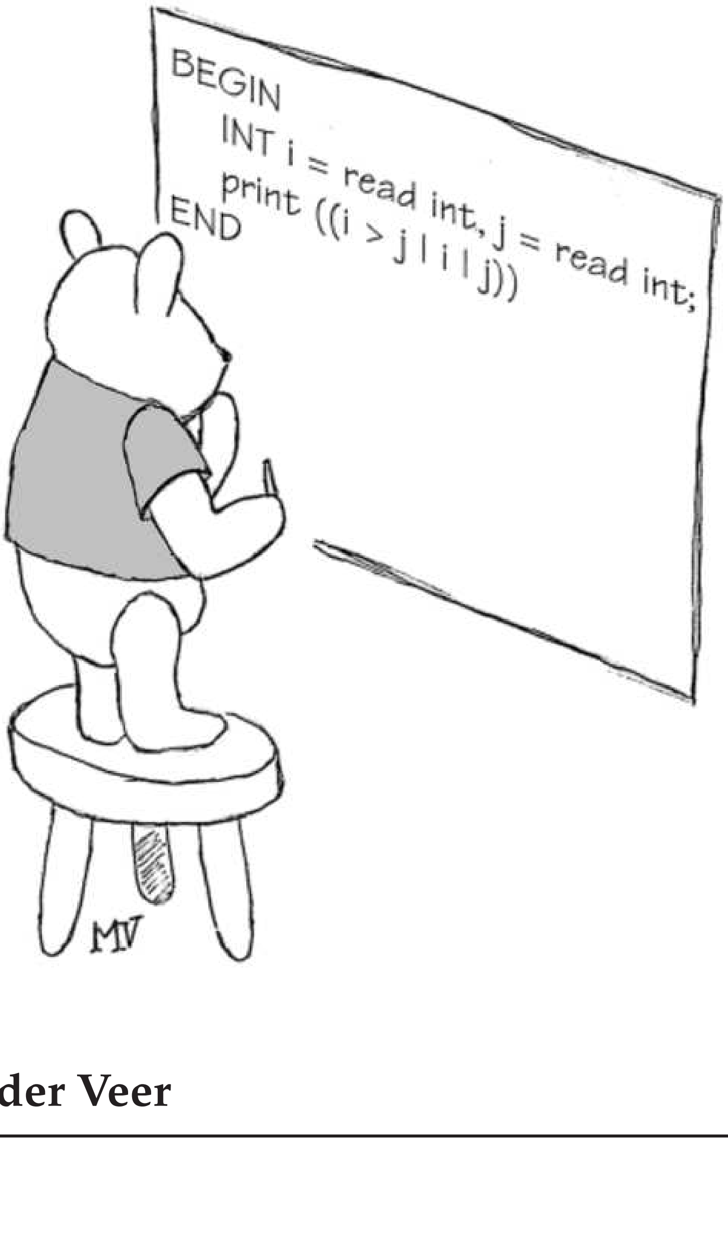
Algol 68 Genie User Manual

Algol 68 Genie Mark 14.1

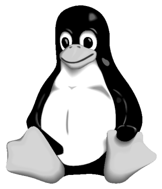


Marcel van der Veer

Algol 68 Genie User Manual

Algol 68 Genie Mark 14.1

Marcel van der Veer



Algol 68 Genie is being developed on GNU/Linux, and *Algol 68 Genie User Manual* is prepared on this operating system using LATEX.

{"... Languages take such a time, and so do
all the things one wants to know about."
The Lost Road. John Tolkien. }

Table of contents

Foreword

This manual is distributed with Algol 68 Genie (*a68g*), an open source Algol 68 interpreter that can be used for executing Algol 68 programs or scripts. This manual documents how to use Algol 68 Genie, as well as its features and incompatibilities, and how to report bugs. It corresponds to Algol 68 Genie Mark 14.1. This manual also provides an informal introduction to Algol 68. The internals of Algol 68 Genie are not documented in this manual.

