought. The *declaration* of real s = real zi ** 2 + im zi ** 2n declares a *real-constant* nm whose value is the square of the modulus of one of the conjugate pairs. The value delivered by the routine is that of nm; consequently nm appears as an *expression* preceding the final endn.
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 \( ma + b \) of \( c [d] - en. \)
d) What is the difference between a •statement• and an •expression•?
e) Is a •base• also a •unitary-clause•?

3.2 Bases
a) Is \( ox + yn \) a •base•?
b) How many kinds of •bases• can be distinguished?
c) List all the •bases• in the object \( c(a(1 > b).Qf.c \sin (X) \cos (X + \pi/2))Qf.c. \)
d) Is \( n [3, n] \) a •base•?
e) Is \( na(b)n \) a •call• or a •slice•?

3.3 Identifiers
a) List the •identifiers• in the object \( ol:ca := char \) of \( file \) + "aS".
b) What is the mode of \( axe \) in \( creal x := 3.14 \).
c) What is the mode of \( on2 \) in \( n2 = m2[3:5, 3:6]Qf.m. \)
d) Do \( cue \) and \( ova \) have the same mode in the •declaration• \( r 1:10 Qf.f2~ u, [1:10]Qf.f2~ v? \)
e) Is \( c$linec \) an •identifier•?

3.4 Slices
In the reach of the •declaration• \( n[1:m, 1:n]real x2, y2n : \)
a) Is \( nx[1][1]n \) a •slice•?
b) Is \( nx2[1]n \) a •slice• and if so what is the mode of its value?
c) Is \( obegin x2 end[1, 1]n \) a •slice•?
d) Is \( n [i] > 0 \) then \( x2 \) else \( y2 \) \( fi[1, 1]m \) a •slice•?
e) Which of the following can be subscripts?
\( u35n, items of an, ni + n * 2n, ni := 2n, ni + := 2n. \)

3.5 Multiple values
In the reach of the •declaration• \( n[1:m, 1:n]real x2, [1:3]int \)
u1 = (1, 2, 3)n:
a) Is \( nu0n \) a •variable•?
b) Is \( nx2[1, 2]n \) a •variable•?
c) Is \( nu1[2] := 2n \) an •assignment•?
d) Is \( nx2[2][1] := 3.14n \) an •assignment•?
e) Is \( nx2[1, 1] := 3.14n \) an •assignment•?

3.6 Trimmers

3.7 Calls
a) Is \( \cos (x := pl/4) \)
b) Is \( orandomQf.m \)
c) Is \( \cos (x > 0) \)
d) Under what condition?
e) Is \( o := (x):Qf.m \)

3.8 Void cast
a) Is a •void-cast-p; \)
b) Is \( o := x \) := \( yo \)
c) Is \( ax := (x):Qf.m \)
d) Is \( u := (x):Qf.m \)
e) Is \( o := (x):Qf.m \)

3.9 Cohesions
a) Is a •cohesion• a •cohesion•?
b) Is a •cohesion• a •cohesion•?
c) Is \( o(a \) of \( b \) of \( c \) of \( d \) of \( e \))Qf.m.
de) Is \( ou := (x):Qf.m \)

3.10 Selections
a) Is a •selection• a •selection•?
b) Is \( ou := na \) in \( na \)
c) Indicate by parentheses the order of elaboration of \( na \) of \( b \) of \( c \) and \( d \)
e) Is \( na := (x)Qf.m \)

3.11 Formulas
a) Is a •formula• a •formula•?
b) What is the value?
c) What is the value?
d) Is \( o4 + \) \( new2 \)
e) What is the value?

3.12 Confrontation
a) Is a •secondary• a •secondary•?
b) Is \( orif[i := i+1]Qf.m \)
c) Is \( orealu := \) •con•?
d) Is \( op:= \) •random•?
e) Is \( op := x := y \)
Using the *declaration* given in 3.5 above:

a) what is the value of nu[2:]*n?  
b) what can be said about the *formula*  
\[ nx2[2:3][2,1] = x2[2,1]*n? 

c) what is the value of nu[2:30][1]*n?  
d) what is the value of nu[32][3]*n?  
e) is nx2[i:=1; j:=1, 3]*n a *slice*? 

3.7 Calls

a) Is \( \cos(x := \pi/4) \) a *call*?  
b) Is \( \text{random} \) in \( ax := \text{random} \) a *call*?  
c) Is \( \cos(x > 0 \mid x \mid \pi/2) \) n a *call*?  
d) Under what conditions is \( na(b) \) in \( ma(b) := cn \) a *call*?  
e) Under what conditions is \( ma(b)(c) \) a *call*? 

3.8 Void cast packs

a) Is a *void-cast-pack* a *primary*?  
b) Is \( (: x) := yn \) an *assignation*?  
c) Is \( ax := (y) *n \) an *assignation*?  
d) Is \( n(: (x)) *n \) a *void-cast-pack*?  
e) Is \( \text{uproc} \ p := x := 3.14 *n \) a *declaration*? 

3.9 Cohesions

a) Is a *cohesion* a *primary*?  
b) Is a *cohesion* a *tertiary*?  
c) Is \( n(x + y) *n \) a *cohesion*?  
d) Is \( of[1:3] \) \( \text{Ref} \) \( \text{struct} \) \( \text{int} \) a, \( \text{Real} \) b \( *n \) a *cohesion*?  
e) Under what conditions is \( na \) of \( b := cn \) an *assignation*? 

3.10 Selections

a) Is a *selection* a *primary*?  
b) Is the \( \text{man} \) in \( na \) of \( bn \) an *identifier*?  
c) Indicate by parentheses the order of elaboration of \( na \) of \( b[c] *n \) and of \( ne \) of \( g(x) *n \).  
d) Is \( n[a \ \text{of} \ b] \) of \( cn \) a *selection*?  
e) Is \( na \ \text{of} \ (b \ \text{of} \ c) \) a *selection*? 

3.11 Formulas

a) Is a *formula* a *tertiary*?  
b) What is the value of \( \text{u2 elem bin} \ 5n \)?  
c) What is the value of \( \text{ulwb} - 3.14 *n \)  
d) Is \( n4 \ := 2n \) a *formula* and if so what is its value?  
e) What is the value of \( n-(1<2 \text{and} 3>4 \text{or} 5=6) \text{or} \text{true} *n \) ? 

3.12 Confrontations

a) Is a *secondary* a *confrontation*?  
b) Is \( nx[i:=i+1] \) a *slice*?  
c) Is \( \text{ureal} \) a *confrontation*?  
d) Is \( \text{uproc} \ := \text{random} \) a *confrontation*?  
e) Is \( np := x := yn \) an *identity-relation* or an *assignation*?
3.13 Identity relations

In the reach of the •declaration• \( \text{mint} \ i, j; \ \text{ref int} \ ii \ := \ i, jj \ := \ in \ :
\)

a) what is the value of \( ii := jjn \)?
b) what is the value of \( ni := jjn \)?
c) what is the value of \( ni := jo \)?
d) Is \( nx := 3.14n \) an •identity-relation•?
e) Is \( nx := x[2]n \) an •identity-relation•?

3.14 Casts

a) Is a •cast• a •primary•?
b) Is \( \text{nint} : 3.14n \) a •cast•?
c) Is \( ox := yo \) an •assignment• or an •identity-relation•?
d) Is \( o[1:1]\text{real} : 3.14n \) a •cast•?
e) Is \( \text{ref int} : ii := 2n \) an •assignment•?

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 \( utn \) a •constant• or a •variable•?
d) What is the mode of \( nsu \)?
e) How many occurrences of an •identity-relation• are there?

4 Clauses

4.1 Conditional clause

The •conditional• programming concept is present in several forms, depending on the choice in the clauses, either:

- or, using another rule, shown in figure 4.1.

```
| if-symbol conditional |
| serial-clause |
| aif a>b |
```

There are two noteworthy. They have the sense that it \( nifn \) or \( n(c, and ends \( n) \). As a consequence, and is, a •primary• positions which might impose a language distinction is made

- •conditional-statements
- •conditional-clauses
- its value is void

\[ \text{R.6.0.1.b}^{(1)} \]

and genuine syntactic permits •conditional

```
\text{if} \ \text{a} > 0
```

which may be used in

```
\text{mal} := \text{if} \ \text{a} > 0
```

\[ \text{R.6.0.1.b}^{(1)} \]

Note that rules are not present only for the language. The no a •program•.
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

\[
\text{\texttt{if \ a > b \ then \ a \ else \ b \ fi}}
\]

or, using another representation

\[
\text{\texttt{u\ (a \ > \ b \ | \ a \ | \ b)\ n}}
\]

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

![Fig. 4.1.a](image)

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 u, and ends with a *fi-symbol*, represented by fi or n. 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* \([R.6.0.1.b]\) and may deliver a value. There is only one genuine syntactic rule \([R.6.4.1]\). This merging of concepts permits *conditional-clauses* like

\[
\text{\texttt{if \ a \ > \ 0 \ then \ sqrt(a) \ else \ go \ to \ error \ fi}}
\]

which may be used in a situation like

\[
\text{\texttt{mal := if \ a \ > \ 0 \ then \ sqrt(a) \ else \ go \ to \ error \ fi}}
\]

---

(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: \( a( p \mid x \mid y ) = 2.3, \ ( q \mid \cos \mid \sin ) (x), \ ( r \mid x \mid y ) + ( f \mid u \mid v ) n, \) in which we have used, for preference, the shorter representations.

A simplified syntax of the *conditional-clause* is:

\[
\text{conditional clause : if symbol, condition, then clause, else clause, fi symbol.}
\]

\[
\text{condition : serial clause.}
\]

\[
\text{then clause : then symbol, serial clause.}
\]

\[
\text{else clause : else symbol, serial clause.}
\]

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.6.1.l.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:

\[
\text{c( } x > 0 \mid x \mid -x ) n
\]

where the value of the *condition* is that of its last *unit*.

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 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*:

\[
\text{c( x > 0 | x | -x ) n}
\]

has as its value the absolute value of \( nx \).

### 4.2 Simple extensions of the conditional clause

A *conditional-clause* like

\[
\text{ni if a then i else if c then d else e then f else g fi fi fi}
\]

may occur frequently in programming situations. For this reason an extension [R.7.4.b] is available whereby the same *clause* may also be written

\[
\text{ni if a then i else if c then d else e then f else g fi fi fi}
\]

The essence of the extension, if the correct representations, the

\[
\text{m( a | b )}
\]

which may be written

\[
\text{m( a | b )}
\]

in the extended list of counting nips, such as

\[
\text{4.2 in the case when}
\]

\[
\text{false}
\]

possesses the value *condition* and is a

A similar extension if may be reincluded, but this extension

\[
\text{ni if a then i else i}
\]

having the same meaning as

\[
\text{ni if a then i else i}
\]

In other representations means the same as

\[
\text{ni if a then i else i}
\]

where the symbol \( c \) of-if-symbol. It is all but no confusion the elaboration of the *effect of \( m( a | b | c | d | e | f ) n \)*, but the former

\[
\text{ni if a then i else i}
\]

In the strict contains an *else-c

\[
\text{ni if a then i else i}
\]

allows *else skip f i

may be written

In the *assignation* *undefined real val

\[
\text{ni if a then i else i}
\]

man is not positive to possess some unde

### 4.3 Case clauses

A case clause in

\[
\text{ni if a then i else i}
\]

clause*, intended
A similar extension [R.9.4.b] exists, whereby the symbols

\textit{when}\ ifn may be replaced by \textit{when}\ fn if the corresponding \textit{nf}\ fn is elided, but this extension may not be so useful. Because of it, \textit{if} a \textit{then} b \textit{then} c \textit{else} d \textit{fn} has the same meaning as

\textit{if} a \textit{then} b \textit{then} c \textit{else} d \textit{fn}

In other representations we have that

\textit{if} a \textit{then} b \textit{then} c \textit{else} d \textit{fn}

means the same as

\textit{if} (a | b | c | d)\textit{fn},

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

In the \textit{strict}\ language the \textit{conditional-clause} always contains an \textit{else-clause}; however, another extension [R.9.4.a] allows \textit{else}\ \textit{skip}\ \textit{fn} to be replaced by \textit{nf}\ \textit{fn}, so that the clause

\textit{if} p \textit{then} go to l \textit{else}\ \textit{skip}\ \textit{fn}

may be written

\textit{if} p \textit{then} go to l \textit{fn}

In the \textit{assignation}\ \textit{nx} := \{a > 0 \} sqrt(a)\textit{fn} therefore, some undefined real value will be assigned to \textit{nx}, if the value of \textit{a} is not positive. This occurs because the \textit{uskipfn} 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 \textit{conditional-clause}, intended to allow for efficient implementation of a
certain kind of `conditional-clause` which may appear frequently. The `clause`

\[ \begin{cases} 
  \text{if } i = 1 \text{ then } x \text{ else if } i = 2 \text{ then } y \text{ else if } i = 3 \text{ then } z \text{ else } a \text{ fin} 
\end{cases} \]

is frequently. The `clause`

\[ a(i \mid x, y, z \mid a) \]

or in another representation,

\[ n(i \mid x, y, z \mid a) \]

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

![Flow of control diagram](image)

in figure 4.3. Observe that \( n(i \mid x \mid a) \) is not a case clause for case clauses contain at least two `unitary-clauses` between the `Case` and the `any`. 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` \( n(i \mid x) \)
- `then-symbol` \( n(i \mid x) \)
- `else-symbol` \( n(i \mid x) \)
- `fi-symbol` \( n(i \mid x) \)

This means that the case clause given above might be written

\[ \text{case } i \text{ in } x, y, z \text{ out a esacu} \]

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

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

\[ \text{while } i \text{ to } n \text{ do } sn \]
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

\[
\text{ufo to } 10 \text{ do randomn}~
\]

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

\[
\text{begin int } j := 1; \quad \text{m: if } j \leq 10 \text{ then random; } j := j + 1; \quad \text{go to a fi} \quad \text{end}~
\]

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

\[
\text{ufo } i \text{ from } a \text{ by } b \text{ to } c \text{ do } x[i] := \text{sqrt}(i) \quad \text{n}
\]

This is defined to be the equivalent of

\[
\text{begin int } j := a, \text{ int } k = b, \text{ l} = c; \quad \text{m: if } (k > 0 \mid j \leq 1 \mid k < 0 \mid j > 1 \mid \text{true}) \quad \text{then int } i = j; \text{ x[i] := sqrt}(i); \text{ j := k; \quad go to a fi} \quad \text{end}~
\]

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:

\[
\text{ufo } i \text{ from } a \text{ by } b \text{ to } c \text{ do } x[i] := \text{sqrt}(i) \quad \text{n}
\]

and

\[
\text{ufo } i \text{ from } a \text{ by } b \text{ while } d \text{ do } e~
\]

These differ in that the first form contains a ufo and the second does not. In both forms ufo in or pby lno or pwhile true may be elided \([R.9.3.c]\) (the statement of this extension is more precise in the Report) and if the *identifier* min does not appear in the *unitary-clause* men, or the *serial-clause* ndn, then ufo in may be elided. Notice that the control *variable* (ojo 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 ufo in means that min is, for each elaboration of ndn and men, an *integral-constant* declared within a range which contains both ndn and men. Consequently no assignment may be made to min. This fact was used in the examples given above.

Before leaving repetitive statements, we should observe that the *unitary-clauses* men, bno and ndn are elaborated collaterally \([R.6.2.2.a]\) and once only, which means, in particular, that a change in the step size bno or in the upper bound ndn, after the initial elaboration, will not affect the further elaboration of the repetitive statement.
4.5 Closed clauses

Some examples of closed-clauses are \( n(x + y)n \), \( n((a))n \) and \texttt{begin \texttt{read}(x, y); print(x + y) \texttt{end}. \) Note that either \( n()n \) pairs (*packs*) or \texttt{begin} \texttt{end} pairs (*packages*) may be used, but that \( n(x + y)n \) is not a closed-clause [R.6.1.1.e, 1.2.5.i, 3.0.1.h,i]. A simplified syntax of the closed-clause is

\[
\text{closed clause : open symbol, serial clause, close symbol ; begin symbol, serial clause, end symbol.}
\]

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

\[
\text{closed clause} = \text{open symbol, serial clause, close symbol}.
\]

4.6 Collateral phrases

A collateral-clause [R.6.2.1.b,c,d,f] consists of two or more unitary-clauses [R.6.1.1.e] separated by comma-symbols and enclosed between a \( n()n \) pair (*pack*) or \texttt{begin \texttt{end}} pair (*package*). An example of a collateral-clause is \( n(1, 2, 3, 4)n \). It may be used in the situations of \texttt{scanf} \texttt{scanf} x1 = \( (1, 2, 3, 4) \) or \( \texttt{scanf} \texttt{scanf} x2 = \( (1, 2) \) \texttt{scanf} \texttt{scanf} z = \( (1, 2, 3, 4) \) \texttt{scanf} \texttt{scanf}. 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 \( n()n \) is not a collateral-clause, for, otherwise, there would be an ambiguity in that \( n()n \) is already a closed-clause.

\[ \]

\[ \text{1) Strictly speaking, "pack" and "package" are prototypical but not prototypical [R.1.1.6], so you will not find them used in the semantic text of the Report.} \]
A simplified syntax of the \texttt{collateral-clause} is
\begin{verbatim}
  collateral clause :
  open symbol, unit list proper, close symbol ;
  unit list proper :
  unitary clause, comma symbol, unitary clause ;
  unit list proper, comma symbol, unitary clause.
\end{verbatim}

but the strict syntax is rather more complicated \cite{R.6.2.1} since
it must take care of the two situations hinted at above together
with the balancing of modes \cite{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 \texttt{collateral-clause} is shown in figure 4.6. If a
\texttt{collateral-clause} is used as a \texttt{statement}, then it may be
preceded by a \texttt{parallel-symbol}, represented by \texttt{uparm}, if
parallel processing is intended \cite{R.10.4}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4_6.png}
\caption{Fig.4.6}
\end{figure}

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

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

\[ \text{ux} := 3n \]

is a *unit*, but for us,

\[ \text{nx} := 3n \]

and

\[ \text{m}12: \text{x} := 3n \]

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

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

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*:

\[ \text{ux} := 3n \]

\[ \text{m}12: x := 3n \]

\[ \text{m}11: y := 2 ; x := 3n \]

\[ \text{opn}() \]

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 \( \text{r} \), followed by a *label*. The following are therefore examples of a *suite-of-clause-trains*:

\[ \text{nx} := 3n \]

\[ \text{m}11: y := 2 ; x := 3n \]

\[ \text{a}() \]

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

- suite of clause trains : clause train ;
- suite of clause trains, completer, clause train.

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

- suite of clause trains : clause train ;
- suite of clause trains, completer, clause train.

The elaboration of the *suite-of-clause-trains* is dramatically different. The effect of the *completer*, as opposed to the *go-on-symbols*, 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 \( m \) is \(-1\), then the value of the *serial-clause* is the value of \( n \) \( = \) \( 1 \) and the *clause-train* \( y := 2 ; x := 3n \) is not elaborated; otherwise, it is the value of \( nx := 3n \). In fact, the effect is the same as that of \( m(i > 0 \mid y := 2 ; x := 3 \mid x := 1) \) \( n \). 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

```plaintext
for k to upper s do ( c = s[k] \( \mid \; i := k \) : \( \text{false} : \) \( \text{true} \) [R.10.5.1.2.2], 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

```plaintext
nx := 3n
```  
```plaintext
ull: y := 2 ; x := 3n
```  
```plaintext
u(r > .5 | 11 | x := 1) . 11: y := 2 ; x := 3n
```  
```plaintext
real x, y ; ( r > .5 | 11 | x := 1) . 11: y := 2 ; x := 3n
```  
```plaintext
ur := random ; real x, y ;
```  
```plaintext
( r < .5 | 11 | x := 1) . 11: y := 2 ; x := 3n
```  
```plaintext
and
```  
```plaintext
real r ; r := random ; real x, y ;
```  
```plaintext
( r < .5 | 11 | x := 1) . 11: y := 2 ; x := 3n
```  
```plaintext
```

A simplified syntax of *serial-clause* is:

```plaintext
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.
```  
```plaintext
declaration prelude : single declaration, go on symbol;
statement prelude, single declaration, go on symbol.
```  
```plaintext
single declaration : unitary declaration ; collateral declaration.
statement prelude : unit, go on symbol;
statement prelude, unit, go on symbol.
```  
```plaintext
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

\[ a[0]*x^n+a[1]*x^{n-1} + \ldots + a[n] \].