Algol 68 has been around for four decades, but some who "rediscovered" it in recent years, well aware of how the world has moved on, had a feeling best described as *plus ça change, plus c'est la même chose*; presumably also because much of today's programming is still done using older third-generation languages. Though not a big success in its days, Algol 68 introduced innovations that are actual up to the present. Its expressive power, and it being oriented towards the needs of programmers instead of towards the needs of compiler writers, may explain why, since Algol 68 Genie became available for download, many people appeared to be interested in an Algol 68 implementation, the majority of them being mathematicians or computer scientists. Hence it is expected that there are also people interested in having access to documentation on Algol 68, but nowadays most books on Algol 68 are out of print. Even if one can get hold of a book on Algol 68, it will certainly not describe Algol 68 Genie since this implementation is considerably younger than those books.

The formal defining document, the Algol 68 Revised Report, is also out of print but it is available on the internet in various formats; for instance a HTML version comes with the distribution of Algol 68 Genie. Those interested in the design and formal specification of programming languages should at an appropriate moment study the Revised Report, but it is known to rank among the difficult publications of computer science and is therefore not suitable as an *informal* introduction. In fact, it has been said that at the time Algol 68 was presented, the complexity of the Revised Report made some people believe that the language itself would also be complex, but after reading this manual you will probably conclude that Algol 68 is a rather lean language.

I did not have to write all text from scratch allowing me to spend more time on developing the interpreter *a68g*. This manual contains smaller or larger parts from different open sources that have been edited and blended with *a68g* documentation to form a consistent, new publication. Sian Leitch generously made the TEX source of *Programming Algol 68 Made Easy* available under the GNU General Public License (see section *Bibliography*) through which a comprehensive informal introduction to Algol 68 was at hand. Material has been extensively edited as to let it describe Algol 68 Genie. For instance *a68g* implements partial parametrisation, the parallel-clause as well as formatted transput, for which documentation had to be written. Also, with respect to Sian's book, the section on recursion has been extended but a section on program construction has been left out since software construction is not in the scope of this manual.

I admit that there is some unevenness in the level of this manual. This manual is not an introduction to programming. Neither is it meant as a text for a course in computer science. I assume you are already comfortable with computers and programming. You will not find explicit questions or problems: I assume you are creative enough to think up many ”what is the outcome of this expression” or ”write some program that applies what you just read” exercises. Would you want explicit exercises, then Sian's book is a rich source of them. I have consciously included mathematical content since it will either exactly specify the operation of some routine or operator, or demonstrate some Algol 68 construction. I think that mathematicians will find the mathematics quite trivial while those not versed in the art will understand the demonstration nonetheless.

Acknowledgements

Thanks go to the following people, in alphabetical order, who were kind enough to report bugs and obscurities, to propose improvements, provide documentation, or to make a contribution in another way: Mark Alford, Bruce Axtens, Lennart Benschop, Jaap Boenders, Bart Botma, Paul Cockshott, Huw Davies, Neville Dempsey, Koos Dering, Scott Gallaher, Jeremy Gibbons, Dick Grune, Norman Hardy, Chap Harrison, Jim Heifetz, Patrik Jansson, Trevor Jenkins, Rob Jongschaap, Henk Keller, Wilhelm Kloke, Has van der Krieken, Sian Leitch, Karolina Lindqvist, Charles Lindsey, Richard O'Keefe, Lawrence D'Oliveiro, France Pahlplatz, Steven Pemberton, Richard Pinch, Henk Robbers, Rubin Simons, Robert Uzgalis, Bruno Verlyck, Nacho Vidal García, Merijn Vogel, Eric Voss, Theo Vosse, Andy Walker, Jim Watt, Glyn Webster and Tom Zahm.

Algol 68 Genie would not be what it is without their help.

Marcel van der Veer
Uithoorn
November 2008

Biography

Marcel van der Veer from The Netherlands holds a MSc in Chemistry from the University of Nijmegen and a PhD in Applied Physics from the University of Twente.

During his academic years, he worked with *ALGOL68C*, *FLACC* and *ALGOL68RS*. Probably due to the fact that in chemistry - and physics departments people tend to take a pragmatic approach towards computer science, Marcel was undeterred to take on an Algol 68 implementation for use at home when Algol 68 implementations were phased out as mainframes were replaced by UNIX work-stations which offered C, FORTRAN or Pascal, but not Algol 68.

Photo © Nacho Vidal García 2006.

Development of Algol 68 Genie started 1992. After encouragement by some people (especially Henk Keller) Marcel decided to release Algol 68 Genie under GNU GPL and make it available for download in 2001.

Chapter 1

Preliminaries

At the time of Algol 68's introduction compilers usually were made available on mainframes by computing centres. This photo shows the operator console for an IBM 370 mainframe at NASA which was used throughout the 1980's. The IBM 370 could run ALGOL68C under MVT, MVS and CMS, and FLACC under MVS, CMS and MTS.

Photo from the NASA Glenn Archives.

Source: http://grcimagenet.grc.nasa.gov/.

{"... Lisp and Algol, are built around a kernel that
seems as natural as a branch of mathematics."
Metamagical Themas. Douglas Hofstadter. }

1.1 A brief history of Algol 68

Algol, **algo**rithmic **l**anguage, is a family of imperative computer programming languages which greatly influenced many other languages and became the *de facto* way algorithms were described in textbooks and academic works for almost three decades. There are three consecutive major specifications:

1. Algol 58, originally known as IAL (International Algorithmic Language)
2. Algol 60, revised in 1963
3. Algol 68, revised in 1976

Algol 68 is a rather lean orthogonal general-purpose language that is a beautiful means for denoting algorithms. Algol 68 was designed by IFIP Working Group 2.1 (Algorithmic Languages and Calculi) that has continuing responsibility for Algol 60 and Algol 68. IFIP, the International Federation for Information Processing is an umbrella organisation for national information processing organisations. It was established in 1960 under the auspices of UNESCO. IFIP organises a number of events, including conferences, working groups and workshops, and general assemblies.

In the early 1960's Working Group 2.1 (WG 2.1) started discussing the development of Algol X, a successor to Algol 60. At its 1965 meeting in Princeton, WG 2.1 invited written descriptions of the language based on the previous discussions. At the meeting in St. Pierre de Chartreuse, also in 1965, three reports describing more or less complete languages, by Wirth [1] , Seegmüller, and Van Wijngaarden, were amongst the contributions. Other significant contributions were papers by Hoare and Naur. Van Wijngaarden's paper *Orthogonal design and description of a formal language* (available from [Karl Kleine's collection]) featured a new technique for language design and definition. This paper formed the basis for what would develop into Algol 68.

Adriaan van Wijngaarden (1916 - 1987) is considered by many to be the founding father of computer science in The Netherlands. In 1947 he became head of the computing department of the than new Mathematisch Centrum (MC) in Amsterdam; he directed the MC from 1961 to 1981. He was cofounder of IFIP and one of the designers of Algol 60 and later Algol 68. As leader of the Algol 68 committee, he made a profound contribution to the field of programming language design, - definition and - description.

[1]Wirth left WG 2.1 to work with Hoare on Algol W which influenced the later language Pascal.

Algol 68 has had a large influence on the development of programming languages. It introduced new ideas, for example orthogonality, a strong type system, procedures as types, storage allocation, treatment of arrays, a rigorous description of syntax, and parallel processing, but also ideas that have caused debate over the years such as coercions depending on subtle context rules and quite complicated input-output formatting.

Algol 68 was defined in a formal document, first published in January 1969, and later published in Acta Informatica and also printed in Sigplan Notices. A Revised Report was issued in 1976. The distribution of Algol 68 Genie Mark 14.1 contains a HTML translation of the Revised Report. Algol 68 was the first major language for which a full formal definition was made before it was implemented. Though known to be terse, the Revised Report does contain humour *solis sacerdotibus* — to quote from [Koster 1996]: ". . . The strict and sober syntax permits itself small puns, as well as a liberal use of portmanteau words. Transput is input or output. 'Stowed' is the word for structured or rowed. Hipping is the coercion for the hop, skip and jump. `MOID` is `MODE` or `void`. All metanotions ending on `ETY` have an empty production. Just reading aloud certain lines of the syntax, slightly raising the voice for capitalized words, conveys a feeling of heroic and pagan fun . . . Such lines cannot be read or written with a straight face."