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 main. The number of linear factors is in \( mn \), any constant factor is in \( m \) and the factors \( a[i]*x-v[i] \) are found in the row of integral values \( m \) and \( n \) (1).

```algol68
proc factors = (ref a: int n : for the number of rational linear factors, ref int r : for the constant factors, ref int u, v : for the linear factors (w[i]*x-v[i]), 1 sis[x] 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 ; v[r] := 0 ; n := n - 1 end ; for p to abs a[0] do if a[0] + p = 0 then q := 0 ; while (q := abs q + 1) <= a[n] do if a[n] + q = 0 then g := 0 ; while (g := abs g + 1) <= a[n] do if a[n] + g = 0 then if p divides a[0] then g := 0 ; while (g := abs g + 1) <= a[n] do if a[n] + g = 0 then if q * 1 and p = 1 then look for a constant factor: MORE : for i from 0 to n do if a[i] + q = 0 then if q does not divide a[i] then go to NOCONSTANT fi := :, which replaces the common power of x: for j from 0 to n do a[j] := q ; c := q ; g := a[j] ; end ; if c divides a[j] then go to MORE fi := send the search for a constant factor: NOCONSTANT := : try (p*x-q) as a linear factor: g := 1 ; f := a[0] ; try x = q / p g : for i to n do f := f * g + a[i] * (q := f) ; if f = 0 then (p*x-q) is a factor: u[r] := 1 ; v[r] := q ; n := n - 1 ; end ; for i from 0 to n do compute the residue: 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) g : ( 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.

4.1 Conditional clauses

a) What is the value of
b) Is \( x < 0 \) then \( g \)
c) Is \( x > 0 \) then \( a \)
d) Is \( x > 0 \) then \( b \)
e) Is \( x < 0 \) then \( c \)

4.2 Simple extension

a) What is the value of
b) What is the value of

c) What is the value of

d) Simplify the following:

```algol68
if p then a else p
```
e) Remove the extension

4.3 Case clauses

a) Is \( x \in \{ 1, 2, 3 \} \)
b) What are all the \( x \) in

c) What is the value of

d) What is the value of

e) Is \( x \in \{ 1, 2, 3 \} \)

4.4 Repetitive statement
135

10.2d x or y of x

routines do not seem to follow semantics for void

routine properly.

83

line 1 - missing

6.7 (c) answer is wrong - flipping and direct. Cannot

go together

6.11 (b)(c) answers are wrong the procedures should

have more refer... to be consistent with

other questions.

89

7.4 + 8

x 11.1 => 11.11 y

x 10.5, 2.2, a, => 10.5, 2, 2, 6, z

p/0.6

8.6, t + 1

x 19.2, a => 19.3, a y

37/45

two diff representations of and are used

p 172 9.26-1 heap real t
An ALGOL 68 Companion

57

\begin{verbatim}
fi send else partz
fi send iteration on gz
fi send iteration on pz;
REDUCED: (n = 0 | c *:= a[0] ; a[0] := 1 ) ;
\end{verbatim}

\begin{verbatim}
the degree of the residual polynomial is\# n
endu

In the range of the •declaration• \texttt{m[0:3]}\texttt{int} \texttt{a1} := \texttt{(f \texttt{int} : (1, -1, 2, -2))[@0], int k, number, constant, [1:3]\texttt{int} \texttt{m1}, n1, a •call• of the above \texttt{procedure• might be}
\begin{verbatim}
ok := factors(a1,number,constant,m1,n1)u
\end{verbatim}
whereupon we should have \texttt{nk} = 2, \texttt{a1} = \texttt{(f \texttt{int} : (1, 0, 2, 0))[@0], number = 1, constant = 1, m1 = (1), n1 = (1), corresponding to the factoring
\begin{verbatim}
ax**3 - ax**2 + 2*ax - 2 = (ax**2 + 2)(ax - 1)u
\end{verbatim}
Observe that in the •clause• \begin{verbatim}begin ref int ai = a[i] ; ai := f := (ai + f * q) + p endu\end{verbatim}, 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• \texttt{ef := f * q + a[i] * (g *:= p)u}, which replaces two statements in the original ALGOL 60 version.

Review questions

4.1 Conditional clauses

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

4.2 Simple extensions of conditional clauses

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

4.3 Case clauses

a) Is \texttt{u(1 | 2 | 3)u a case clause?}
b) What are all the representations of the •if-symbol•?
c) What is the value of \texttt{u(2 | 3, u, 5 | 6)u}?
d) What is the value of \texttt{u(0 | 3, 2, 1 | 2)u}?
e) Is \texttt{u(2 | a, b, c) of on 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 elaborated?

a) $\text{for } i \text{ do } e \text{ while } (i < 9) \text{ do } en$

b) $\text{for } i \text{ to } 10 \text{ by } 2 \text{ do } en$

c) $\text{do } do$

d) $\text{while } false \text{ do } en$

e) $\text{while } false \text{ do } en$

Comment on the scopes of $o$ in the following:

f) $\text{of } i \text{ do } e \text{ while } (i < 9) \text{ do } c$

g) $\text{of } 1 \text{ to } 5 \text{ do } f$

4.5 Closed clauses

a) Is $\text{if } x \text{/ y } \text{ do } c$ a closed-clause?

b) Is $\text{if } (x \text{/ } y ) \text{ do } c$ a closed-clause?

c) Is $\text{if } x \text{ and } y \text{ do } c$ a closed-clause?

d) Is $\text{if } x \text{ and } y \text{ do } c$ a closed-clause?

e) Is $\text{if } x : = 1 \text{ and } y : = 2 \text{ do } c$ a closed-clause?

4.6 Collateral phrases

a) Is $\text{if } x \text{ do } c$ a collateral-clause?

b) Is $\text{if } 1 \text{ or } 2 \text{ do } c$ a collateral-clause?

c) Is $\text{if } 1 \text{ and } 2 \text{ do } c$ a collateral-clause?

d) Is $\text{if } x \text{ do } c$ a collateral-clause?

e) Is $\text{if } x \text{ do } c$ a collateral-clause?

4.7 Serial clauses

a) Is $\text{if } x \text{ do } c$ a serial-clause?

b) Is $\text{if } x \text{ do } c$ a serial-clause?

c) Is $\text{if } x \text{ do } c$ a serial-clause?

d) Is $\text{if } x \text{ do } c$ a serial-clause?

e) Is $\text{if } x \text{ do } c$ a serial-clause?

4.8 Program example

a) How many occurrences of a conditional-clause are there in this particular-program?

b) What is the mode of $\text{max}$?

c) What is the mode of $\text{min}$?

d) How many occurrences of a closed-clause are there following the label $\text{NOCONSTANT}$?

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* `oupn` which has two *parameters* `man` and `nbn`, and whose effect is to increment the *real-variable* `man` by the *real-constant* `nbn`. In ALGOL 68 the defining occurrence of such a *procedure* is in the *identity-declaration*:

```
up = (ref real a, real b) : a ::= b
```

and its *call* might be `oup(x, 2)` or `oup(x[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(x[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,2.0)` or `CALL UP(X1(I), Y)`.

In FORTH 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* `oup(x, 2)`, `x` is passed by name to `man` and `2` is passed by value to `nbn`. 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

\[ o((\text{ref } \text{real } a, \text{real } b)) : (a +:= b) \]  

is an example of a \textit{routine-denotation} [R.5.4.1.a] and is essentially what stands on the right of the \textit{equals-symbol} in the \textit{declaration} of \texttt{up} given in section 5.1 above. One may notice that the enclosing symbols \texttt{o} and \texttt{c} 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 \textit{routine-denotation}, like any other \textit{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 \textit{denotation}. For example, the routine possessed by

\[ o((\text{ref } \text{real } a, \text{real } b)) : (a +:= b) \]

is

\[ *(\text{ref } \text{real } a = \text{skip}, \text{real } b = \text{skip}; (a +:= b)) \]  

and it is important to notice that it has the shape of a \textit{closed-clause}, in which each of the \textit{parameters} \texttt{ma} and \texttt{nb} forms part of an \textit{identity-declaration}.

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

\[ \text{up}(x, y) : \text{skip} \]

the \textit{identifier} \texttt{up} is made to possess the routine

\[ *(\text{ref } \text{real } a = \text{skip}, \text{real } b = \text{skip}; (a +:= b)) \]  

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

\[ \text{up}(x, y) : \text{skip} \]

in figure 5.3.a. During the elaboration, \texttt{up} possesses the same \textit{denotation} as \texttt{up} (see the \texttt{formula} \texttt{ma +:= bn} has \texttt{t1 = 2}). Both \texttt{ma} and \texttt{nb} have \textit{reference-to}, a required \textit{operator} \texttt{+:=} [R.10].
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 x[i] = x[i]n might usefully be used to optimize address calculation while working with the vector \*nx1o. An example might be
\[\text{x1[i]} := 3 \times \text{x1[i]} + 2 \times \text{x1[i]} \times 2n\]
rather than
\[\text{nx1[i]} := 3 \times \text{x1[i]} + 2 \times \text{x1[i]} \times 2n\]

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)n is
\[\text{ref real a = x, real b = 2 ; } (a : = b) n\]
and the elaboration of the \*collateral-declaration* which follows its \*open-symbol* results in the relationships depicted in figure 5.3.a. During the elaboration of the \*call* \*up\((x, 2)n, \text{mao possesses the same name as that possessed by } \text{nxn (see figure 5.3.a at 1), and } \text{chn possesses the same value as that}
possessed by \text{nxn (see the figure at 2). This means that the}
\*formula* \(a : = b\)n has the same effect as if it were written \(nxn +:= 2n\). Both \text{mao and } \text{nxn have a mode which begins with}
\*reference-to*, a requirement of the left \*operator* of the
\*operator* \(n +:= \)n [R.10.2.11.e]. Note also that if the \*call* were \*up\((x, y)n, then the \*closed-clause* would contain the
\*declaration* \*real b = yn and this would involve a
dereferencing of \text{yn, depicted in figure 5.3.b at 1. Observe, in
this figure, that _a_ , 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 _a_, e.g., if an assignment is to be made to _a_, then the •formal-parameter• should be _a_ as in:

<table>
<thead>
<tr>
<th>formal-real-parameter equals-symbol actual-real-parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal-real- real-mode-declarer identifier</td>
</tr>
<tr>
<td>real b =</td>
</tr>
</tbody>
</table>

Fig. 5.1. b

but if _a_ is used only as a •constant• of mode _a_, then the •formal-parameter• may be _a_ as:

```
c(real a, real b) : a := 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 _o_ and _c_. Thus in

```
o((ref real a, real b) : a := b) o
```

the object _o_ is the •formal-parameters pack• and _o_ 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, gamma, formal parameter.
gamma : go on symbol, comma symbol.
```

but the strict syntax [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
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

\[ n((\text{real } x)) \text{real : random } \ast x) \]

where the second occurrence of *realu* (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-declarator• is empty). Note that

\[
\text{real} : \text{random} * 100
\]

is not a •routine-denotation• despite the fact that it may appear in the •declaration•. However, the coercion known as "proceduring" [R.8.2.3.1.a] enables the identifier nr100 to possess the routine

\[
(\text{real} : \text{random} * 100)
\]

Actually, it is only necessary to write

\[
\text{uproc usual} \text{real} = \text{real} : \text{random} * 100
\]

and then the routine possessed by nr100 will be

\[
(\text{usual} \text{real} : \text{random} * 100)
\]

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

\[
\begin{align*}
\textproc{usual}\text{a}\text{b} : \text{value} \text{a, b} : \\
\text{begin} \text{i} := \text{i + 1}; \text{a} := \text{a + b end}
\end{align*}
\]

and to explain that, in the scope of the fragments \text{real} \text{x1}[1:10]; \text{integer} \text{i} ; \text{i} := \text{i + 1}, the procedure call \text{usual}\text{a}\text{b}(\text{x1}[1], 2)\text{will, to the astonishment of most, increment the value of x1 [2]m rather than that of x1 [1]m. 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 60 a routine which achieves a similar effect, for simple \text{variables} (not \text{slices}) passed to name, is

\[
\begin{align*}
\textproc{usual} = (\text{ref real} \text{a, real} \text{b}) : [\text{i} := \text{i + 1}; \text{a} := \text{a + b end}
\end{align*}
\]

but the \text{call}\text{usual}\text{a}\text{b}(\text{x1}[1], 2)\text{in the range of x1[1:10]}\text{real} x1 : \text{integer} \text{i} ; \text{i} := \text{i + 1}, will increment the value referred to by x1[1]m and not x1[2]m. Thus the passing of the \text{parameter} \text{x1}[1]m by name, as it was known in ALGOL 60, is not achieved, in ALGOL 60, by using the \text{formal-parameter} \text{ref real a}. The resulting \text{identity-declaration} \text{ref real a = x1[1]}\text{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 \text{variables}, this copy rule of ALGOL 60 amounted to the passing of a \text{procedure} body to the formal \text{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 \text{name part} rather than a \text{value part} [N.5.4.1]. A language should be such that the least effort by the \text{programmer} calls up the simplest implementation \text{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 \text{source program}.

To recapture the strange effect of the call by name of

\[
\text{uproc usual} \text{b} = (\text{uproc usual} \text{a}, \text{b} = \text{0})
\]

for then the first \text{uproc usual} \text{b}(\text{[1], 2})\text{is elaboration of x1[1]m of map in transfer. Thus x1[2]m}

The occurrence of another example of a •routine-denotation• to possess the routine as proceduring [R.8.2.3.1.a] is the example of a •mode-declaration•.

5.6 Program example

The following directed graph \text{gpn} a \text{graph} are represented of a \text{row-of-bits} \text{gpn}. A set \text{of nodes} \text{structure, the ji which is} •true• if the

The set of nodes

\[
\begin{align*}
1 & \rightarrow 2, \quad 1 & \rightarrow 3, \quad 2 & \rightarrow 1, \quad 2 & \rightarrow 3, \quad 3 & \rightarrow 1, \quad 3 & \rightarrow 2
\end{align*}
\]

Since the standard be larger than the mask out the redundant

If the number of •node-declaration• for accordingly. If suffi may use the •mode-row• the operations involved

As an example, for 7 (2,3,4), the algorithm generates

\[
(1), \quad 2(1), \quad 3(1), \quad 2(1), \quad 3(4)
\]

(1) Translated from Algol 60.
ALGOL 60, the example mentioned above should appear as

```
upb = (proc ref real a, real b : (i := 1; a := b)n ,
        for then the first •declaration• arising from the •call•
        upb(x1[i], 2)n is uproc ref real a = x1[i]n. In this case the
        elaboration of x1[i]n occurs at the time of the deproceduring
        [R.8.2.2] of max in qa := bn, and not at the time of parameter
```

The occurrence of x1[i]n in uproc ref real a = x1[i]n is
another example of a •procedured-coercend• for x1[i]n is not a
•routine-denotation•. Nevertheless, the •identifier• max 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 ngn (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 ngn. 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 sgn.
The edges in a family of saplings are recorded in ngn, which,
like ngn, is of mode •row-of-bits•. The boundary of sgn is the
set ngn of nodes neighbouring the nodes of sgn. Initially sgn
contains only node i and ngn its neighbours, i.e., ngn[1]n. The
recursive routine ngn row iterates over the nodes in ngn. For
each node i in ngn it finds all possible edges (new growth)
from sgn to node i. This new growth is recorded in ngn and
removed from sgn. The node i is removed from the boundary ngn.
The procedure ngn row 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 nbits width is 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 nbits width, then the
•node-declaration• for ngn, 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)
```

Obegin mode b = bits for long bits, if necessary:
proc trees = ([1:] b g the given graph,
proc([1:] b) f the action for each family):
begin int n = up b g the number of nodes in the graph;
[1:n] b a the growing family, saplings;
[1:2] 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,
[1:n] b s the nodes of the saplings,
proc b b boundary of the saplings):
if s > mask
then 'the family is complete, so f(a)
done 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 growths;
g[i] := g[i] and ~ s remove the new growths;
grow(loc [1:n] b) := g pass a copy of the residual;
loc unit s the family now includes node i,
loc b := b or q[i] s 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) := q start with a copy,
unit start with node 1,
loc b := g[1] s 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

5.2 Routine denotation

a) Is ref real xy =
b) What is the *form*
c) If proc possesses
a * b l, what
op(x'1, y)n?
d) What is the value
real : a * a
what
ref[1:n] real al

5.3 More on parameters

a) In the reach of
of op(x, y)

b) In the reach of


e) In the reach of


5.4 Syntax of rot

a) Translate the fol-

5.6 Program exam

a) Is unit in a *const*
b) Why is a ref not

c) Why is an *actual-
call*
d) Why was not not i

e) If one is = a

mask?
b) Is the following an identity-declaration?
\[ \text{PROC (real a) real p = a * a?} \]

c) Give a declaration for a procedure \( \text{mr2} \) which has no parameters and delivers a random real value between 0 and 2.

d) Give a declaration for a procedure \( \text{max} \) with two real parameters which delivers the larger of the two.

e) Give a declaration of a procedure \( \text{recip} \) which accepts a real-variable and replaces it by its reciprocal.

5.2 Routine denotations

a) Is \( \text{ref real x} \) \( \text{y = x * y} \) an identity-declaration?

b) What is the formal-parameter of \( \text{[1:]real x1 := (1, 2, 3)} \)?

c) If \( \text{np} \) possesses the routine \( \text{[real a = skip, real b = skip; a * b]} \)\( \text{what is the called-clause is elaborated by the call np(x + y) o?} \)

d) What is the value possessed by the denotation \( \text{u((real a) real s, a * a) o?} \)

e) What is the value possessed by the denotation \( \text{u((int n, m; ref a:n) real a) real : (n < m | a[i] | a[m]) o?} \)

5.3 More on parameters

In the reach of \( \text{real x := 1.2, y := 3.4} \), what is the value of \( \text{np}(x, y) o? \)

a) In the reach of \( \text{p = (real a, b) := 1, 1} \)?

b) In the reach of \( \text{p = (real a, ref real b) real : (b *:= a ; b) o?} \)

c) In the reach of \( \text{p = (ref real a, b) ref real : (1 > 2 | a | b) o?} \)

d) In the reach of \( \text{p = (ref ref real a, ref real b) real : a := b o?} \)

e) In the reach of \( \text{p = (ref real a, b) real : b[1] - a[1] o?} \)

5.4 Syntax of routine denotations

a) Translate the following into ALGOL 68:
\[ \text{procedure p(a, b); value a; integer a, b; b := b * 2 * a?} \]

5.6 Program example

a) Is \( \text{unit a} \) constant or a variable?

b) Why is a ref not necessary in the formal-parameter of so?

c) Why is an actual-parameter \( \text{cloc} := g[i] \) used in the last call?

d) Why was \( \text{a} \) not initialized?

e) If \( \text{a} \) is \( \text{3} \) and \( \text{obits width} \) is \( \text{8} \), what is the value of \( \text{omask} \)?
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 \( x_2.0O \) is equivalent to the integral value possessed by \( x_2.0O \). Derivation of an equivalent value is usually accomplished automatically, i.e., by no conscious effort of the programmer. An example is

\[ x : = 2 \]

where the value possessed by \( x_2.0O \) 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 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 6.1, 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"

\[ \text{real} : = 2 \]

the a priori mode of \( x_2.0O \) 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 (8.6.2.5.1.a), whose details we will not now unravel.

6.2 Classification of coercions

There are eight different coercions. They are

- "dereferencing", as in \( \text{real} : x \)
- "deproceduring", as in \( \text{real} : \text{random} \)
- "proceduring", as in \( \text{real} : \text{proc real : } \text{random} \)
- "uniting", as in \( \text{real} : \text{union(int, bool)} : \text{true} \)
- "widening", as in \( \text{real} : 2 \)
- "rowing", as in \( \text{string} : "a" \)
- "hipping", as in \( \text{real} : \text{skip} \)
and "voiding", as in the "void-cast-pack" of: \( \text{p} \)

These coercions are classified into subsets as follows:

dereferencing and dereproceduring as "adjusting"; and all readers will find the metanotions (R.1.2.3) figure 6.2. Some of this in useful programs.?

\[ \text{strong}......\text{ADAPT} \]
\[ \text{firm}......\text{ADJUST} \]
\[ \text{weak}......\text{FIT} \]
\[ \text{soft}......\text{deprocede} \]

6.3 Fitting

The result of dereferencing is that it refers to. This is called "strongly-declaring", as in

\[ \text{strongly-declaring} \]

There are eight different coercions. They are

- "dereferencing", as in \( \text{real} : x \)
- "deproceduring", as in \( \text{real} : \text{random} \)
- "proceduring", as in \( \text{real} : \text{proc real : } \text{random} \)
- "uniting", as in \( \text{real} : \text{union(int, bool)} : \text{true} \)
- "widening", as in \( \text{real} : 2 \)
- "rowing", as in \( \text{string} : "a" \)
- "hipping", as in \( \text{real} : \text{skip} \)
and "voiding", as in the "void-cast-pack" of: \( \text{p} \)

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 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.

```
    strong-real-unit.............. (2)
       strong-real-base
          strongly-dereferenced-to-real-base (3)
             reference-to-real-base
                reference-to-real-mode-identifier
                   \(x\)
                      \(x\) (1)
                          \(x\) \(\rightarrow\) \(x\)

Fig.6.3
```
The result of deproceduring is the elaboration of a routine (without parameters), e.g., the \texttt{cast} \texttt{real} : \texttt{random} forces the elaboration of the routine possessed by \texttt{random} and delivers the next random real value as the value of the \texttt{cast}. Both dereferencing and deproceduring are classified together as "fitting" [ibid. b), and are the two coercions which occur most frequently.

6.4 Adjusting

Both deproceduring and unifying, 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 deproceduring is a routine. For example, the value possessed by the \texttt{cast} \texttt{oproc real} : \texttt{xWij} is the routine \texttt{real} : \texttt{xWij}. 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 \texttt{parameters}, there are no \texttt{routine-denotations}. The deproceduring coercion makes them unnecessary.

Deproceduring only has a syntactic effect. In the terms of the report, the elaboration of a united \texttt{coercend} is the same as that of its pre-elaboration [ibid. a]. This means that no change of value is involved. Actually, an implementation will find it necessary, upon unifying, 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 unnecessarily.

Uniting has only a syntactic effect. In the terms of the report, the elaboration of a united \texttt{coercend} is the same as that of its pre-elaboration [ibid. a]. This means that no change of value is involved. Actually, an implementation will find it necessary, upon unifying, 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 unnecessarily.

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

\[ a (r e a l : 2) = 20m \]
\[ a (c o m p i : 2) = 20 \text{ i } 0m \]

One may also widen from bits to \texttt{row of boolean} [ibid. c] and from \texttt{bytes} to \texttt{row of character} [ibid. d]. If \texttt{bits width} is \texttt{m}, then \texttt{m real} : \texttt{0} has a value which is that of \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}. Similarly, if \texttt{bytes width} is \texttt{m}, then

\begin{verbatim}
true false 10
\end{verbatim}
on of a routine
random forces
unrandom and
of the “cast”,
ied together as
which occur in

be terms of the
is the same as
means that no
statement will
to the value some
later, especially
the particular
of concern to the
probably occurs
the subject
shall postpone to

The effect of rowing is to deliver a multiple value which
is a row of zero or one elements. It occurs, for example, in
\texttt{m \{real : x\} : and in \{ \texttt{int : 2n. The value in the first case is a
row of zero elements, each of mode \texttt{real}. In the second case
one obtains a row of one element of mode \texttt{integral}. Note that
\texttt{m \{ \texttt{int} : \lbrack \texttt{int} : \rbrack : 2n} involves two consecutive rowings which
result in a one by one matrix. The same effect can be obtained
by \texttt{m \{ \texttt{int} : 2n, since rowing is recursive [R.8.2.6.1.a]. The
“cast” of \texttt{bool} \colon \texttt{n} 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 \texttt{REFETY} in the rule
for rowing [R.8.2.6.1.a]. Thus, \texttt{real : x} will have the
effect of creating a new multiple value whose only element is
\texttt{x} and the “identity-relation” \texttt{m \{ (ref \texttt{real} : x) \lbrack \rbrack \colon z[1] : = \texttt{x}},
possesses the value \texttt{true} no matter what value is referred to
by \texttt{zx}. Of course, it is arranged [R.8.2.6.1.b] that an empty
cannot be rowed to a “variable”, i.e., \texttt{m \{ (ref \texttt{real} : n} is
syntactically invalid.

The coercion known as hopping takes care of the “skip”,
“skips”, the “nil” \texttt{nil}, and “jumps” like a 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
“nil” may be coerced only to a mode which begins with
“reference-type”. The elaboration of a “skip” delivers some
(undefined) value of the required mode, e.g., the value of \texttt{ref
: skips some real value. The value of a “nil”, represented
by \texttt{nil}, is a unique name which refers to no value. This
means that \texttt{m \{ (ref \texttt{real} : nil) \lbrack \rbrack \colon z[1] : = \texttt{true}, although
m \{ (ref \texttt{real} : \texttt{skip} \lbrack \rbrack : = \texttt{(ref \texttt{real} : \texttt{skip})\texttt{n} is unlikely to be\texttt{nil}}. Observe that \texttt{m \{ (ref \texttt{int} : nil \lbrack \rbrack : = \texttt{(ref \texttt{real} : nil)\texttt{n} is not an
“identity-relation” because the modes of its “tertiaries” do not
agree. Also, \texttt{m \{ (ref \texttt{real} : \texttt{ref ref real} : nil} cannot be
elaborated, since no dereferencing can be done on a “nil”
[R.8.2.1.2 Step 2]. The elaboration of a coerced “jump” is a
“jump” except in a case like \texttt{m \{ (proc \texttt{void} : go to l) \lbrack \rbrack : go to l}, where the
value delivered is a routine and the “jump” itself is not
performed [R.8.2.7.2.b]. Note however that \texttt{m \{ (ref \texttt{proc \texttt{void} : go to l})\texttt{n 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.
\( n(2) \) will not deliver the value \( 2 \). The \texttt{void-cast-pack} \( n(\text{random}) \) delivers neither a routine nor a real value, but causes \texttt{random} to be elaborated (deprocedured) once, whereupon the real value delivered is discarded (see \texttt{NONPROC} \[R.8.2.8.1.b\]). This may indeed be just what the programmer desires. In the reach of \texttt{nproc real p := random}, the \texttt{n} in \( n(\text{p}) \) is dereferenced, deprocedured and then voided. The \texttt{void-cast-pack} \( a(\text{random}) \) delivers neither a routine nor a real value, but causes \texttt{random} to be elaborated (deprocedured) once, whereupon the real value delivered is discarded (see \texttt{NCNPROC} \[R.8.2.8.1.b\]). This may indeed be just what the programmer desires. In the reach of \( p := \text{random} \), the \texttt{op} in \( a(\text{p}) \) is dereferenced, deprocedured and then voided. The \texttt{declaration} \( c := (\text{p}) \) is delayed until \( c \) is elaborated. He who can correctly perform the syntactic and semantic analysis of \( p := \text{random} \) and \( c := (\text{p}) \); (\text{g}) \; \text{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 \texttt{cast} because its \texttt{unitary-clause} is strong and in this position all coercions can occur; moreover, strong coercion is the main purpose of the \texttt{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 \texttt{actual-parameters*}, e.g., \( \texttt{n2n in real x := 2n} \), \texttt{sources*}, e.g., \( \texttt{n2n in x := 2n} \), \texttt{conditions*}, e.g., \( \texttt{nx=yn in p(\text{v} \neq 1 )n} \) and \texttt{subscripts*}, e.g., \( \texttt{nx in nx[i]n} \). In these positions the a posteriori mode (i.e., the mode after coercion), is dictated by the context. Examples of firm positions are \texttt{operands*}, e.g., \( \texttt{nx in ngs xe} \), and \texttt{primaries*} of \texttt{calls*}, e.g., \( \texttt{mcall in ncos(x)n} \). Examples of weak positions are \texttt{primaries*} of \texttt{slices*}, e.g., \( \texttt{nx in nx[i]n} \) and \texttt{secondaries*} of \texttt{selections*}, e.g., \( \texttt{mcall in most of cells} \). Examples of soft positions are \texttt{destinations*}, e.g., \( \texttt{nx in x := y} \) and \texttt{tertiaries*} of \texttt{identity-relations*}, e.g., \( \texttt{nx in xx := xx} \). Figure 6.6.a shows an \texttt{assignment} in which many of these positions occur.

\[
\begin{array}{cccccccc}
 n(x := re of z | xx | x[i]) := \sin(x + (real : pi)) & \mapsto \\
 S & T & V & T & V & T & F & F & T & S \\
 O & D & O & O & S & S & S & S & S & S \\
 S & S & O & S & S & S & S & S & S & S \\
 (S = \text{strong, F = firm, W = weak, O = soft}) \\
\end{array}
\]

Fig. 6.6.a

It is clear that \texttt{operands*} cannot be strong, for otherwise one could not determine which operation is to be performed in \( n1 + 2n \). Since of real value uncertainty, restricted to making \texttt{operations*} permitted deproceduring \texttt{skips} can positions, \texttt{formula*}.

We may then the result \texttt{selections*} \texttt{variables}. The deproceduring \texttt{dereferencing}, \texttt{final} \texttt{referencing} may wish to \texttt{dereferencing}\texttt{mode} wish to \texttt{dereferencing}.

Finally, \( \texttt{oxm in nx := xx} \); position is known.

Note that strongly coer\texttt{from outside a}

In the above arising from \texttt{is generally what be used}\texttt{particular, the} \texttt{6.6.b, contain are in a strong} \texttt{identities of \texttt{integral}}

\[
\begin{align*}
&\texttt{mof} = \texttt{integer} \\
&\texttt{mode} = \texttt{identifiable} \\
&\texttt{no assignment}
\end{align*}
\]
\[ n_1 + 2n. \] Since both \( \ast \text{operands} \ast \) could be widened, is it addition of real values or addition of integral values? Because of this uncertainty, the coercions involved in \( \ast \text{operands} \ast \) must be restricted to those classed as adjusting. This is achieved by making \( \ast \text{operands} \ast \) firm [R.8.4.1.d,f]. The only coercions permitted for \( \ast \text{operands} \ast \) are therefore dereferencing, deproceduring, proceduring and uniting. In particular, since a \( \ast \text{skip} \ast \) can only be hipped and hipping can only occur in strong positions, we conclude that the object \( \text{nskip} + \text{skip} \) is not a \( \ast \text{formula} \ast \).

We may recall that if a \( \ast \text{variable} \ast \), say \( \text{uxln} \), is sliced, then the result, say \( \text{ux}[i]n \), is a \( \ast \text{variable} \ast \). Similarly the \( \ast \text{selection} \ast \) \( \text{next of celln} \) from the \( \ast \text{variable} \ast \) \( \text{ncelln} \) is also a \( \ast \text{variable} \ast \). 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 \( \ast \text{reference-to} \ast \) from the a priori mode. Remember that we may wish to write \( \text{ux}[1] := 3.14 \) or \( \text{next of cell} := \text{celln} \) and that the mode of a \( \ast \text{destination} \ast \) must begin with \( \ast \text{reference-to} \ast \). Such a position is known as weak. It involves only those coercions known as fitting, with the special proviso concerning dereferencing.

Finally, in the \( \ast \text{destination} \ast \) of an \( \ast \text{assignation} \ast \), e.g., \( \text{uxn in nx} := \text{yn} \), 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 \( \text{ma, b, c, de and ceto} \), all of which are in a strong position. Note that \( \text{min} \) is the \( \ast \text{identifier} \ast \) of an \( \ast \text{identity-declaration} \ast \) and is therefore not coerced. Its mode is \( \ast \text{integral} \ast \) (not \( \ast \text{reference-to-integral} \ast \)) and therefore

\[
\text{strong-unitary-void-clause}
\]

\[
\text{ifor i from a by b to c while d do en}
\]

\[
\text{integral-mode integral-clause serial-unitary-boolean-void-clause}
\]

Fig.6.6.b

no assignment may be made to it. Moreover, the value of this \( \text{min} \)
is unavailable outside of the *clauses* and men, 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 | x[i] | y[i]) in, the *conditional-clause* n | i < 9 | x[i] | y[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 min is 2, then the routine possessed by s0n is s[ref real : x[i]](). It is therefore the *base* x[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., x[i], a *cohesion*, e.g., max of cell[i], a *formula*, e.g., nabs x or a *confrontation*, e.g., nx := y. 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, *procedure-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.9.2.3], 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 np = a *procedure-void-node-identifier* and *random* is a *procedure-real-node-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

```
*proceudre-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 have

```
*strongly dereferenced procedure void base.*
```

Thus the mode *procedure-void* apply to deprocedured feed into each sense used above) through this route.

The rules for

```
*strongly procedure-void base ;
```

are strongly defined strongly uni through strongly width strongly row.

Each of these must be not a *void base* the others the *procedured-to-void* *rowed-to-void* le.

By examining [R.8.2.5.1], rowin see that productions base* through any rules for hiping only to *skips* *these*.

This complex random is not an

Note that for

```
*strongly procedure-
```

is *procedure-real*.

6.9 The syntactic

The coercions modes, all contain the Report. A rule requires a knowl their use. The real analysis (parsing) give below a couple in the *cast* or *declaration* mint *reference-to-integ *reference-to-integ
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 *procedure-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.8.2.1.1.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, randomc
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.B.2.7.1) cannot be used since they apply
only to *skips*, *nihils* and *jumps* and randomc is not one of
these. This completes our deduction that aproc **void** p =
randomc is not an *identity-relation*.

Note that for aproc **void** p = (: random)c, the significant
production is

*strongly procedured to procedure void base*:
  *void base*.
[R.8.2.3.1.a]. Also, for aproc real p = random only the empty
coercion is required for randomc 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 min
in the *cast* real : in, where min is in the reach of the
*declaration* min in. The *identifier* min is thus a
*reference-to-integral-mode-identifier* and its a priori mode is
*reference-to-integral*. The *reals* in the *cast* indicates that
the a posteriori mode is \(\text{\texttt{real}}\). 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
\*letter i*{3.0.2.b} ........................................... 10
\*reference to integral mode identifier*{8.6.0.1.a} ........ 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* \*\(\text{\texttt{cic}}\) 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* \(\text{\texttt{bool p, real x, y, real xx, x1, real xx1, n}}\), an example of soft balancing is

\[
\text{\texttt{if symbol condition then n = if p then 3.14 else x fi}}
\]

is shown in figure 6.4.1.d, in which the sort of syntactic position involved is \*reference-to-real*. However, \*real* may be coerced to \*integer* and that of \(\text{\texttt{xx}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1p}}\) may be coerced to \(\text{\texttt{integer}}\). However, \*real* may be coerced to \*integer* and that of \(\text{\texttt{xx}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1p}}\) may be coerced to \(\text{\texttt{integer}}\). However, \*real* may be coerced to \*integer* and that of \(\text{\texttt{xx}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1}}\) may be coerced to \(\text{\texttt{real}}\) and then \(\text{\texttt{xx1p}}\) may be coerced to \(\text{\texttt{integer}}\). However, \*real* may be coerced to \*integer* and that of \(\text{\texttt{xx}}\) may be coerced to \*real* and then \(\text{\texttt{xx1}}\) may be coerced to \*real* and then \(\text{\texttt{xx1p}}\) may be coerced to \*integer*.

---

\(^1\) 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 coercable 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 \( o2 \cdot 3 + ( p \mid \text{skip} ) \text{go to} \ k \downarrow \), which is therefore not a *formula*.

### 6.11 Soft balancing

A simple example of soft balancing is

\[
\text{go}(p \mid xx \mid x) := 3.71n
\]

Examination of this object suggests an *assignment* in which the mode of the *destination*, \( n(p \mid xx \mid x)n \), 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 \( xxn \) is *reference-to-reference-to-real*, whereas that of \( xxn \) is *reference-to-real*. The mode of \( xxn \) may be coerced to the balanced mode by dereferencing (once) and that of \( xxn \) by the empty coercion. If we recall that the only coercion which is relevant in soft positions is deproceduring, then it is clear that \( xxn \) cannot be softly coerced to the balanced mode. One must therefore allow \( xxn \) to be softly coerced and \( xxn \) may then be strongly coerced (dereferenced). A sketch of the parse of the *destination*

\[
\text{reference-to-real-destination} \quad | \\
\text{soft-conditional-reference-to-real-clause} \quad | \\
\text{if-symbol condition} \quad | \\
\text{soft-choice-} \quad | \\
\text{fi-symbol} \quad | \\
\text{reference-to-real-clause} \quad | \\
\text{strong-then-} \quad | \\
\text{soft-else-} \quad | \\
\text{reference-to-real-clause} \quad | \\
\text{reference-to-real-clause} \quad | \\
\text{reference-to-real-clause} \quad | \\
\text{xx} \quad | \\
\text{x} \quad | \\
\]

Fig. 6.11

This 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`:

\[ c(p|1;2;3):=3.14n \]

for in this case `nnn` must be strongly coerced.

Now consider the `assignation`:

\[ c(p|x|y):=3.14n \]

Here either `nnn` or `nnn` may be chosen to be soft. It follows that `c(p|x|y):=3.14n` may be parsed as a `reference-to-real-destination` in two distinct ways, i.e., either the `nnn` or the `nnn` 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

\[ o( p|1|2|3)c \]

Here the `clause` `o( p|1|2|3)c` is the `secondary` of a `selection` and is therefore in a weak position [R.8.5.2.1.a]. The mode of `o1|2o` is `complex`\(^{(1)}\), but that of `o3o` is `integral`. It is clear that the object `o3o` must be widened (twice) to `complex`, but widening cannot occur in a weak position. Thus `o1|2o` must be weakly coerced (the coercion is empty) and `o3o` may then be strongly coerced (widened twice). The balanced mode of `o( p|1|2|3)c` is therefore `complex`. A sketch of the parse of this `secondary` is shown in figure 6.12.

\[ \begin{array}{l}
\text{weak-conditional-complex-clause} \\
\text{weak-complex-secondary} \\
\text{weak-then-} \quad \text{strong-else-} \\
\text{complex-clause} \quad \text{complex-clause} \\
\end{array} \]

\[ o( p|1|2|3)c \]

Fig.6.12

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

for parsing the

...It follows that
reference-to-real-
her the
axe
or the
g. This is one of
in ALGOL 68. The
t the cost of
ince no semantic
the grammar is

\[ n \ (R.8.5.2.1.a) \]
that of \( c_3c \) must be widened
occur in a weak
ed (the coercion is
wedged twice). The
first *complex*. A
down in figure 6.12.

\[ \begin{align*}
\text{conditional-clause}\&
\text{ operand}\&
\text{ formula}\&
\text{ firm-clause}\&
\text{ conditional-clause}
\end{align*} \]

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
\[ n2.3 + ( p | 4.5 | 6 ) \]
for dereferencing is permitted in a firm position and both \( nxxu \)
and \( uxxu \) may be firmly coerced to *real* by dereferencing.

6.14 Strong balancing

A simple example of a strong balance is
\[ ny := ( p | x | 1 ) \]
Here the *conditional-clause*, $o(p \mid x \mid 1)n$, is a *source* and is therefore in a strong position [R.8.3.1.1.e]. Both $nxn$ and $nlo$ must therefore be strongly coerced to the balanced mode which is *real*. This means that $nxn$ is dereferenced and $nlo$ 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]
  
- *balance* in a *collateral-clause* [R.6.2.1.e]

- *suite-of-clause-trains* in a *serial-clause* [R.6.1.9]
  
- *identity-relation* [R.9.3.3.1.a]
  
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:

$$n(p \mid 1); q \mid x \mid \tau (3.14 \mid 5) + 2.35n$$

requires a firmly balanced mode of *real* for the left *operand* of the *operator* $n*$. This is achieved by dereferencing and then widening $nlo$, by dereferencing $nxo$, by the empty coercion upon $3.14n$ and by widening $nzo$. Since an *operand* must be firm, either $nxn$ or $3.14n$ 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 $nlo$ and $nzo$ must be strong.

Another example of firm balancing in the extended language is

$$n(1 \mid 1, 3.4, x, rand, xx, skip \mid go\text{ to \texttt{error}}) + 1n$$

in which either $3.14n$ or $nxn$ or $xlo$ or $xro$ or $xxn$ 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.6.2.7.c,o]. Examples, in the reach of $	exttt{m[1:3]real x'1n}$ are

$$\texttt{augb(x, i, 1)n}$$

for firm balancing and

$$\texttt{nx1 := (x, i, 1)n}$$

for strong balancing.
Balancing may occur in a •serial-clause• which contains a •completer•. A trivial example is
\[ \Omega(\{ p \mid 1 \} ; 3.14 \cdot 1 : 1 + 2\pi) \]
Here, if \( \mu \eta \) is •true•, the \( \eta \) \( \mu \) is widened to •real• before the addition is performed (despite the fact that the right •operand• is •integer•), 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
\[ nxx := xx \]
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•
\[ nx := xx \]
the choice must be made in the opposite order. The •identity-relation•
\[ nx := yz \]
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
\[ \nu(\text{ref cell : next of cell}) := \text{nil} \]
in which the \( \text{nil} \) 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 \( \text{go to} \) 110 in the innermost •conditional-clause•. This could perhaps be eliminated by using a •call• of a recursive •procedure• at the •label• 110:n.