Probably because of the formal character of the Revised Report, which requires study to comprehend, the misconception got hold that Algol 68 is a complex language, while in fact it is rather lean. Algol 68's grammar is a two-level W-grammar ('W' for Van Wijngaarden) that can define a potentially infinite grammar in a finite number of rules. It can rigorously define some syntactic restrictions that otherwise have to be formulated in natural language, despite their essentially syntactical content; for example, demanding that applied identifiers or - operators be declared, or demanding that modes result in finite objects that require finite coercion, et cetera. In that respect, a context-free syntax needs extra rules formulated in natural language to reject incorrect programs. This is less elegant, but defining documents for programming languages with a context-free grammar indeed look less complex than the Algol 68 (Revised) Report — see for instance the context-free grammar of Algol 68 Genie.

Another factor that hindered Algol 68 from spreading rapidly is that the language was designed for programmers, not for compiler writers, in a time when the field of compiler construction was not as advanced as it is today. Implementation efforts based on formal methods generally failed; Algol 68's context-dependent grammar required some invention to parse (consider for instance `x(y, z)` that can be either a call or a slice depending on the mode of `x`, while `x` does not need to be declared before being applied). Algol 68 Genie employs a multi-pass scheme to parse Algol 68 [Lindsey 1993], see section 9.13.

At the time of Algol 68's presentation compilers usually were made available on mainframes by computing centres, which may explain why Algol 68 was popular in locations rather than areas, for instance Amsterdam, Berlin or Cambridge. It appears that Algol 68 was relatively popular in the UK, where the *ALGOL68RS* and *ALGOL68C* compilers were developed. An important clue to understanding why Algol 68 did not become a widely used language lies in the fact that industry did not pick it up and that initiatives to commercialise Algol 68 compilers were relatively unsuccessful, for instance the *FLACC* compiler sold just twenty-two copies [2] (but these served twenty-two mainframes, thus probably thousands of users had access to it). The market for universal programming languages evidently did not develop as hoped for during the decade in which the language was developed and implemented.

Algol 68 can be placed in the history of programming languages more or less like in this graph:

```
1955                       FORTRAN
                            |
1958   Algol 58            FORTRAN II (and later IV/66, 77, 95)
        |
1960   Algol 60 ---------------------------------
        |                   |                   |
1962    |                  SIMULA               |
        |                   |                   |
1963   Algol 60 (Revised)   |                   |
        |                   |                   |
1967    |                  SIMULA 67            |
        |                                       |
1968   Algol 68                                Algol W
        |                                       |
1970    |                                      Pascal (and later Modula, Oberon)
        |
1976   Algol 68 (Revised)

1983                       Ada (and later Ada 95, 2005)
```

Few claim that Ada is Algol 68's successor but most dispute that. Therefore Ada is mentioned, but there is no line drawn from Algol 68 to Ada.

Algol 68 lives on not only in the minds of people formed by it but also in languages as C and C++, even though the orthogonality in the syntax, elegance and security have been mostly lost. The language is still referred to in teaching material, discussions and publications. Because of the rigour and precision of its definition, one could argue that those interested in the design and formal specification of programming languages, such as students of computer science, should at an appropriate moment study Algol 68 to understand the influence it had, irrespective of whether the language did or did not spread widely at the time it was presented. To preserve the language is of historical, educational, and therefore of cultural importance.

For programmers, the world has of course moved on, but the reactions on Algol 68 Genie suggest that many people who have seriously programmed in Algol 68 in the past, only moved to other programming languages because the Algol 68 implementations they were using were phased out.

[2]Source: Chris Thomson, formerly with Chion Corporation, on `comp.lang.misc` [1988].