```algol
\begin{verbatim}
\textbf{PROC} \text{gcdn} = (\text{ref} \{1\} \text{int} \ a \ \text{the given set of integers\}; \text{ref} \{1\} \text{upb} \ a \ \text{the resulting multipliers\};)
\begin{align*}
\text{end a result \( \text{result} \text{set of integers}\)}
\end{align*}
\begin{align*}
\begin{align*}
\text{PROC} \text{gcdn} = (\text{ref} \{1\} \text{int} \ a \ \text{the given set of integers\}; \text{ref} \{1\} \text{upb} \ a \ \text{the resulting multipliers\};)
\begin{align*}
\text{end a result \( \text{result} \text{set of integers}\)}
\end{align*}
\end{verbatim}
```

\( (1) \) Translated from algorithm 386 by G.H. Bradley, Communications of the Association for Computing Machinery, Vol 13, No 7, 1970.
ref int an = 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 9, y1 := 1, y2 := 0, c2 := abs ai ; k := i ;
17: if ai = 0
then ai := 1 ; z[i] := 0
else l10:
if g := c2 + c1 ; (c2 #: := c1) #: 0 then y1 := g * y1 ; q := c1 + c2 ; (c1 #: := c2) #: 0
then y1 := g * y2 ; go to l10 #eliminate the jump?
else l15: (c1 := c2, y1 := y2)
fii :
120: z[i] := (c1 - y1 * am) + ai ;
ai := y1 ; am := c1 fi ;
l10: skip end ;
8 if k=n, then the following iteration is empty#
l125: 160: for j from k+1 to n do (165: z[j] := 0) ;
l140: for i 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 ;
l150: am
fii
endn

Review questions

6.1 Fundamentals
a) What three things determine the particular coercions?
b) What are the four sorts of syntactic position?
c) Is or~!!: i!!!o : cast? 
d) Is oreal : boolo a *cast? 
e) What coercion occurs in if bool : 101n?

6.2 Classification of coercions
a) How many different coercions are there?
b) What coercions occur in oreal : intn?
c) What coercions are classified as fitting?
d) What coercion occurs in ref real : 3.14n?
e) What coercion occurs in mint : 100 to kn?

6.3 Fitting
a) What coercions occur in oreal : ref ref ref realn?
b) In the reach of oreal : ref ref real xxx, what coercions occur in
eroal : xxx?
c) In the reach of oreal proc int rpin, what coercions occur in
mint : rpin?

d) In the reach of
bool : prbn?
e) What rules are *real-cast? 

6.4 Adjusting
a) What coercions
b) Is uniting a f
(c) What coercions
d) Is oproc *void
(e) Is oproc *void

6.5 Adapting
a) Is hipping an
b) What coercion
(c) What coercions
d) What coercions
(e) What coercions

6.6 Syntactic
a) What coercions
b) Of what sort is
(c) Of what sort is
(d) In the range of
in real : xxx[2] :=
(e) Of what sort is

6.7 Coercends
a) What are the four
b) List all the
(c) Is
(d) Is
(e) Is

6.9 The syntax
a) What rules are
b) Is ocomplex : uni
(c) What rules are
(d) In the range of
void-unit?
e) Is

6.10 Balancing
a) Can the modes be balanced to re
b) Can the modes be
(c) What is the
•reference-to-
algorithm $
\begin{array}{c}
1 \gets i ;
1 +:= c2) \neq 0
\end{array}$
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 `x : = 3.14` ambiguous?

b) In the reach of `nproc ref real` `pxn`, how is `nx` balanced?

c) In the reach of `nproc ref real` `pxn`, how is `nx` 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 `n[1]Real x1n`, how is `nx` balanced?

b) Can the modes `reference-to-real` and `union-of-real-and-integral-mode` be weakly balanced?

c) Is `n1 : real of ( p | 1.2 | 3.4 | 5 )` a `formula`?

d) Is one of `( p | 1 | 2 | 3 | 4 )` syntactically ambiguous?

e) How is `nx` balanced?

6.13 Firm balancing

a) Is `n` 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 `n2 : ( p | 1.2 | 3.14 )` syntactically ambiguous?

e) Is `n value ( p | true | "a" )` a `formula`?

6.15 Positions of balancing


b) Is `n( i | xx, nil, skip | go.to error ) : = xu` an `identity-relation`?

c) Is `n( ( p | 11 ) ; true . 11 : ( i > 0 | 12 ) ; false . 12 : 1 )` a `closed-clause`?

d) How is `n mod ( 1, 2, 3, 4 i 5, 6, x, xx, i )` balanced?

e) Is `n( p | nil | skip ) : = 3.14` an `assignment`?

6.16 Program example

a) Describe the coercions involved in the elaboration of `n(m + := 1) > n.`

b) Describe the elaboration of `nuint c1 := am := abs amn.`

c) What is the purpose of the `declaration` `nref int ai = a[i]n`?
of modes •real•, •reference-to-
ed?

ambiguous?

( p | px | xx )

c( p | px | go_to

• and •reference-to-boolean

softly balanced?

( p | x1 | 2 )[1]n

union-of-real-and-
num?•

tly ambiguous?

ced?

reference-to-integral-
real• be firmly
biguous?

reference-to-procedure-
dare-reference-to-
and •reference-to-

x to an •identity-
2 ); false | 12 :

balanced?

ation?

formation of e :=

= abs amr.

def list ai = a[i]n?

d) Why does a •skip• occur on line 110: skip ends?
e) Can you eliminate the go_to loop 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

\[
\text{union}(\text{int}, \text{bool}), \text{union}(\text{real}, \text{char}), \text{union}(\text{ref} \text{ int}, \text{ref} \text{ real}), \text{union}(\text{a, union}(\text{b, c}), \text{d})\n\]

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*

\[
\text{union}(\text{a, b, c}), \text{union}(\text{a, c, b}), \text{union}(\text{a, union}(\text{c, a}), \text{union}(\text{c, a})\n\]

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

\[
\text{cib} := \text{union}(\text{int, bool})\text{ibu}, \text{whereupon the mode of nib is reference to union of integral and boolean mode}.
\]

An assignment may be made to such a *variable*,

\[
\text{reference-to-union-of-integral-and-boolean-mode-assignation}
\]

![Diagram](image)

Fig 7.2
but the \( *\text{assignaton} * \) \( \text{mb} := \text{true} \) is syntactically possible only because of the uniting coercion to which the \( *\text{base} * \), \( \text{true} \), resulting from its strong position as a \( *\text{source} * \), is subjected (see figure 7.2 at 1). The \( *\text{assignaton} * \) \( \text{mb} := 1n \) is also valid. In both these assignments the internal object assigned does not change under coercion, and the object \( \text{true} \) possesses the same value whether it is considered, a priori, as a \( *\text{base} * \), or, a posteriori, as a \( *\text{source} * \) (see the figure at 2). Note that \( \text{mb} \) possesses a name (see figure at 1), whose mode is \( *\text{reference} * \) to union of integral and boolean mode, but that this name may refer to a value which is either of mode \( *\text{integral} * \) or of mode \( *\text{boolean} * \), since values are not of united mode (i.e., a mode which begins with \( *\text{union of} * \)). Also, the mode of the value referred to by such a \( *\text{variable} * \) as \( \text{mb} \), can be determined, in general, only at the time of elaboration of the \( *\text{program} * \) (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 \( *\text{program} * \) where a run time check of the mode of a value is necessary. Such a check is unnecessary for the \( *\text{assignations} * \) \( \text{mb} := \text{true} \) and \( \text{mb} := 1n \). These checks are known as \( *\text{conformity-relations} * \). Before passing to these we examine two further \( *\text{assignations} * \). 

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

7.3 Conformity relations

\( *\text{Conformity-relations} * \), like \( *\text{assignations} * \), \( *\text{identity-relations} * \) and \( *\text{casts} * \), are \( *\text{confrontations} * \). Examples of \( *\text{conformity-relations} * \) are: \( \text{mi} := \text{ir} \), \( \text{real} := \text{x of qu} \) and \( \text{ua} \) and \( \text{b} := 1 + 2 * \text{x} \). The syntax of \( *\text{conformity-relations} * \) might be written

\text{conformity relation} : \text{tertiary, conformity relator, tertiary.}

\text{conformity relator} :

\text{conforms to and becomes symbol ; conforms to symbol.}

This syntax makes the \( *\text{conformity-relation} * \) 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 \( *\text{tertiary} * \) 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 \( *\text{tertiary} * \) must begin with \( *\text{reference-to} * \). We may recall that the \( *\text{destination} * \) of an \( *\text{assignation} * \), i.e., the \( \text{x} \) in \( \text{mx} := 3.14 \) is soft, so that there is some similarity between \( *\text{assignations} * \) and \( *\text{conformity-} \)
This is intentional, for the elaboration of a *conformity-relation* often results in an assignment. The right *unit* of an *assignment*, e.g., \(3.14n\) in \(nx := 3.14n\), however, is strong. Thus the right *unit* of an *assignment* is strongly coerced but the right *tertiary* of a *conformity-relation* is not coerced.

We may now ask what the difference is between \(nx := 3.14n\) and \(nx := 3.14\). In the case of \(nx := 3.14n\), an assignment is made. In the case of \(nx := 3.14\), an assignment is also made but not before checking that such an assignment is possible. Another difference is that the value of \(nx := 3.14\), after its elaboration, is the same possessed by \(nx\), but the value of \(nx ::= 3.14n\) is a truth value, viz., *true*.

Now consider \(nx := 1n\) and \(nx := 1n\). In the case of \(nx := 1n\) an assignment of the real value, \(1.0e\), is made to \(nx\) after the widening of \(1n\) to a value of mode *real*, but \(nx := 1n\) delivers the value *false* and no assignment takes place. Note that the \(1n\) in \(nx ::= 1n\) 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 \(nx ::= 3.14n\) is *true* and that the value of \(nx ::= 1n\) 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 \(1n\) in \(nx ::= 1n\) is not coerced. Therefore we may ask what will happen with \(ux ::= y0\) and \(ux ::= 1n\). 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 \(y0\) in \(ux ::= y0\) is dereferenced as often as is necessary or possible. Thus \(ux ::= y0\) will deliver *true* and \(ux ::= 1n\) will deliver *false* and in arriving at this, both the \(y0\) and the \(1n\) are dereferenced once.

The only difference between \(3.14n\) and \(3.14\) despite the fact that \(3.14n\) is shown in (it does nothing) is *true*. A skeletal program for \(3.14\) is shown in Fig. 7.3. We see therefore finding out whether united modes, this is known at compile time, that the *conformity-relation* given above are mere fundamentals of the practical programming.

7.4 Conformity and unification

Suppose now that
\[
\text{um} := \text{"a"}; \text{i} := \text{ic}
\]
unifies \(\text{i} : \text{ic} := \text{"a"}\) following the logic determined at compile time. The reader who is linked to the Report [R.11.10.10.5.4.2.b] where relations in actual programs are used.

We wish to write a procedure which will deliver a boolean value and will deliver a boolean value and will deliver a boolean value.

\[
\text{begin int i; if i := a} \text{else c := ...}
\]
In the body of this determines whether *true*

The value of the *conformity-relation* since one knows that *true*; however, if *operator* *abab* is

7.5 Conformity extension

*Conformity-relation* for the convenience of allowing more efficient Examples of these ex...
The only difference between the *conformity-relations* \( \text{nx} ::= 3.14\alpha \) and \( \text{nx} ::= 3.14\alpha \) is that no assignment occurs in \( \text{nx} ::= 3.14\alpha \) despite the fact that the value yielded by \( \text{nx} ::= 3.14\alpha \) is *true*. A skeletal parse of the *conformity-relation* \( \text{nx} ::= 3.14\alpha \) 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*

\[
\text{union} \left( \text{int}, \text{char} \right) \text{ic}. \text{Then the value of the *clause* }\left( \text{int } i; \text{ic} := "a" \right) \text{and the value of the *clause* }\left( \text{int } i; \text{ic} := 7; i :: \text{ic} \right) \text{is *true*}. \text{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] 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]. Thus suppose 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

\[
\text{proc translate} = (\text{union} (\text{int}, \text{char}) a) \text{ union} (\text{int}, \text{char}) : \begin{align*}
\text{begin} & \text{ int } i, \text{ char } c ; \\
\text{ if } i ::= a \text{ then } & \text{ repr } i \# R.10.1.k \\
\text{ else } & c ::= \text{ abs } c \# R.10.1.j \# \text{ fi endn } \end{align*}
\]

In the body of this procedure the *condition*, \( u_i ::= a \), determines whether the value delivered is \#repr\# or \#abs\# \#. The value of the *conformity-relation* \( nu ::= a \# \) 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].
We begin by explaining them in a simple way.

The *conditional-clause*

\[
u( \ a ::= u \mid 1 ; \ b ::= u \mid 2 ; \ c ::= u \mid 3 )\]

In this particular situation— the two enclosing case clauses might be omitted. A case clause must therefore be

\[
u( a, b, c ::= u \in f(a), g(b), h(c) \text{ out error exit} \text{ esac} \]

and its interpretation is the following: if \( u \) conforms to and becomes \( u \), then the value is \( f(a) \); otherwise, if \( f(a) \) conforms to and becomes \( u \), then the value is \( g(b) \); otherwise, if \( g(b) \) conforms to and becomes \( u \), then the value is \( h(c) \); otherwise, if \( h(c) \) conforms to and becomes \( u \), then the value is undefined. Note that if both \( a ::= u \) and \( b ::= u \) possess the value \( \text{true} \), 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.g,ah]. We could perhaps write the procedure of section 7.4 as follows:

```plaintext
case a, b, c ::= u in f(a), g(b), h(c) put error exit esac
```

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 \( u(x, x ::= u \mid 9, 8 \mid \text{error})u \). It is clear that if \( x \) is interpreted as the equivalent of \( u(x ::= u \mid 9, 8 \mid \text{error})u \), then the value \( \text{true} \) can never be delivered. This is fortunate, 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 should therefore 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* \( u[\ast x, x ::= u \ast]u \), in the reach of \( \text{real} \ x, \text{union}(\text{int}, \text{real}) \) \( u \), is equivalent to the following

```plaintext
\[
u(\text{int} i, \text{sema} s = /1; \text{union}(\text{int}, \text{real}) k = u ; \\
\text{par}(x ::= k \mid \text{down} s ; i ::= 1 ; m), \\
( x ::= k \mid \text{down} s ; i ::= 2 ; m) ; 0 \text{.} \text{.} ; i)\]
```

The *declaration* \( \text{union}(\text{int}, \text{real}) k = u \) ensures that the elaboration of \( u \) occurs once only; its value is then held in \( k \). The *declaration* \( \text{sema} s = /1 \), 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 \( \text{true} \). The programmer can perhaps write the procedure of section 7.4 as follows:

```plaintext
optoc \text{translate} = (\text{union}(\text{int}, \text{char}) \text{a}) \text{union}(\text{int}, \text{char}) : \\
\text{begin} \text{int} i, \text{char} c ; \\
\text{case} i, c ::= a \text{ in \text{rept} i, abs c esac} \\
\text{end}
```

7.1 Unite a)

- Is \( \text{union}(\text{relation}) \)?
- Is \( \text{union}(\text{int}) \)?
- What is \( \text{union}(\text{bool}) \)?

7.2 Assign

a) In the reach of \( \text{assignat} \)?
- b) In the reach of \( \text{assignat} \)?
- c) In the reach of \( \text{assignat} \)?
- d) Is \( \text{union}(\text{bool}) \)?
- e) Is \( \text{union}(\text{bool}) \)?

7.3 Conform

a) In the reach of \( \text{assignat} \)?
- b) What is the value \( \text{true} \)?
- c) In the reach of \( \text{assignat} \)?
- d) In the reach of \( \text{assignat} \)?
value $1^n$. The two clauses beginning with $nx := x^n$, are, if
this conformity is successful, followed by the "formula" & which drops the value of the semaphore to $0^n$ 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 $1^n$ or $2^n$ 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 $S(\, S / 1)^n$ rather than
$S/n$ because the operator $n/n$ as a monadic-operator with an
integral right operand could be redefined by the programmer.
The letter $\sigma$ stands for the standard prelude and therefore
returns to the original meaning of $n/n$ as a monadic-operator
which accepts an integer as right operand and delivers an
equivalent semaphore.

Review questions

7.1 United declarers

a) Is $\mathtt{union}((\mathtt{int}, \mathtt{bool}) := \mathtt{union}((\mathtt{bool}, \mathtt{int})$ an identity
relation? 

b) Is $\mathtt{union}((\mathtt{int}, \mathtt{bool}) := \mathtt{bool}$ an assignation? 

c) What is the value of $\mathtt{union}((\mathtt{int}, \mathtt{union}((\mathtt{bool}, \mathtt{char})) := 
\mathtt{union}((\mathtt{bool}, \mathtt{char}, \mathtt{int}))$? 

d) Is $\mathtt{union}((\mathtt{char}, \mathtt{int}) := \mathtt{union}((\mathtt{bool}, \mathtt{int}))$ a declarer? 

e) Is $\mathtt{union}((\mathtt{int}, \mathtt{struct}((\mathtt{int} a)) := \mathtt{struct}((\mathtt{int} a)$ a declarer? 

7.2 Assignations with united declarers

a) In the reach of $\mathtt{union}((\mathtt{char}, \mathtt{bool}) := \mathtt{cbo}$, is $\mathtt{cbo} := \mathtt{true}$ an assignation? 

b) In the reach of $\mathtt{union}((\mathtt{real}, \mathtt{bool}) := \mathtt{rb}$, is $\mathtt{rb} := \mathtt{true}$ an assignation? 

c) In the reach of $\mathtt{union}((\mathtt{real}, \mathtt{bool}) := \mathtt{br}$, what is the mode of
the value referred to by the name possessed by $\mathtt{br}$? 

d) Is $\mathtt{union}((\mathtt{bit}, \mathtt{bytes}) := \mathtt{bit}$ an identity relation? 

e) In the reach of $\mathtt{union}((\mathtt{int}, \mathtt{char}) := \mathtt{ic}$, is $\mathtt{ic} := \mathtt{ic} + \mathtt{1}$ an assignation? 

7.3 Conformity relations

a) In the reach of $\mathtt{union}((\mathtt{real}, \mathtt{char}) := \mathtt{rcn}$, what is the value of
$\mathtt{rcn}$? 

b) What is the value of $\mathtt{nx} := \mathtt{true}$? 

c) In the reach of $\mathtt{mode} \mathtt{br} := \mathtt{union}((\mathtt{bool}, \mathtt{real}); \mathtt{union}((\mathtt{int}, \mathtt{br})$
$\mathtt{ibr}, \mathtt{br} := \mathtt{br}$, what is the value of $\mathtt{nbr} := \mathtt{br}$? 

d) In the reach of $\mathtt{union}((\mathtt{bool}, \mathtt{int}) := \mathtt{bi}$, is $\mathtt{bi} := \mathtt{i} := \mathtt{1}$ an assignation?
e) Is $n \times := x := x \times n$ a *conformity-relation*?

7.4 Conformity and unions

a) In the reach of `union(char, bool) cbn`, is $n \times := cbn$ a *conformity-relation*?

b) In the reach of `union([real, real] rtrn, is $n \times := 3.14$ a *conformity-relation*?

c) Can `union([int, [ref int]]) be contained in a proper *program*?

d) In the reach of `union(int, real) iru, can $n \times := 1n$ 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 $n(x, i, b := 1|3, 4, 5|6 )n$?

b) What is the value of $n(real, real, real := 3.14|7, 8, 9 |10 )n$?

c) Is usea $p = 1n$ a *declaration*?

d) Is `case x, i, b := u in f(x), q(i) out h esacn` a valid ALGOL 68 object?

e) In the reach of `union(char, int, bool) cbn`, is $n \times := skip$ a *conformity-relation*?

f) Is $n \times := go\_to kn$ 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

```
ax + y * zn
```

is elaborated in the order suggested by \( ax + (y * z)n \). 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.
```