1.2 Algol 68 Genie

The language described in this manual is that implemented by the Algol 68 Genie interpreter available from

http://www.xs4all.nl/~jmvdveer/algol.html

Algol 68 Genie implements almost all of Algol 68, and extends that language. To run the programs described in this manual you will need a computer with a Linux or BSD (for example OpenBSD, FreeBSD or MacOS X) system. Chapter 10 describes how you can install the Algol 68 Genie interpreter *a68g* on your system, and how you can use it.

1.3 Terminology

Small parts of program text is typeset in fixed-space font like this:

```
PR precision=1000 list PR
INT i := read int;
print (i)
```

while larger program fragments or routines are in a smaller fixed-space font, with line numbers in the margin that are for reference purposes only and are not part of the program text:

```
1  #
2  Takeuchi's Tarai (or Tak) function. Moore proved its termination.
3  See http://mathworld.wolfram.com/TAKFunction.html
4  #
5
6  PROC tarai = (INT i, j, k) INT:
7       IF i <= j
8       THEN j
9       ELSE tarai (tarai (i - 1, j, k),
10                  tarai (j - 1, k, i),
11                  tarai (k - 1, i, j))
12      FI;
13
14 tarai (1, 2, 3)
```

Algol 68 terminology tends to be pedantic. This is a consequence of the rigorous description of Algol 68, which is very precise. Here and there I invent alternative terminology for that in the Revised Report. For instance, in Algol 68 there is a strict difference between a `MODE` and `VOID` while in this manual modes include `VOID` but I indicate where you cannot specify `VOID` as a mode. Also, I use the term ”statement” meaning either ”declaration” or ”(part of a) unit”.

This text contains references that are listed in the *Bibliography*. A format [Mailloux 1978] means the entry referring to work of Mailloux, published in the year 1978. On various places you will see a reference to the Revised Report formatted as for example [RR 5.2.3.2] referring you to chapter 5, section 2.3.2 of the Revised Report. An indication `AB39.3.1` means `ALGOL BULLETIN`, volume 39, article 3.1, which are still available on-line.

Although the text is an informal introduction to Algol 68 Genie, many subjects can be placed in a wider context. To include more in-depth material without disrupting the argument, annotations have been placed in the text which are indented and have an icon depicting their meaning:

⚠ These notes give additional or advanced information on the topic at hand.

⚠ These notes indicate a potential problem or pitfall.

1.4 Organisation of this manual

This text is meant as an informal introduction to Algol 68 Genie. The approach adopted here is that of other texts on programming languages that explain a minimal set of features early on with which one can program simple exercises; in due course this basis is elaborated thus dispersing the elaboration of ideas over a number of chapters. Therefore the chapters should be read in order, but it is not possible to explain Algol 68 in a strict ”left-to-right” manner. Hence I sometimes make forward references and leave it up to you to first read the referenced material.

Further chapters are organised as follows:

Chapter 2 *Values, names and modes* introduces standard modes, denotations, values and names. The name-value duality is a central concept in Algol 68. A name refers to a value, and a name can itself be a value. Nowadays this sounds familiar (variables, pointers and pointer variables) but at the time Algol 68 was presented this was new, and the strong typing of Algol 68 helps you avoid the many pitfalls involved in programming using pointer variables.

Chapter 3 *Formulas* explores formulas, which are statements in which operators work on operands. The most common example is an arithmetic expression, but Algol 68 lets you write formulas operating on any mode you declare.

Chapter 4 *Program structure* describes conditional and case statements that let you control program flow depending on the value of boolean - or integer conditions. It also describes loops.

Chapter 5 *Stowed modes* describes ordered sets of values: rows and structures. It explains how to extract sub-rows from a row, how to select a diagonal in a matrix, et cetera. This chapter also shows how to group objects into structures. `STRING` and `COMPLEX` are introduced.

Chapter 6 *Routines* explains how to declare procedures and operators, both having a routine-text as "denotation". This chapter brings together recursion and data structures and is a demonstration of Algol 68's expressive power. This chapter also describes recursion and partial parametrisation. *a68g* is the only Algol 68 implementation ever to implement partial parametrisation.

Chapter 7 *United modes* completes the description of constructing modes by introducing the united mode. It also gives the rules for a well-formed mode.