Thus, *priority-NINE* has the meaning that one might expect. It is evident that the metanotation, *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 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.
```

Fig.8.1.a
We may now present, in figure B.1.a, a simplified parse of the formula \(e + y \ast ze\), remembering that \(e + o\) is a \(p6\)-operator and \(e \ast o\) is a \(p7\)-operator.

Because a \(p\)\(6\)-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 \(nx + y + zn\) is elaborated as if it were \(n(x + y) + zn\), for the only possible parse is that sketched in figure 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 \(p\)\(6\)-operator \( \text{og}_6 \) has priority 4, \( \text{og}_7 \) has priority 2, and so on, we know that the formula

\[
ch_{\text{og}_7} i \; j \; d_5 \; k \; d_4 \; l \; d_7 \; m \; d_9 \; n_0
\]

must be elaborated in the order suggested by

\[
u(h_{\text{og}_7} i) \; d_2 \; ((j \; d_5 \; k \; d_4 \; l \; d_7 \; m \; d_9 \; n))_0
\]

without any knowledge of the modes of \(ch\), \(i\), \(j\), \(k\), \(l\), \(m\), \(n\), \(o\), and \(p\). The compiler writer appreciates the necessity for this mode independence and the programmer gains because of the resulting clarity in the meaning of the formulas.

### 8.2 Priority declarations

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

\[
\text{priority } + \ast = 6n
\]

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

\[
\text{priority-declaration : priority symbol,}
\]

For example, in the language

\[
\text{priority } + \ast = 6n
\]

and a simple expression

\[
r - (e \ast f) + g
\]

it would be necessary to insert a parenthesis, while in parentheses

\[
(r - e \ast f) + g
\]
A simplified parse of the
expression for the only possible
formula
•p6-operator
is a •p6-operator

The 

simplified parse of the
"p6-operator"

expression for the only possible
formula
•p6-operator
is a •p6-operator


priority NUMBER indication, equals symbol, NUMBER token.

[R.7.3.1.a], where we may observe that the metanotation •NUMBER
[R.1.2.4.f] is used as a counter to ensure that the value of the

priority-declaration

\[
\begin{align*}
\text{priority-symbol p6-indication} & \quad \text{equals-symbol} \\
\text{6-token} & \quad \text{6t}
\end{align*}
\]

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

\[
\begin{align*}
priority \, d1 &= 1, \quad priority \, d2 &= 2, \\
priority \, d3 &= 3, \quad priority \, d4 &= 4
\end{align*}
\]

but all of them might be declared more compactly by using an

extension [R.3.2.c] which allows elision of prioritis, as in

\[
\begin{align*}
priority \, d1 &= 1, \quad d2 &= 2, \\
d3 &= 3, \quad d4 &= 4
\end{align*}
\]

Observe that the programmer may choose his own •dyadic-

indications, like n1d1 and n2d2 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 n1d1 and n2d2 together with

such characters as n?m and n?n, 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 metanotation •PRAM

to convey information about the number of and the modes of the

•parameters

and the metanotation •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

\[
\begin{align*}
\text{op (real, real) real max =} \\
\text{((real a, real b) real : (a > b | a | b))op}
\end{align*}
\]

and a simple parse is shown in figure 8.3. In the extended

language it may be written
\[ \text{Fig. 8.3} \]

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 \( \text{priority max } = \text{7n} \).

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

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

\[ \text{op (string, int) int si = string into} \]

is an *operation-declaration* in which the *actual-parameter* is an *identifier*. The *operator* \( \text{op} \) is then made to possess the same routine as that possessed by \( \text{string into} \) \([R.10.5.2.2.c]\).

In the reach of this *declaration* the *formula* \( \text{op"+123" si 10n} \) will possess the same value as that possessed by the *call* \( \text{string into("+123", 10)} \). Observe that

\[ \text{op si = string into} \]

is not an *operation-declaration* because \( \text{string into} \) is not a *routine-denotation* so the *plan* \( \text{op (string, int)} \) into 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

\[ \text{op interchange } = (\text{Ref Real a, b) : (a := b | Real t = a ; a := b ; b := t)} \]

whose *operator*, \( \text{interchange} \), could be used in the *formula* \( \text{mx interchange yu} \). The same effect would be obtained by means of the *identity-declaration*.
(a :> b | real t = a ; a := b | b := t) n

whose *identifier* could then be used in the *call*
function \( \text{interchange}(x,y) \). 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
\[
\text{nx max y max 0.1n}
\]
is sometimes a more pleasing expression of thought than a
nesting of *calls* like
\[
\text{max(max(x, y), 0.1)n}
\]
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
\[
\text{op max = (real a, b) real : (a > b | a | b) n}
\]
the *operator* \( \text{umax} \) is made to possess the routine
\[
\text{(real a = skip, real b = skip ; real : (a > b | a | b)) n}
\]
This is, of course, already the value possessed by the *routine-
function* 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*
\[
\text{if (priority max = 7 ;}
\]
\[
\text{of max = (real a, b) real : (a > b | a | b) ;}
\]
\[
\text{x := x max y + 3.14)}
\]
there are three occurrences of the object \( \text{umax} \). 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 \( \text{umax} \) as a *dyadic-indication* and its defining
occurrence as an *operator* [R.4.3.1.b, 4.3.2.a]; the third
occurrence of \( \text{umax} \) 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 \( \text{umax} \) represents
two notions, both of which are involved in the identification
process. Since an applied occurrence must always identify a
defining occurrence [R.4.4.1.b], the last occurrence of \( \text{umax} \)
identifies two defining occurrences, 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 \( \text{max} \) and indicate by "\(<===\)" how the identification occurs.

<table>
<thead>
<tr>
<th>priority-declaration</th>
<th>operation-declaration</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dyadic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indication (&lt;===)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{op \ max} )</td>
<td>( \text{fn \ max \ yo} )</td>
<td></td>
</tr>
</tbody>
</table>

**Fig.8.5**

It is thus helpful to remember that an object like \( \text{max} \), 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* \( n.14 + 4.25 \) and \( n.23 + 456 \) 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* \([R.4.2.2.i]\). 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

\[ \text{op \ priority \ + = \ 1n} \]
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

```plaintext
define priority max = 7;
def op max = (real a, b) real : ( a > b | a | b );
def x := 1.23 max y;
def ( priority max = 5 ;
def x := 2.34 max y )
```

the fifth occurrence of \texttt{op 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 \texttt{op 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 coercable to the mode of the first formal-parameter in the operation-declaration and the mode of the right operand must be firmly coercable 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.

```plaintext
define priority o = 8;
def op o = (real a, b) real : 3.14 ;
def ( op o = (real a, int b) real : 3.15 ;
def ( op o = (bool a, b) real : 3.16 ;
def 2.3 o x))
```

The question to be answered here is, which defining occurrence is identified by the operator \texttt{op} in the formula \texttt{n2.3 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 \texttt{op}. This is the first requirement. However, since the mode of the operand \texttt{n2.3} cannot be firmly coerced to \texttt{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 $\text{nx}$ cannot be firmly coerced to 
$\text{integral}$. (Note that it is sufficient to have the failure
occur in only one $\text{operand}$.) We must now search in the next
outer $\text{range}$, which contains yet another $\text{operation-declaration}$, 
in line 2, using the same $\text{dyadic-indication}$.
This time the identification succeeds since the mode of both
$m2.3s$ and $\text{nx}$ can be firmly coerced to $\text{real}$. The value yielded
by the $\text{formula}$ is therefore $3.14$.

8.8 Elaboration of formulas

In section 5.1 we discussed the elaboration of a $\text{call}$. The elaboration of a $\text{formula}$ is similar. As an example,
consider the $\text{clause}$
\begin{verbatim}
1 ( PRIORITY max = 7 ; $\text{#1} 
2 \text{of max} = (\text{real } a, b) \text{ real : } $\text{#2} 
3 ( a > b | a | b ) ; $\text{#3} 
4 x := 3.14 \text{ max } y )
\end{verbatim}

here the $\text{operator} \text{max}$, in line 2, possesses the routine
\begin{verbatim}
* (real a = skip, real b = skip ; real : (a > b | a | b)) *.
\end{verbatim}
The elaboration of the $\text{formula}$, in line 4, then has the
following effect. In a copy of the routine possessed by \text{max},
the two $\text{skips}$ are replaced by the $\text{operands}$ of the $\text{formula}$. The resulting object
\begin{verbatim}
* (real a = 3.14, real b = y ; real : (a > b | a | b)) *
\end{verbatim}
which is a $\text{closed-clause}$, replaces the $\text{formula}$ and is
elaborated. Its value is then the value of the $\text{formula}$. There
is therefore nothing new to tell about the elaboration of
$\text{formulas}$.

Since it seems that each operation in a $\text{formula}$ involves
a sequence of actions like those in the elaboration of a $\text{call}$,
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 $\text{formula}$ like $\text{nx} + \text{yn}$ (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 $\text{monadic-operators}$ is
that they are always of priority ten. There are no $\text{priority-declarations}$ for
$\text{monadic-operators}$. 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 $n-1 ** 2n$ is $n$
and not $-1$, contrary to its meaning in ALGOL 60 and in
\text{FORTRAN}. The reason for making this choice has been explained
earlier in section 3.11.

Because of the syntax
\begin{verbatim}
monadic formula : monadic operator ; monadic operand.
monadic operand : monadic formula ; secondary.
\end{verbatim}
the elaboration of a $\text{formula}$ containing a
sequence of $\text{monadic-operators}$ proceeds from right to left.
Thus the *formula*
\[ \text{unin \ round} \ - \ xn \]
is elaborated in the order suggested by \( \text{unin ( round ( - x ) \ ) un} \). A sketch of the parse of this *formula* is shown in figure 8.9.

![Fig.8.9](image-url)

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 coercable to that of the *formal-parameter*. An example is

\[ \text{unin \ round} \ - \ xn \]

in which the *operator* \( \text{unin} \), in line 3, identifies the *operator* in line 1, since the value possessed by \( \text{unin} \) cannot be firmly coerced to a value of mode *integral*. The value of the *formula* \( \text{unin true} \) is therefore \( \text{100} \).

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 \( \text{unin} \) and \( \text{unin} \). These are related because both can be firmly coerced from the mode specified by \( \text{unin} \). (We shall find it convenient here to shorten the phrase "the mode specified by \( \text{unin} \) to "the mode \( \text{unin} \), or even to "\( \text{unin} \)."") Thus \( \text{unin} \) may be coerced to \( \text{unin} \), by the empty coercion, and to \( \text{unin} \), by dereferencing and then procedureing. One reason for defining this relationship between modes is to exclude some dubious unions from proper *programs* [R4.4.3.d]. Consider, for example, the *declaration*

\[ \text{union (unin, unin) pr := xn} \]

Since \( \text{unin} \) is in a strong position it may be subjected to dereferencing, procedureing and then unifying, whereas the assignment can occur. On the other hand the assignment can also occur with an immediate unifying of \( \text{unin} \). 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:
\[
\begin{align*}
\text{op } m & = (\text{proc } \text{real } \text{int}) : 0 ; \\
\text{op } m & = (\text{ref } \text{real } \text{int}) : 1 ; \\
x & : = 3.14 ; \\
i & : = \pi(x) n.
\end{align*}
\]
What is the value assigned to \(m\)? Is it 0 or 1? Since \(m\) may be firmly coerced both to the mode \(\text{proc } \text{real}\) and to the mode \(\text{ref } \text{real}\), 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:
\[
\begin{align*}
\text{op } + & = (\text{ref } \text{real } a, b) \text{ real} : 0.0 ; \\
\text{op } + & = (\text{ref } \text{real } a, b) \text{ real} : 1.0 ; \\
x & : = (x, y) + (y, x) n.
\end{align*}
\]
In this example the modes of \(\text{real}\) and of \(\text{ref } \text{real}\) 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 Wëssner 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* \(nx, ny, \text{plot}, \text{and moveo}\). Both \(nx\) and \(ny\) are *real-variables*, the two coordinates of the plot pen. The *procedure* \text{plot} first lowers the pen and then plots a straight line from its current position to the position whose coordinates are \(nx(x, y) n\). The *procedure* \text{moveo} first raises the pen and then moves it to the position \(nx(x, y) n\).

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 \(a\). The first approximant \((n = 0)\) is a single point at the centre of the

Case \(n = 1\) to \(\text{length} a d / 2^{\text{first} E}\), then \(a\) by three lines or more step. The process is repeated

\[
\begin{align*}
\text{by three lines of}\ & \\
\text{than} \ & \\
\text{then} & \\
\text{The process is repeated}
\end{align*}
\]

\[
\begin{align*}
\text{case} & \\
\text{then} & \\
\text{than} \ & \\
\text{The process is repeated}
\end{align*}
\]
square. To obtain the second approximant \((n = 1)\), one divides the original square into four squares each of side \(\text{md} / 2\). 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 by three lines of length \(\text{md} / 2 \times 2\) 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 case \(n = 1\) to the four quarters, but scaled down and re-oriented. These four plots are again joined by straight lines of length \(\text{md} / 2 \times 2\) 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

![Diagram with four orientations and truth values]

**(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 \( p, q, n \), the *formula* \( op \ast qn \) plots an approximant with the orientation \( n(p, q) \), and the *formula* \( op \ast qn \) plots a straight line of the required length and with orientation \( n(p, q) \).

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* at "true" *true".

```algol68
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 ;  
plot it (true * true)  
end
```

\(^{(1)}\) From an algorithm of A. van Wijngaarden.

---

8.12 Chinese rings

The next example is the Chinese rings. The puzzle may be extended by having an elongated plate with rings attached, by which this is done in such a way that the rings have been removed, but not the rings. The solution for rings 1 and 2 is done in this way.

In the following, we assume that the rings are all of the same size and rings 1 and 2 are removed. The *formula* that we know how to remove all the rings, then all rings, remove ring 2.

In the following, we assume that the rings are all of the same size and rings 1 and 2 are removed. The *formula* that we know how to remove all the rings, then all rings, remove ring 2.

---

8.1 Formulas

a) \( ax := ay \) \*formula\)
b) \( ax := ay \) \*formula\)
c) What is the order of \( a + y \) - \( a \) ?
d) How many priority e) \( ax := ay \) \*formula\)
f) What is the value of \( ax := ay \) \*formula\)

8.2 Priority dec
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 \( m - 2n \) 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 \( n \) rings, then all \( n \) 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 \( \textbf{formula} \) \texttt{uk down} in removes \( n-k \) in rings. The \( \textbf{formula} \) \texttt{uk up} in replaces \( n-k \) in rings. The \( \textbf{formula} \) \texttt{mn down} on then drives the algorithm by removing all the \( n \) rings.

\begin{verbatim}
begin
  op down = (int a, b) :
    ( int a := a1 ;
      if (a := b) > 0 |
        a down 2 ; print ("remove", a) ; a up 2 ; a down 1 ));
  op up = (int a1, b) :
    ( int a := a1 ;
      if (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
\end{verbatim}

Review questions

8.1 Formulas

a) Is \( ax := yu a \textbf{formula} \)?

b) Is \( nx := yu a \textbf{formula} \)?

c) What is the order of elaboration of \( ax :- yu | \text{abs} i \over 2n \)?

d) How many priority levels are there for \textbf{dyadic-operators}?\textbf{?}

e) Is \( nx := yu a \textbf{formula} \)?

f) What is the value of \( n/3 \)?

8.2 Priority declarations

\footnote{D.O.Shklarsky, N.M.Chentzov, I.M.Yaglom, The USSR Olympiad Problem Book, Freeman & Co. 1962, pp 80-84.}

\footnote{This algorithm is due to Sharon Dyck and in its final form to W.L.Van der Poel.}
8.3 Operation declarations

a) Is \( \text{op} ::= = (\text{ref \ real \ a, b}) : a = \text{bn} \) an \(*\text{operation-declaration}*)?

b) Is \( \text{op} \ t = : = (\text{true}) \text{\ real} \) an \(*\text{operation-declaration}*)?

c) Is \( \text{op} * = : (\text{real \ a}) \text{\ real} : \exp(a)\text{\ real} \) an \(*\text{operation-declaration}*)?

d) Is \( \text{op} = =: (\text{ref \ real \ x, y}) \text{\ ref \ real} : (\text{random} > .5 (x \mid y) \text{\ an \ *operation-declaration}*)?

e) Declare an \(*\text{operator} m\text{\ create} \) so that \( m\text{\ create} \) has the same value as \( \text{m\ create}(x, a) \) in line 10.5.12.c.

8.4 Elaboration of operation declarations

a) What is the value possessed by \( \text{op} m \) in the reach of \( \text{op} g = (\text{real \ a}) \text{\ int} \) round a? 

b) Is \( \text{op} (\text{real}) \text{\ real} g = \text{random} \) an \(*\text{operation-declaration}*)?

c) What is the value of the \(*\text{formula} m^* + 123 \text{\ si} (m^* + 1000 \text{\ si} 2)\text{\ si} \) using the declaration of \( m\text{\ in} \) as in 8.3?

d) Is \( \text{op} g = (\text{prog bool \ a, b}) \text{\ bool} : (a \mid \text{true} \mid b)\text{\ an \ *operation-declaration}*)?

e) Is \( \text{op} (\text{real, real}) \text{\ real} a = +m \) an \(*\text{operation-declaration}*)?

8.5 Dyadic indications and operators

a) How many defining occurrences may be identified by an applied occurrence of \(*\text{dyadic-indication}*)?

b) How many operator defining occurrences of \(*\text{operator} m\text{\ in} \) are in the \(*\text{standard- Tue}*)?

c) How many \(*\text{priority-declarations}*) are in the \(*\text{standard- Prelude}*)?

d) Where is the \(*\text{priority-declaration}*) for the \(*\text{operator} m\text{\ in} \) in line 3 of 70.5.1.i in the Report?

e) Is \(*\text{op} ::= = m\text{\ dyadic-indication}*)?

8.6 Identification of dyadic indications

a) Is \(*\text{opriority} + = 8, + = 9g\text{\ a \ *priority-declaration}*)?

b) Can a proper \(*\text{program} contain\)

\(*\text{priority abs} = 9, x ::= \text{abs} x\text{\ in} ?

c) Why does the \(*\text{abs} x\text{\ occur in the description of the repetitive statement [line 2, a, b, 9.c]} ?

d) Are \(*\text{dyadic-indications}*) subject to protection?

e) Are \(*\text{operators}*) subject to protection?

8.7 Identification of operators

a) In line 11.11.y of the Report, the \(*\text{fo1u1a} m\text{\ value of ec - in occurs. Where is the defining occurrence of its \(*\text{operator}*)?
b) In line 11.11. at of the Report, the *formula* m of occurs. Where is the defining occurrence of its *operator*?

c) In line 11.11.1 of the Report, the *formula* m = zeron occurs. Where is the defining occurrence of its *operator*?

d) Where is the defining occurrence of the *operator* mgn in the *formula* m[0] or m[5n]?

e) Where is the defining occurrence of the *operator* m in the *formula* m"a" < (setting )?

8.8 Elaboration of formulas

a) What is the value possessed by m in m[0] = (real a) bool : a > 0a?

b) What *closed-clause* is elaborated as a result of the elaboration of the *formula* m[0] in the reach of the *declaration* above?

8.9 Monadic operators

a) What is the value of n2 +3 -4 -3n?

b) Is nx :=: yn a *formula*?

c) Is nx :== real : random a *formula*?

d) Is real * real : a *formula*?

e) What is the value of n-1 i 2 = -1 i -2n?

8.10 Related modes

a) Are the modes uproc int and nreal related?

b) Are the modes uproc ref int and nref proc int related?

c) Are the modes uproc union(int, Real) and nunion(proc int, bool)n related?

d) Can the *declarer* nunion(proc Real, proc)n be contained in a proper *program*?

e) Can m[op - = (union(bool, ref char) aj int : 2 : op = (union(ref int, char) a) : 3 ; (Char := "a")n be contained in a proper *program*?

8.11 Peano curves

a) What would the *formula* mfalse + falso accomplish?

b) Write this algorithms using four mutually recursive procedures.

c) Translate the algorithms into FORTRAN.

8.12 Chinese rings

a) What is printed by n2 down 0n?

b) What is printed by n3 down 0n?

c) What is the purpose of the *declaration* mint a := a1?

d) What is printed by n6 down 2n?

e) Rewrite this algorithms 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-case• 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•.
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
- FORELSE : 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
- FORELSE : ADIC formula ; cohesion ; base.
[R.1.2.4.c]. It provides three production rules for the metalanguage, which are
- FORELSE : ADIC formula.
- FORELSE : cohesion.
and
- FORELSE : 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
The terminal productions of metanotions 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

formula, cohesion, base

firm, weak, soft.

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.

*RMODE* : MODE.

*LIST* : list ; sequence.

*RFIELDSETY* : FIEL.

*ROWWSETY* : ROW.

Secondly, the metarule

*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

*LOSETY* : LMODE

*ROWSETY* : RMODE.

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* : LMODE

*ROWSETY* : RMODE.

The two rules

*LFIELDSETY* : FIEL.

*AFIELDSETY* : and

[R.1.2.2.i,v] are a rule for *selection*.

There are two rules of the metanotion *EMPH:. *

*COMPLEX* ; struct.

*LENGTH* : letter

[R.1.2.2.s] and

*MODE* ; real field

[R.1.2.2.v]. This metanotions in some shortening the rule:

9.4 The hyper-rules

A good introduction to a section 3.0.1 of several rules which extensively elsewhere as follows:

*NOTION* option :

*NOTION* option

As a next step we substitute *integral part* of

*EMPTY*. This will strict language. The

*integral part of *

Note that *suggest*, i.e., one of its terms

*notions in ALGOL (protonotions) use
The two rules

\[ \text{LFIELDSETY : FIELDS and ; EMPTY.} \]
\[ \text{and} \]
\[ \text{RFIELDSETY : and FIELDS ; EMPTY.} \]

\[ \text{[R.1.2.2.q, r]} \text{are another pair which play a similar role in the} \]
\[ \text{rule for *selections* [R.8.5.2.1.a].} \]

There are two metarules in which the only direct production

\[ \text{of the metanotion is a protonotion. They are} \]
\[ \text{COMPLEX : structured with real field letter r letter e} \]
\[ \text{and real field letter i letter m}\]
\[ \text{[R.1.2.2.s]} \text{and} \]
\[ \text{LENGTH : letter l letter o letter n letter g.} \]
\[ \text{[R.1.2.2.v]. This means that the presence of one of these} \]
\[ \text{metanotations in some hyper-rule is merely for the convenience of} \]
\[ \text{shortening the rule and plays no other grammatical role.} \]

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

\[ \text{NOTION option : NOTION ; EMPTY.} \]
\[ \text{[R.3.0.1.b]. The first step in deriving production rules of the} \]
\[ \text{strict language, from the hyper-rules, is to make two new rules} \]
\[ \text{as follows:} \]
\[ \text{NOTION option : NOTION.} \]
\[ \text{and} \]
\[ \text{NOTION option : EMPTY.} \]

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.*
\[ \text{[R.5.1.2.1.b]. Examples are \texttt{m3.45n} and \texttt{n.45n}. Many of the} \]
\[ \text{notions in ALGOL 68 are similarly chosen so that the words} \]
\[ \text{(protonotions) used give some suggestion of the} \]

*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 \( axn \) is a certain \( *n \), then \( c(x)n \) is an \( *n \)-pack* and \( \text{begin } x \end{align} \) 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 *NOTIONs 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, \( \text{real } x \), \( \text{int } i \) is a *collateral-declaration* but \( \text{real } x o \) 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
\( \text{\#1234567890} \)
which is a *chain-of-digit-tokens-separated-by-EMPTIES*,
\( n_1, n_2, n_3 \)
which is a *chain-of-strong-integral-units-separated-by-commasymbols*, and
\( n_1, n_2, n_3 \)
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
\( \text{xyz}, \text{xyzxzzz}, \text{xxxxxzyzzz} \ldots \)
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.
\[
\text{begin \begin{align*}
\text{string a, b, c} ; \end{align*}}
\text{do print( \( a := \text{"x"}) \end{align} \) + \( ( b := \text{"y"}) \) + \( ( c := \text{"z"}) \) \end{align}}
\end{align}
\]
The reason that this language is of interest is that it is known
[R.H] that it cannot be described by a context-free grammar such
as that used for...

The three symbols are
*letter
*number
*letter

This corresponds to
The three hyper-rules
(R.1) *sentence :
(R.2) *number :
(R.3) *letter :

These three rules
Chapters 2 up
expresses the req
of the different
used to interpre
t one by one.
Rule
with

It is similar to
metanotion
*ALPHA*
we should need
is essential. In
productions of t
one*, *one-plus-o
somewhat similar
metanotion

We shall now
some of the produc
by the above grammar
1.4 and 1.5
production rules
minilanguage above)

If we substit
viz., "one", for t
new rule
(a) *sentence : o
The direct produ
terminal, since it
The three symbols of the language and their representations are

<table>
<thead>
<tr>
<th>symbol</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter x symbol</td>
<td>xxn</td>
</tr>
<tr>
<td>letter y symbol</td>
<td>yyn</td>
</tr>
<tr>
<td>letter z symbol</td>
<td>zzn</td>
</tr>
</tbody>
</table>

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.g of the Report, where the multiplicity of a *row* is being counted. Rule (iii) is present in order to satisfy the requirement of ALGOL 68 that only productions 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
To remedy this we use hyper-rule (iii) and, replacing \( \text{LETTER} \) by each one of its terminal productions in turn, we obtain

(b) \( \text{one letter } x : \text{ letter } x \text{ symbol} \)

(c) \( \text{one letter } y : \text{ letter } y \text{ symbol} \)

and

(d) \( \text{one letter } z : \text{ letter } z \text{ symbol} \)

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

\( \text{letter } x \text{ symbol, letter } y \text{ symbol, letter } z \text{ symbol} \)

is a terminal production of the notion \( \text{sentence} \). This means that we may speak of \( \text{xyz} \) as a \( \text{sentence} \) in the representation language.

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

(e) \( \text{sentence : one plus one letter } x \), \( \text{one plus one letter } y \), \( \text{one plus one letter } z \)

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

(f) \( \text{one plus one letter } x : \text{ one letter } x \), \( \text{one letter } x \)

(g) \( \text{one plus one letter } y : \text{ one letter } y \), \( \text{one letter } y \)

and

(h) \( \text{one plus one letter } z : \text{ one letter } z \), \( \text{one letter } z \)

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

\( \text{letter } x \text{ symbol, letter } x \text{ symbol, letter } y \text{ symbol, letter } z \text{ symbol} \)

is also a terminal production of \( \text{sentence} \). In the

\[
\text{sentence} \\
\text{one-plus-one-letter-x} \\
\text{one-plus-one-letter-y} \\
\text{one-plus-one-letter-z}
\]

representation language we may therefore now say that \( \text{xyz} \) is a \( \text{sentence} \) of the strict language. A sketch of the parse of this \( \text{sentence} \) is shown in figure 9.5. Perhaps we have now done enough of this to suggest that it is easy to show that \( \text{xyz} \) is a \( \text{sentence} \). A crucial new rule in this process is

\( \text{one plus one plus one plus one} \)

\( \text{one plus one plus one plus one} \)

\( \text{letter-x symbol, letter-x symbol, letter-y symbol, letter-z symbol} \)

is known. With this

It will also be described more concisely by

(II) \( \text{L : } x ; y ; \text{N : } ; \text{N p.} \)

and if we drop the \( \text{symbol} \) by agreeing then even more concisely by

(I) \( \text{L : } x ; y ; \text{N : } ; \text{N p.} \)

For the student of mathematics by nature an impossibly concise language we can afford are not algebraists understand them.

9.6 How to read the

representation

\[
\text{sentence} \\
\text{one-plus-one-letter-x} \\
\text{one-plus-one-letter-y} \\
\text{one-plus-one-letter-z}
\]

...
An ALGOL 68 Companion

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 \( \text{LETTER} \) are \( \text{letter-x-symbol}, \text{letter-y-symbol}, \) and \( \text{letter-z-symbol} \). 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.

The parse of \( \text{LETTER} \) is:

\[
\text{LETTER} : \text{letter-x-symbol} \mid \text{letter-y-symbol} \mid \text{letter-z-symbol}.
\]

\( \text{let}\text{ter-x-symbol} \) is

\[
\text{LETTER} : \text{one plus one plus one LETTER}. \\
\text{one plus one LETTER}, \text{one LETTER}.
\]

moreover, the process for finding more \( \text{sentences} \) of the language should be clear.

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

\[
(\text{I}) \quad \text{L} : \text{x} \mid \text{y} \mid \text{z}. \\
(\text{II}) \quad \text{N} : \text{; N P}. \\
(\text{III}) \quad \text{L} : \text{LETTER symbol}.
\]

and if we drop the requirement that every terminal must end with \( \text{*symbol*} \) by agreeing that \( \text{x}, \text{y} \) and \( \text{z} \) are already terminals, then even more concisely by

\[
(\text{I}) \quad \text{L} : \text{x} \mid \text{y} \mid \text{z}. \\
(\text{II}) \quad \text{N} : \text{; N P}. \\
(\text{III}) \quad \text{L} : \text{LETTER symbol}.
\]

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.
and comprehend their meaning.

The most important metanotion in ALGOL 68 is \texttt{*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

![Diagram of the \texttt{MODE} metanotion]

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 \texttt{*MODE*}, in which \texttt{*LETTER*} and \texttt{*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 hypernotations separated by semicolons and/or commas and followed by a point; e.g.,

•strong COERCEND : COERCEND;

strongly ADAPTED to COERCEND. [R.6.2.0.1.d]. If, in a given hypernotion, one or more of its metanotions is consistently replaced by a production of that

\[\text{strongly-ADAPTED-to-COERCEND}\]

---

\[\text{MOOD FORM}\]

\[\text{MODE}\]

\[\text{ADJUSTED}\]

\[\text{MOOD}\]

\[\text{TYPE FORESE}\]

\[\text{PLAIN}\]

\[\text{INTREAL}\]

\[\text{strongly-deprocedured-to-real-base}\]

---

\[\text{INTREAL}\]

\[\text{PLAIN}\]

\[\text{TYPE FORESE}\]

\[\text{MOOD}\]

\[\text{MOODE}\]

\[\text{STIRnly-deprocedured-to-MOODE-FOAM}\]

Fig.9.7

metanotion, then we have another hypernotion, 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 hypernotations have a common offshoot. For example, *strongly ADAPTED to COERCEND*

and

*STIRnly deprocedured to MOODE-FOAM* 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*.

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 :*

and [R.8.2.2.1.a]

*STIR~ly deprocedured to MOID FORM 820d :*

procedure MOID FORM :

STIR~ly 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 an hyper-rule for strict this case it is the *strongly a dead end. In

9.1 The syntactic

a) Is *MODE base*?
b) Is *all-missy-w*?
c) Is *cast* a not-d*
d) Is *MABEL identi*
e) Is *long-integr*

9.2 The metaru

a) How many produ

b) How many produc

c) How many produc

d) How many produc

e) What are the te

9.3 The metaru

a) Is *LETTER : LET*
b) How many produ

c) Is *NONSTOWED : metalanguage?*
d) Are the termi

e) Is *FIELD* a pr

9.4 The hyper-

a) Is *PARAMETER : *
b) Is *digit-toker

c) Is *in a +
some mode?*
d) What production

e) What production

9.5 A simple
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 *VICRAL*?

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 n( )n a *strong-closed-[m]-clause*, where [m] represents some mode?
d) What production of *LFIELDSETY* would be used in parsing m*m

9.5 A simple language
a) Define, by means of a two-level grammar, the language whose sentences are printed by
\begin{verbatim}
begin string a, b := "y", c ;
do print((a := "x") + (b := "y") + (c := "zz"))
endn.
\end{verbatim}
b) Define, by means of a two-level grammar, the language whose sentences are printed by
\begin{verbatim}
begin string a, b, c ;
do (print(a +:= "x") + (b +:= "y") + (c +:= "zz"))
endn.
\end{verbatim}
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-1` 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, b]?
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.8.2.5.1]?
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 declaration

10.1 Syntax

A typical `mode` declaration, which, by virtue of the `MODE` which the `standard-programmer` may assume, made a similar `decl

\begin{verbatim}
| mode-symbol mode |
| - -------------- |
| mode |
\end{verbatim}

shown in figure 10. `mode declaration equals symbol`, page [R.7.2.1.a]. The terminal `mode` of the `actual-indicator` more subjected to the expected that most `symbol standards` are `symbol bytes`, long long that one may write or each of which is le

10.2 Development

One purpose of shorthand whereby t uses some compoic out in full each ti a numerical analyst wish to use the con
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);
```

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 *ustring* and *uflon*, i.e., objects which look like identifiers but are in bold face (or underlined). Objects which are *mode-standards* are *ustring*, *uflon*, *file*, *compl*, *bits*, *bytes*, *long* *bytes*, *long* *bits*, *long* *long* *long* *compl*, etc. This means that one may write

```
mode file = into
```

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*
\[ \begin{align*}
\text{mode } v &= [1:n] \text{ real}, \\
\text{mode } m &= [1:n, 1:n] \text{ real}
\end{align*} \]

In the reach of this •declaration•, he may now use these •mode-indications• as •declarers• by declaring a vector variable with
\[ m \times x \text{ n} \]
or a matrix variable with
\[ m \times x \text{ 2 n} \]
It should be carefully noted that the value of \( m \times m \) which occurs in the •bounds• of these multiple variables is that which is possessed by \( m \times m \) at the time of elaboration of the •declaration• \( m \times x \text{ 1} \), \( m \times x \text{ 2 n} \) 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 \( \text{int } n \text{n} \), the elaboration of
\[ m := 5 ; \text{ mode } v = [1:n] \text{ real} ;
\]
\[ n := 3 ; y \times x ; \text{ print( } y \times x \text{ })
\]
should print the value \( 3 \) and not the value \( 5 \). This means that the •declaration• \( m \times x \text{ n} \) acts as though the eye were replaced by \( [1:n] \text{ real} \). This process is known as "developing" the •declarers• \([R.7.1.2.c]\). An important consequence is that, in the reach of the •declaration•
\[ \text{mode } v = [1:n] \text{ real},
\]
\[ \text{realvec} = [1:n] \text{ real}
\]
the •mode-indications• \( m \times m \) and \( m \times \text{realvec} \), 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
\[ \begin{align*}
\text{ref int } \text{name1} &= 1, \text{name2} = 2
\end{align*} \]
means that both \( \text{name1} \) and \( \text{name2} \) possess (different instances of) the same name. In the reach of the •declaration• \( \text{mode } f = [1:3] \text{ real} \), \( g = [1:2] \text{ real} \), the •indicants• \( m \times m \) and \( g \times m \) 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 \([R.8.1.2.c] \text{ Step 3}\).

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•
\[ \text{struct link} = (\text{int } \text{val}, \text{ ref link } \text{next}) ;
\]
In its reach, the elaboration of
\[ \text{link} a := (1, \text{ link} := (2, \text{ link} := (3, \text{ nil})) )
\]
will generate values linked together as shown in figure 10.3. In such a linked list, the value of the last name is \( \text{nil} \). If we try to write the mode specified by \( \text{link} \), using small syntactic marks, it will be
\[ \text{structured-with-real-field-letter-v-} \]

Call it an infinite however, always work that this infinite size. It is

### 10.4 Shielding and...
use "mode-declaration" carefully unde it represents the same mode which we are trying to write. Since the mode contains itself, it is not unnatural to 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.

Fig.10.3

10.4 Shielding and showing

If we consider the mode specified by \( m_n \), in the reach of
\[
\text{mode } m = \text{struct(} \text{real } v, m \text{ next)}
\]
we soon come to the conclusion that, unlike \( \text{link} \) above, the field selected by \( m \text{next} \) 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", \[\text{struct(real } v, m \text{ next)}\], "shows" \( [R.4.4.4.b] m_n \), which is the "mode-indication" on the left. It is therefore illegal. However, in
\[
\text{mode } n = \text{struct(} \text{real } v, \text{ref } n \text{ next)}
\]
the "actual-declarer" \[\text{struct(} \text{real } v, \text{ref } n \text{ next)}\] does not show \( m_n \), 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* `type l`, examples of *declarers* in which that is shielded are

```groff
struct(int k, ref m n)
,
ref struct(a n, char a)
,
uproc(a, int)
,
uproc(real) mu
```

and

```groff
[1: (mode `type l ; a k ; read(k) ; k)] real
```

Examples of *declarers* in which is not shielded are

```groff
a
,
ref a
,
uproc a
,
[1:n] a
```

and

```groff
union(int, a)
```

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 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

```groff
mode a = a,
b = proc b,
c = ref b,
d = [1:n] d,
e = union(e, char)
```

However, examination of the *declaration*

```groff
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 `a`, the elaboration of `ref g` [R.7.1.2 Step 1] involves the development of `a` and would give us the `declarer` `ref proc f`, which does indeed show `a`. 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 *sense*, *mode-indications* same manner as order that they *indication* used

whereupon the value of identification

Although the same way for

---

```
sub- symbol |
| unitary |
| [ | [ | [ | [ |
| for | for |
| u( ( a r |
| | for |
| | open- para |
symbol ) ] |
```

watched carefully a *mode-indication*

The reason for thi
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 *identifiers* and *dyadic-indications*, in order that they may not become confused with the same *indication* used elsewhere. It is possible therefore to write

```
begin
  (mode r = real ; r x := 2 ;
   g t----<-----
g)
  (mode r = int ; r x := 1
   g t----<-----g)
  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 watched carefully. It is that no *indicant* may be used both as a *mode-indication* and as a *monadic-indication* [R.1.4.5.h]. The reason for this is best shown by the following example.

```
(1) begin int b, c, e ; g ...
g
(2) begin mode a = real ;
(3) (a b) : b + c ) d e
(4) g ... g
```
10.6 Equivalence of mode indications

In the *mode-declaration*

\[ \text{mode } \text{a} = \text{ref real}, \]
\[ \text{mode } \text{b} = \text{ref real} \]

it is rather obvious that both \text{nau} and \text{mbn}, 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

\[ \text{struct } \text{a} = (\text{ref } \text{a} \text{ left}, \text{ref } \text{a} \text{ right}), \]
\[ \text{struct } \text{b} = (\text{ref } \text{b} \text{ left}, \text{ref struct } \text{b} \text{ right right),} \]
\[ \text{struct } \text{c} = (\text{ref } \text{d} \text{ left}, \text{ref } \text{e} \text{ right),} \]
\[ \text{struct } \text{d} = (\text{ref } \text{e} \text{ left}, \text{ref } \text{c} \text{ right),} \]
\[ \text{struct } \text{e} = (\text{ref } \text{c} \text{ left}, \text{ref } \text{d} \text{ right).} \]

\[ \text{in which it is not immediately clear whether the modes specified by } a, b, c, d \text{ and } e \text{ 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

We shall now consider binary trees. These are illustrated in figure 10.7.a. In each node the branch\( ^* \) in which it is not immediately clear whether the modes specified by \text{a}, \text{b}, \text{c}, \text{d} and \text{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.

\[^1\] For an authoritative reference, see Kn's section 2.3.1.
10.7 Binary trees

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. It 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 information, perhaps a string, which is an attribute of that particular node.

The necessary "mode-declaration" would be

```
struct node = (ref node left, string val, ref node right);
```

We may observe that the mode specified by `node` 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.10.7.a
```

```
Fig.10.7.b
```

```
Fig.10.8.a
```

```
Fig.10.8.b
```

(1) For an authoritative discussion of binary trees, see Kauth [Knj 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 ns into a tree whose root is referred to by nroot is as follows.

\[ \text{proc insert} = (\text{string } s, \text{ref ref node } \text{root}) : \]

\[ (\text{ref ref node } n := \text{root} ; \]

\[ \text{while} (\text{ref node } : n) :\!\! = \text{nil} \text{ do} \]

\[ n := (s < \text{val of } n \mid \text{left of } n \mid \text{right of } n) ; \]

\[ (\text{ref ref node } : n) := \text{node} := (\text{nil, s}, \text{nil}) \]

Suppose that we start with an empty tree, i.e., the declaration:

\[ \text{ref node } \text{tree} := \text{nil} \]

and then elaborate the call \( \text{insert}(\text{"jim", tree}) \). The 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 nroot and the actual parameter tree are the same, viz., that specified by \( \text{ref ref node} \), so that no coercion occurs when the parameter is passed.

The declaration \( \text{ref ref node } n := \text{root} \) implies that the mode of \( \text{mn} \) is that specified by \( \text{ref ref node} \). Since \( \text{nroot} \) is of mode specified by \( \text{ref ref node} \), the initializing assignment to \( \text{mn} \) involves no coercion. In the assignment:

\[ \text{node} := (\text{nil, s}, \text{nil}) \]

the second occurrence of \( \text{node} \) is a "global-generator" generating a name of mode \( \text{ref node} \), to which is assigned the value of the "structure-display" \( s(\text{nil, s}, \text{nil}) \). Because the mode of \( \text{mn} \) is \( \text{ref ref node} \), it must be dereferenced once before the new node is assigned. This is the reason for the cast \( \text{ref ref node} : n \). This cast is necessary. In fact, \( n := \text{node} \) is not an assignment, for there is one reference to \( n \) too many on the left.
If now we elaborate the `call minsert("sam", tree)`, we have what is shown in figure 10.8.e. Here we have effectively elaborated the assignment `nn := right of nu in going from figure 10.8.d to 10.8.e. In the `selection` right of `nn` 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 `nn` is thus `ref ref node`, since the field

![Fig.10.8.e](image1)

![Fig.10.8.f](image2)

selected by `right of `nn` is thus a name which refers to a name in a node. Since the mode of `nn` is `ref ref ref node`, the assignment now takes place without further coercion. This moves `nn` down the tree by one node. After elaboration of `minsert("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:

```algol68
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 `nn` 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* `csearch("sam", tree)` would result in the situation pictured in figure 10.9.

The *variable* `nnn` serves to remember where the node was found. In the *assignment* `nnn := nn`, `nnn` is dereferenced twice. Note also that in the *formula* `as = val of nn`, first `nn` is dereferenced twice, then `val of 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.

```algol68
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 n then go_to done
         else n := (s < val of n ? left of n : right of n)
     done:
     (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 `nnn` which follows the label `done : n`, after this `nnn` has been dereferenced once.

### 10.11 Tree walk

Another instance of a "tree walk" node as the node, e.g., preorder walk is called by Knuth [Kn] for the first reaching of branches, or for the tree would perform a walk in the order.

We shall call the binary tree. The tree provides a new node empty, then do we know how one, first we walk the right

In lines 3 and 6 of the nodes, in a binary tree. We shall call the pre walk we for the tree walk (tree) := show that the actual

We may now it to perform a form of a *procedure

```algol68
proc post wa:
begin proc
    (x #: n)
    q(left := q(root))
end
```
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 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)
```

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 `nil`). For the pre walk we interchange lines 3 and 4 (except for the `nil`). 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))
      g(root)
      end
```

```
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 up, qua and qua 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 qua, ref node right)
```

The steps in the process, after which the situation pictured in figure 10.12.a

The situation at some moment in moving down the tree is

```
<table>
<thead>
<tr>
<th>node</th>
<th>qua</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

only add some way to say the "condition"

```
mn val qua
```

One should also check that the procedure correctly and works properly when the walk crosses the node. The situation is

```
<table>
<thead>
<tr>
<th>node</th>
<th>qua</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

and the steps in the process are

```
left
```

right
The steps in the process of moving down are

\[
\begin{align*}
\text{u} & := \text{left of } q ; \\
\text{left of } q & := p ; \\
p & := q ; \\
q & := r \} u
\end{align*}
\]

after which the situation is as shown in figure 10.12.b. We need

\[
\begin{align*}
\text{aqn} & \quad \text{apn} \\
: & \quad : \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
\end{align*}
\]

Fig.10.12.b

only add some way to stop this process. This is accomplished by the *condition*

\[
\text{u(ref node : q)} := \text{nile}
\]

One should also check that the process starts from the root correctly and works properly when \( \text{u(ref node : q)} := \text{nile} \).

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

\[
\begin{align*}
\text{aqn} & \quad \text{apn} \\
: & \quad : \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
o & \quad o \\
\end{align*}
\]

Fig.10.12.c

and the steps in the process are

\[
\begin{align*}
\text{ur} & := q ; \\
q & := \text{right of } p ; \\
\text{right of } p & := \text{left of } p ; \\
\text{left of } p & := \text{ru}
\end{align*}
\]
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

```
\text{tag of } p := \text{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,

```
\text{when} ( \text{tag node : } q) := \text{root}
```

and whether we should change to moving across

```
\text{tag of } p := \text{false}
```

Also, as we move up, the value of the flag field is restored to \text{false}.

```
\text{p} := \text{nil}
```

Fig.10.12.d

The complete algorithm is expressed as follows:

```
\text{proc} \text{walk} = ( \text{ref node root, proc (ref node) action}) : 
\text{begin} \text{ref node } p := \text{root, } q := \text{root, } r ;
\text{if} \text{root} := \text{nil then}
\text{down} := \text{while} ( \text{ref node : } q) := \text{nil do}
\text{(see figure 10.12.a)}
\text{r} := \text{left of } q ; \text{left of } q := p ; \text{p} := q ;
\text{q} := r \text{ (see figure 10.12.b)} ;
\text{across} := \text{see figure 10.12.c}
\text{r} := q ; \text{q} := \text{right of } p ; \text{right of } p := \text{left of } p ;
\text{left of } p := r ; \text{(see figure 10.12.d)}
\text{tag of } p := \text{true} ; \text{action}(p) ;
\text{if} ( \text{ref node : } q) := \text{nil} \text{then} \text{down} \text{fi ;}
\text{up} := \text{while} ( \text{ref node : } q) := \text{root do}
\text{if} \text{tag of } p
\text{then} \text{tag of } p := \text{false} ; \text{r} := \text{right of } p ;
\text{right of } p := q ; q := p ; \text{p} := r
\text{else} \text{across}
\text{fi}
\text{fi}
\text{end walk}
```

10.1 Syntax

a) Is \text{mode real} = \text{real}.
b) Is \text{mode a = [1:10]}.c)
\text{Is mode a = [1:10].}
d) Is \text{union a = (b, c).}
e) Is \text{abstract u = (b, c).}

10.2 Development

a) In the reach of \text{proc bm, develop b}.
b) What is printed \text{print (ypb v) on c).}
c) Develop the \text{*dec}.
d) Develop the \text{dec}.
e) Develop the \text{dec}.

10.3 Infinite modes

a) What are the two \text{10.3}?
b) What are the three \text{10.3}?
c) Is the mode specified?
\text{b, b = struct (a +, b) build the list st e) Is slink a := (1, 1).}

10.4 Shielding a

a) Is \text{nm a shielded i}
b) Is \text{nm shown in n a = [1:10]}.b =
c) Can \text{mode a = ref}
d) Can \text{mode a = re}
e) Can \text{mode m = a}

10.5 Identification

a) Is \text{n(a : u) a v}

10.6 Equivalence

a) In the reach of \text{specified by n}
b) Are the nodes specified \text{a = struct (ref a}}
c) Simplify the \text{*struct (int u, ref a).}
d) In the reach of n
10.1 Syntax

a) Is umode real = long into a •mode-declaration•?
b) Is umode a = [ 1:n ]real a •mode-declaration•?
c) Is umode f = [ ]real a •mode-declaration•?
d) Is union a = ( b ) u a •mode-declaration•?
e) Is ustruct u = ( int q, real s ) u a •mode-declaration•?

10.2 Development

a) In the reach of umode EIQ£ c, develop the •declarer•.
b) What is printed by
   print(ubegin mode a = [ 1:2 ] int ; ref a v ;
   print(yubh v) endl?c)

c) Develop the •declarer• in 11.11.t of the Report.
d) Develop the •declarer• in 11.11.k of the Report.
e) Develop the •declarer• in 11.12.w of the Report.

10.3 Infinite modes

a) What are the two occurrences of ulinks on line 4 in section 10.3?
b) What are the three occurrences of ulinks on line 6 of section 10.3?
c) Is the mode specified by u, in the reach of umode a = ref
   b, b = struct(a a, d d)n, an infinite mode?
d) Build the list structure shown in figure 10.3 from top down.
e) Is ustruct a := (1, (2, (3, nil))) u a •declaration•?

10.4 Shielding and showing

a) Is uni shielded in u[ 1:n ] ustruct(a a, int b)u?
b) Is uni shown in ustruct(ref a a, b h)u, in the reach of umode
   a = [ 1:10 ]m, b = proc m?
c) Can umode a = ref proc m be contained in a proper •program•?
d) Can umode m1 = ref m2, m2 = struct(m1 f)v be contained in a
   proper •program•?
e) Can umode m1 = union(m2, m3), m2 = struct(ref m1 a, [ 1:n ]m3
   b), m3 = proc(m1)u be contained in a proper •program•?

10.5 Identification

a) Is uni u( b : u ) u a •formula• or a •declaration•?

10.6 Equivalence of mode indications

a) In the reach of umode a = [ 1:10 ] charn, are the modes
   specified by uni and ustringo equivalent?
b) Are the modes specified by uni and uint, in the reach of umode
   a = struct(ref a x), b = ref struct(b x)n, equivalent?
c) Simplify the •mode-declaration• ustruct a = ( int u, ref
   struct(int u, ref a v) v)u.
d) In the reach of ustruct a = ( ref b r, bool s), b = ( bool s,
10.7 Binary trees
a) In the reach of mode node = ref struct (node l, string val, node r), does node 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 uinsert("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
uprint(val of left of tree)?
c) For figure 10.8.f, what is the value of n( (ref node : root) := : nil)?
d) For figure 10.8.f, what is the value of nleft of tree := : nil and that of nleft of n := : (ref node : nil) nil?

d) For figure 10.8.f, what is the value of nleft of n := : (ref node : nil) nil?

d) For figure 10.8.f, what is the value of nleft of n := : (ref node : nil) nil?

10.9 Tree searching
a) Rewrite the *declaration* of ossearch without using a *completer*.

10.11 Tree walking
a) Define a *procedure* np1o such that np1o (tree) will print the strings of a tree (see figure 10.11) in the form
(((\!A\!)B((\!C\!))))
b) Define a *procedure* np2o such that np2o (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

\begin{verbatim}
uprint
and
cread
\end{verbatim}

We have met them before in several examples in preceding chapters. The procedure `uprint` is used for unformatted output to the standard output file (probably a line printer) and the procedure `cread` is used for unformatted input from the standard input file (probably a card reader). Examples of their use are

\begin{verbatim}
uprint("answer",".,i")
uprint((new page,title))
and
cread(x)
cread((i,j))
cread((x1,new line,y1))
cread((a,space,b,space,c))
\end{verbatim}

An important point to notice is that both `uprint` and `cread` accept only one *actual-parameter*. Thus `uprint(x,y)` is incorrect. The mode of the *parameter* of `uprint` and `cread` begins with *row-of-* . This means that `cread((i,j))` or `uprint((i,j))` is acceptable since `a(i,j)` is a *row-display*. Note that `uprint((x))` is as good as `uprint(x)`, for `a(x)` is a *closed-clause* whose value is `ox` and `ox` will be rowed to a multiple value, a row with one element.

Observe that, in addition to *variables* like `ox` (and for `uprint`, *constants* like `"answer",".`), the *units* of the *row-display* (or the single *parameter*) may be certain layout procedures like `ospace`, `backspace`, `new line`, or `new page`, to allow for a rudimentary control over the standard input and output files. Thus `uprint((new page,""page10"",new line,"name",space,"address"))`, should result in the following output at the top of a new page.
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* `outtypename` and `intypename` [R.10.5.0.1.b,e]. We call these "hidden" because, although they appear in the Report in the form of `outtypename` and `intypename`, 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 a *indicant* with no defining occurrence.

The declaration of `outtypename` may be paraphrased as follows: `outtypename` specifies a union of the modes `int`, `real`, `begin` and `output`, together with prefixed unions where applicable, and all multiple and/or structured modes built from these. Examples are `of list, string, complex and of struct[int n, [bool b]n]`. Note that values of each of these modes are constants.

If we consider a union of the same modes as for `outtypename`, but each preceded by *reference to*, then we have the mode specified by `intypename`. Examples are `ref list, ref char, ref int, ref string, ref complex and ref struct[int n, [bool b]n]`.

Thus, `outtypename` is an appropriate union of those constants which we might expect to print and `intypename` is a union of the corresponding *variables*.

It is now perhaps convenient, for our discussion, to suppose that there is a *mode-declaration*:
```
*mode printer = union(outtypename, proc(file));
*readtype = union(intypename, 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 `printf` is of the mode specified by `outtypename` and that of `read` is that specified by `intypename`. This means, in particular, that the `ux` in `printf(x)` will be subjected syntactically to the coercion of dereferencing to `uxval`, unifying to `outtypename` and then moving to `outtypename`, whereas in `printf((x, y))`, the last coercion is not necessary since `ux` in `printf(x)` will be already of `outtypename`.

In `printf(new page)`, the `new page` is of a priori mode `proc(file)` and it is unified to `outtypename` and moved to `outtypename`. These particular coercions are of little concern to the programmer except perhaps that their understanding helps to prevent such errors as `printf(x, y)`.

11.4 Standard output format

We shall now examine what to expect of the appearance of *constant* on the standard output file `stand out` as a result of a *call* of `printf`. For this purpose, the mode specified by

The hidden *indicant* `out`, `complex`, `bool` plagues, if at all, the output appearance of values of each of these.

We shall also see if we are to give that in our environment widths [R.10.5.1.d] is `A, B, C` and **c** in [R.10.5.1.m, 10.5.1.n].

With these assumptions, `printf(x)` is:

```
+1 +0
-3.40000002E-3
```

The value `-3.4000` was not enough room for a value occupies 6 constant 13 (real complex value 28 and each of these is unless we are at the

Multiple values specified by `printf`. For example, the result of `printf(3.4)`
```
+1 +0
-3.40000002E-3
```

Actually, the description of `printf` specifies that each of the *parameters* `printf(a, b, c)` [R.10.5.0.2.d] to a `printf` and printed with the subsequents for example, that

The exceptions *string* has the `printf("abcd")` is if it were treated as
```
+1 +0
```
rather than
We shall also need some assumptions about the environment, if we are to give illustrative examples. Therefore let us assume that, in our environment, mint width \([R.10.5.1.3.h]\) is \(\#5\), real width \([R.10.5.1.3.i]\) is \(\#7\), next width \([R.10.5.1.3.j]\) is \(\#2\) and max char\(\text{stand out channel}\)\(\text{m}\) (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`

```plaintext
oprint((newline, true, false, 1, 0, -1, 1.2, 0.0, -.0034, "a", "abc", \[12\]))
```

is

```
+1 +0 -1 +1.200000E +0 +0.000000E +0
-3.400000E -3 A ABC +1.000000E +0 +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 (mint width + 1n) print positions, a real constant 13 (real width + exp width + 4n) 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 `mouitypen` and therefore multiple values may be printed. For example, in the reach of \([1:3]\)\[lint\] \(u1 = (1, 2, 3)\)\(n\), the result of `oprint((u1, 4))\(n\)` is

```
+1 +2 +3 +4
```

Also, in the reach of \([1:2, 1:2]\)\[lint\] \(n2 = ((5, 6), (7, 8))\)\(n\), the result of `oprint(n2)\(n\)` is

```
+5 +6 +7 +8
```

Actually, the description of `oprint` \([R.10.5.2.1.a,b]\) indicates that each of the `units` of a `row-display` \(n(a, b, c, d)\)\(n\) is first "straightened" (unravelled) \([R.10.5.0.2.c]\) to a value of mode specified by \(n[\text{simpout}]\) 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`s in `oprint(n2)\(n\)` given above, is, within the `procedure` `oprint`, straightened from `mouitypen` to `n[\text{simpout}]` \([R.10.5.2.1.b, 10.5.0.2.a]\). Thus, all multiple values and all structures (except for `ncomplex` and `nstring`), which are already in `n[\text{simpout}]` are straightened to `n[\text{simpout}]` before printing.

The exceptions for `nstring` and `ncomplex` are that, although `nstring` has the mode `row of character`, the result of `oprint("abcd")\(n\)` is `ABCD` and not `A B C D`, which would be the case if it were treated like other multiple values, and `oprint(1.2 \[3:4])\(n\)` gives

```
+1.200000E +0 I+3.400000E +0
```

rather than
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
\[ \text{print}(x) + \text{print}(y) \]
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
\[ \text{cprint}(x) ; \text{print}(y) \]
c is exactly the same as that of \( \text{cprint}((x, y)) \). In particular, each call of \text{cprint} 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 \text{new lines} or \text{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 \text{int string}, \text{real string}, \text{dec string} and the same preceded by \text{long string} if applicable [R.10.5.1.J.c,d,e].

Thus, if it is desired to print the integral value \( a \in \mathbb{Z} \) using a width of three print positions, this can be done by
\[ \text{cprint}(\text{int string}(a, 3, 10)) \]
The second parameter of \text{int string} is the string length and the third is the radix. The call
\[ \text{cprint}(\text{int string}(25, 3, 8)) \]
would yield \( +31 \), because \( 25 = 3 \times 8 + 1 \). For real values the value of \( \text{real string}(3.14, 10, 3, 2) \) is \( +3.1400000000 \) and the value of \( \text{dec string}(3.14, 10, 3) \) is \( 3.140 \).

Notice that the value of \( \text{int string}(25, 8, 10) \) is \( +0000025 \), so that those who require zero suppression must either accept what they get from \text{cprint} or use formatted output. Another possibility is to do the zero suppression oneself by defining a procedure like the hidden procedure \text{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 variable of \text{read} is \text{read types}, where \text{read types} is \text{union(int type, proc(file))}. Thus, in \text{read}(a, b, c), the user is either a layout procedure, like \text{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 \text{simp layout}, each preceded by \text{reference to}, i.e., \text{ref int}, \text{ref real}, \text{ref compl}, \text{ref bool}, \text{ref char}, \text{ref string} and their long version convenience let us simplify. We shall turn.

In the reach of \( a \in \mathbb{Z} \) would be satisfied
\[ +304 \]
and \( a \in \mathbb{R} \) would be satisfied
\[ +1.2000000E+0 +3.4000000E+0 \]
which would be the case if it were treated in the same way as the other structured values.

The procedure \text{read} from the current position finds as a value of possibility that, integral-denotations and the first digit blanks may appear before characters may be present.\text{long symbol} is not.

In the reach of \[ \text{read}(1x, x) \] would be satisfied
\[ 2 \]
or by
\[ 6.789 \] or by
\[ 123-4.56 \]
Note that the values are separated by blanks, naturally do this.

In the reach of \( a \in \mathbb{R} \)
\[ 3.456 \]
would be satisfied
\[ -0.000345 \]
Observe that although when necessary, there to \text{compat}. If the user \text{compat}, then it expect separated by a \text{plus-i}.

In the reach of character from the input character is a blank \text{read}(c) \] will and assign these to \text{acters} of line or one of the \text{term} of standard characters are taken by whichever bound is of them are flexible, \text{flex} character silly, the lower bound of \( a \) for terminators as for ex
In the reach of \texttt{mint} \(i\), \texttt{long int} \(ln\), the \texttt{cancl \ urread(\{i, ln\})} would be satisfied by two \texttt{integral-denotations} like
\begin{align*}
3
& \quad -2
\end{align*}

or
\begin{align*}
+ 304 & \quad - 00000005
\end{align*}

The \texttt{procedure \ urread} 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 \texttt{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 \texttt{nint} as for \texttt{long int} (a \texttt{long-symbol} is not used).

In the reach of \texttt{real} \(x\), \texttt{long real} \(1xn\), the \texttt{cancl \ urread(\{lx, x\})} would be satisfied by
\begin{align*}
2 & \quad -3.95
\end{align*}

or by
\begin{align*}
6.789 & \quad e + 2 .00001
\end{align*}

or by
\begin{align*}
12.456 & \quad e + 3.89
\end{align*}

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 \texttt{complex} \(z\), \texttt{bool} \(bn\), the \texttt{cancl \ urread(\{z, b\})} would be satisfied by
\begin{align*}
3.456 & \quad e -3 i + 7.69
\end{align*}

or by
\begin{align*}
.00034560
\end{align*}

Observe that although \texttt{urread} will widen from \texttt{nint} to \texttt{real}, when necessary, there is here no widening from \texttt{nint} or \texttt{real} to \texttt{complex}. If the \texttt{variable} to be assigned to is of mode \texttt{pref \ complex}, then it expects two values acceptable as \texttt{real} and separated by a \texttt{plus-i-times-symbol}.

In the reach of \texttt{character} \(c\), \texttt{urread(c)} merely reads the next character from the input file and assigns it to \texttt{c} even if that character is a blank. In the reach of \texttt{[1:10]char c1r}, \texttt{urread(c1r)} will read exactly 10 characters, including blanks, and assign these to \texttt{c1r}. If however, we have \texttt{[1:3] flex \ character c1r}, \texttt{urread(c1r)} reads characters until it finds the end of line or one of the characters which belongs to the string \texttt{term of stand in} \(R.10.5.1.mm\), whereupon the preceding characters are taken to be those to be assigned to \texttt{c1r}. Whichever bound is flexible is then adjusted suitably. If both of them are flexible, e.g., in the reach of \texttt{[0 flex: 0 flex] char silly}, the \texttt{cancl \ urread(silly)} will result in a lower bound of \texttt{1} for \texttt{nnsilly}. The programmer may specify the terminators as for example in \texttt{term of stand in := "?!"}, which
changes the set of terminators to "?" or "!".

For multiple and structured variables in the union union, the first step is to straighten to o ig!E!n, where o ig!E!n is the union of modes discussed above. Thus, in the reach of of 1: 3, 1: 2 real x2, struct (int a, bool b) cn, the call nread((x2,c)) would be satisfied by 3, 1: 6, 4: 2 , 7, 5 0.

11.7 String to numeric conversion

The procedure nread must of necessity convert character strings to integral or real values, and in doing so it makes use of three standard procedures, nstring int, string dec and nstring real [R.10.5.2.2, 3, 4, 5]. These procedures are not hidden. The programmer may use them himself. The first procedure, nstring 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 nstring int("+23", 10) is +23 (the second parameter is the radix). The procedure nstring dec converts a *variable-point-numeral*, e.g., \( n^{*}2.3450^{m} \), to a real value and nstring real converts a *floating-point-numeral*, e.g., \( n^{*}2.345e^{-2}^{m} \), to a real value. The value of nstring dec("2.345") is 2.345 and that of nstring real("2.345e-2") is 2.345e-2. These procedures, although available, are not likely to be useful for input since nread itself has all the flexibility needed. However, they may well be used for internal manipulation of strings.

Another procedure which may be mentioned here is nchar in string [R.10.5.1.2, 3, 4, 5]. 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 nchar 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, nchar number, line number and page number [R.10.5.1.2, 3, 4, 5], each yielding an integral value, the three coordinates of the nbook. In the case of the standard input file, the call nchar number(void in), line number(void in) and page number(void in) should each yield the value 1 after the call nread((c, back space)) if this is the first call of nread and is in the reach of nchar cn. Notice that these procedures deliver integral values and not names, so that they are for end position in the file.

There are also three and ofile ended [R.10.5.1.2, 3, 4, 5] appropriate boolean value made between ofile exit capacity has been reached [R.10.5.1.2. k], which the file has been exhausted if it is a card reader, always to be false become true each time a particular job. The of always yield false, but [R.10.5.1.1, 3, 4, 5] outo is not an input for become true when the page is out or when the box of paper.

11.9 Other files

It is worthwhile noting as outp(stand out, x) is, in fact, this is defined [R.10.5.2.1, 3, 4, 5]. If file is available, say fa, then what we have standard files applies as outp(f, x) and ofout(closed) by the programme another chapter. Another standard opened automatically, is saving intermediate real program. When the end will be lost, since the by the standard-postlude and overwrite bin and overwrite bin is of.linspace. For example want temporarily to save this could be accomplished array can then be recalled and saved is available x1 and xget bin(f, x1); then these two calls:

11.2 Print and read

a) Is aprint (new page, new output)
that they are for enquiry only and cannot be used to alter the position in the file.

There are also three •procedures• outlined, page ended and file ended \([\text{R.10.5.1.2.b, i,j}]\), 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 \([\text{R.10.5.1.2.k}]\), which tests whether the usable information in the file has been exhausted. In the case of the file \(\text{stand in}\), if it is a card reader, then file ended(\text{stand in})\(\) is likely always to be \(\text{false}\), but logical file ended(\text{stand in})\(\) may become \(\text{true}\)-each time we reach the end of the data for a particular job. The •call• logical file ended(\text{stand out})\(\) will always yield \(\text{false}\), because any possible(\text{stand out channel})\(\) \([\text{R.10.5.1.1.f, 10.5.1.3.b}]\) is likely to be \(\text{false}\), i.e., \(\text{stand out}\) is not an input file. But file ended(\text{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 \text{print}(x)\(\) is the same as \text{put}(\text{stand out, } x)\(\) and \text{read}(x)\(\) is the same as \text{get}(\text{stand in, } x)\(\); in fact, this is the way that \text{print} and \text{read} are defined \([\text{R.10.5.2.1.a, 10.5.2.2.a}]\). This means that if another file is available, say in the reach of the •declaration• file \(f\), then what we have said about unformatted transput on the standard files applies also to the file \(f\) by using, e.g., \text{put}(f, x)\(\) and \text{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 \text{stand back}, is \text{read} bin. 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 \([\text{R.10.5.1.ii, 10.5.1.2.t}]\) by the •standard-postlude•. The two relevant •procedures• here are \text{write bin} and \text{read bin}. The mode of the •parameter• of \text{write bin} is \(\text{int}\), and that of \text{read bin} is \(\text{array}\). For example, in the reach of \(f[x:n]\) \text{real} \(x\), if we want temporarily to save the values of a rather large array, this could be accomplished by the •call• \text{write bin}(x)\(\). The array can then be recalled by \text{read bin}(x)\(\). If another file, say \(\text{msg}\), is available, the same could be done by \text{put bin}(f, x)\(\) and \text{get bin}(f, x)\(\), and if the file \(\text{msg}\) is not locked then these two •calls• might appear in different •programs•.

review questions

11.2 Print and read

a) Is \text{uprint}(\text{new page, new line})\(\) a •call•?
b) Is \textbf{aprint}(\texttt{nil}) a \texttt{•call•}?
c) What is the result of \textbf{aprint}(\texttt{get possible(stand in channel)})?
d) In the reach of \textbf{ref real xx} := \texttt{loc real} := 3.14, what is the result of \textbf{aprint(xx)}?
e) In the reach of \textbf{ref real xx} := \texttt{loc real} := 3.14, what is the result of \textbf{aprint(Ref real : xx)}?

11.3 Transput types

a) What is the result of \textbf{aprint}(\texttt{for i by 2 to 10 do 3})?
b) Can \texttt{nil} be coerced to \texttt{of printtype}\?
c) In the reach of \textbf{ref real xx}, can \texttt{xxxx} be coerced to \texttt{of printtype}?
d) In the reach of \textbf{struct (ref c next, int n) s} := (\texttt{nil}, 2), what is the result of \textbf{aprint(s)}?
e) In the reach of \textbf{struct (int n, char c) fn}, is \texttt{read(f)} a \texttt{•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 \textbf{aprint(\texttt{("?", int width)})}?
b) What is the result of \textbf{aprint(\texttt{("?", space, "abc")})}?
c) In the reach of \textbf{ref real xx} := \texttt{loc real} := 3.14, what coercions occur to \texttt{xxxx} in \textbf{aprint(\texttt{("?", 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 \textbf{aprint(\texttt{("a", "b", "c")})} \texttt{ABC} or \texttt{A B C}?
References


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., n.m, 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 m[n.

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.

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., m123n and m000123n. b) No, but it is a *formula*. c) Yes. d) No, not if this value would exceed \( \text{max intn} \) [R.10.1.b].

1.8  a) Yes, e.g., possibly n2.34n and n23.4e-1m. 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*.


2.1  a) No. b) Yes. c) pref 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) No.

2.4  a) No, but value. b) Yes. c) No, time, but in the course of time.

2.5  a) Yes, but No, but the value may be changed. d) block

2.6  a) No. b) Yes, to-reference-to-inte

2.7  a) Yes. b) Yes.

2.8  a) pref ref

2.9  a) No. b) Yes.

2.10 a) Yes. b) reference-to-real. f)

2.11 a) The typ is No.

2.12 a) The typ is No.

2.13 a) the \( \text{nn} \) is a *reference-to-int*.

2.14 a) Four. b) B is equivalent to \( n \) := real. e) *reference-to-

2.15 a) No.

3.1  a) No. b) *expression* may possess

3.2  a) No. b) Five call and *void-cast-

3.3  a) m1, ca, ft

3.4  a) Yes. b) Yes

3.5  a) No. b) Yes.

3.6  a) The same as value *true* only whe
2.3  a) None. b) alloc char. c) alloc 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 var may be changed. d) alloc[1:3]proc real.

2.6  a) No. b) Yes, in the extended language. c) reference-to-reference-to-integral. d) of[1:3]proc real p

2.7  a) Yes. b) Yes. c) No. d) No.

2.8  a) ref real xx = loc ref real. b) ref real x = loc ref real. c) ref real y = loc real. d) ref real y (= loc real := 3.14n. d) It is not possible; moreover, if var has its usual meaning, then this is not a •declaration•.

2.9  a) No. b) Yes. c) No. d) No, but a rather foolish one.

2.10 a) Yes. b) Yes. c) No. d) ny + 2n. e) reference-to-reference-to-real. f) No.

2.12 a) The myn is dereferenced and the n3.14n is not. b) No.

2.13 a) the nun is an •integral-mode-identifier• but the nun is a •reference-to-integral-mode-identifier•; i.e., nun is a •constant• and nun is a •variable•. c) No.

2.14 a) Four. b) Both map and map are dereferenced. c) It is equivalent to nj := j + in. d) Yes. e) min. It's mode is •long-real•. e) •reference-to-long-real•.

3.1  a) No. b) Yes. c) [a + (b of (c[d]))) - en. 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) ma[1], a, i, c, sin(x), sin, x, cos(x + pi/2), cos, x, pi, 2n. d) No. e) It could be either, depending on the mode of map [R.9.2.g].

3.3  a) n1, ca, fn. 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) n35, item of a, i + n * 2, i := 2n.

3.5  a) No. b) Yes. c) No. d) Yes. e) Yes.

3.6  a) The same as that of n(2,3)n. b) It possesses the value •true• only when nx2[3,1] = x2[2,1]n. c) •2m. d) •2m. e)
No, because $n := 10$ is not a \textit{tertiary} and therefore not a \textit{lower-bound}.

3.7 a) Yes. b) No, it is a \textit{deprocessed-coercend} [R.8.2.2.1.a]. c) No, but $\cos(x > 0 | x | \pi/2) \equiv 1$ is a \textit{call}. d) When the mode of $n$ is \textit{procedure} with $M_1$ parameter reference to $M_2$ where $M_1$ and $M_2$ are terminal productions of $\text{MODE}$. e) When the mode of $n$ is \textit{procedure-with-M1-parameter-procedure-with-M2-parameter-M3}, i.e., $n$ is a \textit{procedure} with one \textit{parameter} which delivers a \textit{procedure} with one \textit{parameter}, and the modes of $n$ and $n$ are $M_1$ and $M_2$ respectively.

3.8 a) Yes. b) No, $n := x$ has no mode. c) Yes, provided that the mode, after soft coercion, of $n$ is \textit{reference-to-procedure-void}. d) Yes. e) No [R.8.2.3.1], but $\text{opro} p := (x := 3.14)$ is a \textit{declaration}.

3.9 a) No. b) Yes. c) No. d) Yes. e) When the mode of $n$ is structured, has a field selected by man whose mode is \textit{reference-to-M1} where $M_1$ is the a posteriori mode of $n$, or when $n$ is a \textit{variable} and will refer to structured values that have a field selected by man whose mode is $M_1$.

3.10 a) No. b) No, it is a \textit{field-selector} [R.7.1.1.i]. c) $n := 0 \text{ if } (h[c])$, e) $\text{of}(g(x))^n$. d) $n := a$ of $b$ is not a \textit{field-selector}. e) Yes, it could be.

3.11 a) Yes. b) \textit{false} (if the value of $n$ bits width $n$ is $\leq 3$). c) \textit{false}. d) No, the left \textit{operand} of the \textit{operator} $\cdot := n$, as declared in the \textit{standard- prelude}, must possess a name. e) \textit{false}.

3.12 a) No. b) No, $n := i + 1$ is not a \textit{tertiary}. c) No. d) $n$, $n$ is not a \textit{assignment}. e) Yes.

3.13 a) \textit{false}. b) \textit{true}. c) \textit{true}. d) Yes. e) $n := 3.14$ does not possess a name. e) Yes.

3.14 a) No. b) It looks like one, but $c.14$ cannot be strongly coerced to an integral value. c) An \textit{identity} relation. d) No, because $[1:1]$ is real is not a \textit{virtual-declarer}. e) No, because $\text{pref int} = n$ is not a \textit{tertiary}.

3.15 a) None. b) Eleven. c) A \textit{constant}. d) \textit{real}. e) None.

4.1 a) The same as that of $n$ if $n$ is not an \textit{assignment}. b) No. c) No. d) Yes. e) Yes. f) Yes.

4.2 a) \textit{false}. b) Some undefined integral value. c) \textit{false}. d) $\text{if} p \text{ then } q \text{ else } r \text{ then } b \text{ else } c \text{ then } e \text{ else } \text{ skip } \text{ then } \text{ skip } \text{ then } i$. e) $n(\text{ a }) (\text{ b }) \equiv (d | e | \text{ skip } | \text{ skip } | i \text{ skip })$.

4.3 a) No. b) \textit{case}, \textit{case} and $n(n \text{ c}) \equiv \textit{false}$. d) \textit{false}. e) No.
4.4  a) No. b) No. c) Yes, \textit{real} 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 \textit{min} identify the first, but \textit{min} := 2 \* i + 10 is not an *assignment* since \textit{min} does not possess a name. g) The last three occurrences of \textit{min} 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 \texttt{abcde}.

e) Yes, e.g., if the order of elaboration happens to be \texttt{cj} := i ; i := jn.

4.7  a) Yes. b) Yes. c) No. d) Yes.

e) \( n( x \text{ or } y \leq 1 ) ; n := 2 ; s . l : n := 1 ; rn. 


5.1  a) No, \texttt{real proc} is not a *declarer*. b) No, \texttt{real a} is not a *virtual-plane* \( \{ \texttt{R.7.1.1.x} \} \). c) \texttt{oproc real r2 = 2 \* random.} d) \texttt{oproc max = (real a, b) real : \{ a > b | a \} b }). e) \texttt{oproc recip = (ref real a) : a := 1 / an}.

5.2  a) No, unless \texttt{n} \* \texttt{m} has been redeclared and possesses an operation which delivers a name. b) \texttt{pref [real x[m]}. c) \texttt{m(real a = x + 1, real b = y ; a * b })\texttt{. d) n(real a = skip ; real \) a * a }). e) \texttt{e(int n = skip, int m = skip ; ref t:n)real al = skip ; real : \{ n < m | a[l][n] | a[l] })*.

5.3  a) The value is voided. b) \texttt{4.6*}, in the sense of numerical analysis. c) That of \texttt{ny}. d) The object \texttt{up(x, y)m} is not a call, since \texttt{pref ref real a = x[n]} is not an *identity-declaration*. e) \texttt{2.2w}, in the sense of numerical analysis.

5.4  a) \texttt{oproc p = (int a, proc ref int b) ; b *:= 2 \* an, but in most applications \texttt{oproc p = (int a, ref int b) ; b *:= 2 \* an would be sufficient. Note that since \texttt{ab} in passed by name in \texttt{ALGOL 60}, the side effects of \texttt{ab} := \texttt{b} \* \texttt{2} \* \texttt{an} occur twice but in \texttt{ab} := \texttt{2} \* \texttt{an} they occur only once.}

5.6  a) A *constant*. b) Because no assignment is made to \texttt{csu}. c) Because \texttt{csu} is a *constant* and \texttt{upgrow} requires a *variable* in its last *parameters*. d) It's value is irrelevant for it is used only in the *formula* \texttt{nt or -tn}. e) The same as that of \texttt{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, deproceduring and deproceduring. d) Dereferencing, deproceduring and dereferencing. e) 804a, 71b, 41b, 61e, 81a, b, c, 820d, 822a, 860a, 41b, c, 302b.

6.4 a) Deproceduring and unifying. b) No. c) A routine. d) No. e) No, x=0 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 m5a. d) Deproceduring and rowing. e) None, this is not a •cast• since rowing cannot be followed by unifying [R.8.2.4.1.b].


6.7 a) •Base, cohesion, formula, confrontation• b) cb, a of b, x, 2, x := 2, x, y, 3, y + 3, x := y + 3n. c) Yes, but its elaboration is undefined since the dereferencing of a •null• is undefined [R.8.2.1.2 Step 2].

6.8 a) 834a, 71b, 421b, 61e, 81a, b, c, d, 820d, 825b, a, 821a, 860a, 41b, 302b. b) No, there is no deuniting coercion. c) 74a, 54e, 71b, w, a, 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; 83c, 61e, 81a, 820d, 828a, 830a, 831a, b, 81b, c, d, 820g, 860a, 41b, 302b; 83c, 61e, 81a, b, c, d, 820d, 828b, 822a, 860a, 41b, c 302b. e) No, hipping cannot occur in a firm position.

6.9 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.

6.10 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) np on is softly deprocedured and m on is strongly dereferenced. c) m on is softly deprocedured and m on to kn is strongly hipped to •reference-to-real•. d) 6. e) No.

6.12 a) o1 on is weakly coerced, m on is strongly widened and then rowed to •row-of-real•. b) Yes, strongly-weakly to •real•. c) Yes. d) Yes. e) unrandom is strongly deprocedured and widened and m on is weakly coerced.

6.13 a) No. b) No. c) Yes, firmly-strongly. d) 6. e) No.

6.15 a) Yes. b) Yes, the balanced mode is •reference-to-real•. c) No, it cannot be balanced. d) m on is firm, the others strong. e) No.

6.16 a) The object m on := 1 on is interpreted as m on := m + 1 on so mm is dereferenced once, m on := 1 on is dereferenced as the left operand of m on (b) This is equivalent to •ref int c1 := loc int := m on := abs m on. First m on is dereferenced to •integral• and the absolute value of this integer is found. It is assigned to nam. Then a nam := abs m on is to by nam) is assigned the name. same name as that produced by repetition of the occurrences of main d) This is the FORTRAN program. It is not permitted for the

7.1 a) Yes, b) Yes, but rather use

7.2 a) No, m in character and boolean firm, so the m n •boolean•. d) Yes that it is in the •operator• m n.

7.3 a) •true•. b) x n is not a •tertiary•.

7.4 a) Yes, •true•. c) Yes [R.4.1] (union (int, real)

7.5 a) •true•. b) It should be use:

7.6 a) No, it is

8.1 a) No, it is

8.2 a) No, m :=

8.3 a) No, m :=

8.4 a) •(real •
to name. Then a name is created by bloc into, the assignment
name := abs ab is dereferenced and the integral value (referred
to by name) is assigned to this name. Finally name is made to
possess the same name as that possessed by name. This happens for each
repetition of the repetitive statement, in which there are five
occurrences of name, thus saving time or subscript calculation.
d) This is the position of the statement number 30 in the
FORTRAN program. It is redundant in ALGOL 68, but 130: 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 (m) 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 men.

7.3 a) true. b) false. c) true. d) Yes. e) No, nx :=
xn 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) nproc sqrt = (int
union (int, real) : (real x = sqrt(i) ; int j = round x ; ( j *
j = i ) j x )) n.

7.5 a) 4. b) Either 7 or 8 or 9 [R.10.4.2]. c) No,
if should be nse a p = /1n. d) Yes, surprisingly, and if the
value of num is of boolean mode, then the value of the
expression is that of num. 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) u(x + (-y))
- (( -( - (abs i)) ) over 2)n. d) Nine. e) No, it is a
confrontation. f) *2n.

8.2 a) No, m := n 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 m := n as a dyadic-indicant. e) No, perhaps the intention
was m := t. f) = 6, i = tn.

8.3 a) No, m := n 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, m := n is not a monadic
operator [R.3.0.4.a, 4.2.1.f, 4.3.1.c]. Think about n2n. a) Yes.
no (ref file, int) create = create.

8.4 a) (real a = skip ; int : round a) * b) No, remand
possesses a routine which has no parameters. c) Yes. d) Yes. e) No, m:n 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 mahan is that of a monadic-indication, and does not identify the first. c) In order to restate 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) c(real a = skip ; bool : a > 0)n. b) n(real a = x ; bool : a > 0)n.

8.9 a) m = 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 mn to or from nproc into. d) Yes. e) No, there is a multiple definition of m-n.

8.11 a) It draws a straight line of length ndm in the direction S. b) Try, mn, s, e, w. c)!

8.12 a) Remove 2, remove 1. b) Remove 1, remove 3, replace 1, replace 2, remove 1. c) The formula requires that m:n should be a variable. d) Remove 2, remove 1, remove 4, replace 1, replace 2, remove 1, replace 1, replace 2, remove 1, replace 2, remove 1. e) Try nproc_upn and nproc_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.8.8.5.2.1.a]. e) real.


9.6 a) No. b) only procedure- or row-of.

9.7 a) void-b) virtual NONSTO FORM d) strongly united to MOD.

10.1 a) No, 1.1.5.b. b) No, 1.1.5.b. c) R.9.2.b.

10.2 a) struct def a vu or def 'a' : contains mn with [R.7.1.2.c]. c) def ccall c). d) struct ccall left operand, triple, ref call, title, ref book next.

10.3 a) The first indication and the virtual-declarer are two are global- and next of a := link

10.4 a) No. b) only.

10.5 a) If mn is formula and ref indication, then rower.

10.6 a) Yes. b) e) Yes.

10.7 a) Yes. b) "jim", node := (nil

10.8 a) uleft (nil)n. b) BOB. c) 

10.9 a) In line become nfr ; b :=

10.11 a) nproc nfr | pl(left of root ; print("m")) n. b) left of root :=
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 REPETY row of MODE FORM•. e) •STIRM• united to •MOD FORM•.

10.1 a) No, a realm is not a •mode-indication• [R.4.2.1.b, 1.1.5.b]. b) No, an is an •identifier•, not an •indicant•. c) No, a realm is not an •actual-declarer•. d) Perhaps, if a realm already specifies a united mode [R.7.1.1.cc, 9.2.b]. e) Yes [R.9.2.b].

10.2 a) nostruct(ref b a, proc b d)u b) This is undefined. In uref a v or uref v = loc ref au, the •generator• aloc ref au: contains an which is virtual and is therefore not developed [R.7.1.2.c]. c) union(ref const, ref var, ref triple, ref call)u, d) nostruct(union(ref const, ref var, ref triple, ref call)left operand, int operator, union(ref const, ref var, ref triple, ref call)right operand)n. e) nostruct([1:0 flex] char title, ref book next)n.

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) plink a := (1, nil); next of a := link := (2, nil); next of next of a := link := (3, nil). e) No [R.6.2.1.f].

10.4 a) No. b) Yes. c) No. d) Yes. e) Yes.

10.5 a) If an is a •dyadic-indication•, then it is a •formula• and ob : un is a •cast•; if an is a •mode-indication•, then it is a •declaration• and ob : un is a •row-ofrow•.

10.6 a) Yes. b) No. c) nostruct a = (int u, ref a v)n. d) No. e) Yes.

10.7 a) Yes. b) node tree := (node := (nil, "bob", nil), "jim", node := (nil, "sas", nil))n.

10.8 a) left of right of tree := node := (nil, "ron", nil)n. b) Bob. c) •false•. d) •true•. e) •false•, •true•.

10.9 a) In line 2, insert bool b := true; lines 7 and 8 become a[i] := b := false; done := bn.

10.11 a) nproc p1 = (ref node root) : (print("("); (root #: nil | p1(left of root) ; print(val of root) ; p1(right of root)) ; print(")"))n. b) noproc p2 = (ref node root) : (root #: nil | left of root := (ref node : nil) and right of root := (ref
node : nil) | print(val of root) | print("\n") | p2(left of root) | print("\n") | print(val of root) | print("\n") | print(right of root) | print("\n") | n.

10.12  a) Remove \texttt{naction(p);} from line 12 and insert it in line 8.

11.2  a) No, \texttt{uprint} has only one parameter. b) No, \texttt{uniln} 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) [R.10.5.1.1.e, 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 \texttt{uprinttypes}. b) No [R.8.2.4.1.b]. c) Yes, dereference to \texttt{pref \_real}, unite to \texttt{mintypep} and then row it. d) Undefined, since \texttt{usm} cannot be coerced to \texttt{nouttypep}. e) No, \texttt{ufomata} cannot be coerced to \texttt{treadtypep}.

11.4  a) ? +5. b) ? ABC. c) Twice dereferenced and then united to \texttt{uprinttypep}, ? $+3.400000E +0$. d) Four and 9 spaces left over. e) Nine and 2 spaces left over. f) A B C.

This document must be returned and paid for within 30 days. Please send your check and bill me.

Name: ____________________________
Address: ____________________________
left of root)
int(right of
 insert it in

No, until can
ited, it is
10.5.1.1.f,
¥0.
ent is void
open. b) No
unite to
un cannot be
coerced to
anced and then
end 9 spaces

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 ......................................

............................................