Chapter 8 *Transput* is about transput which is an Algol 68 term for input-output. Formatted transput is described in this chapter.

Chapter 9 *Units and coercions* describes units and coercions in detail. This lets you exploit the full potential of the language.

Chapter 10 *Installing and using Algol 68 Genie* describes the Algol 68 Genie interpreter *a68g*, how to install it on your computer system and how to use the program.

Chapter 11 *Prelude* is an extensive description of the standard-prelude and library-prelude. Standard Algol 68 predefines a plethora of operators and procedures. Algol 68 Genie predefines many operators and procedures in addition to those required by the standard-prelude, that form the library-prelude. This chapter documents these extensions.

Chapter 12 *Example programs* lists a number of programs to demonstrate the material covered in this manual.

Appendix A *Reporting bugs* gives information on how and where to report bugs in Algol 68 Genie.

Appendix B *Algol 68 Genie context-free syntax* provides a quick reference for Algol 68 Genie syntax.

Appendix C *Release history* describes the history of Algol 68 releases.

Appendix D *Manual page* is the text for the manual page for *a68g*.

Appendix E *ASCII table* is for reference purposes.

Appendix F *GNU General Public License* is a copy of *a68g*'s license.

Appendix G *GNU Free Documentation License* is a copy of *a68g*'s documentation license.

Appendix H *Bibliography* has references and suggestions for further reading.

Chapter 2

Values, names and modes

{"For the things we have to learn before
we can do them, we learn by doing them."
Aristotle, Nichomachean Ethics}

2.1 Values and modes

Three of the guiding principles of Algol 68 are the concepts of *value*, *name* (or *variable*) and *mode*. In other programming languages as for instance Pascal these terms would be *constant*, *variable* and *type*. An Algol 68 program manipulates values to produce new values thus performing a task that suits you. Values are such entities as numbers and letters, but you will see in later chapters that values can be compounded and complicated.

A value has unique mode. For example, the whole number represented by the digit 1 has mode `INT`. The symbol `INT` is called a mode-indicant. A mode-indicant is a declarer. In later chapters you will see that declarers can be compounded. The representation of a value in Algol 68 is called a denotation because it denotes a value. Note that the denotation of a value is not the same as the value itself. For example, if you write 1, 1 and 1 you have three separate denotations denoting a single value, the integer "one". Instead of saying "the value of mode `INT`", we will sometimes say "the `INT` value" or even informally "the `INT`".

You will be meeting many more mode-indicants in the course of this manual and they are all written in capital letters. The strict definition of a mode-indicant is that it consists of a series of one or more capital letters. No intervening spaces are allowed, though *a68g* allows intervening underscores. There is no limit to the length of a mode-indicant. These are some mode-indicants which you will meet in this manual:

```
INT
```

```
REAL
BOOL
CHAR
COMPLEX
STRING
FILE
TREE
NODE
```

Section 5.20 explains how you can define your own mode indicants. Although you can use any sequence of valid characters, meaningful mode-indicants can help you to understand your programs.

2.2 Integers

In Algol 68 ”integers” are whole numbers, elements of $\mathbb{Z}$ (but not the other way round; not all elements of $\mathbb{Z}$ are Algol 68 ”integers” since a computer is a finite object). If you want to write the denotation of an integer in Algol 68, you use digits 0 to 9. A sign is not a part of an integer-denotation. Hence an integer-denotation is in $\mathbb{W}$, not in $\mathbb{Z}$. A sign is a monadic operator, so if you write -1 or $+1$ you will have written a formula, see chapter 3. Although you cannot use commas or decimal points in an integer-denotation, spaces can be inserted anywhere. Here are some examples of integer-denotations:

```
3
0003
3000001
2 147 483 647
```

The mode of each of these denotations is `INT`. Note that 3 and 0003 denote the same value 3 because leading zeroes are not significant. The following are not integer-denotations:

```
4,096
-2
1e6
```

The first contains a comma, the second is a formula consisting of a monadic-operator followed by an integer-denotation, and the third contains an exponent starting with the letter `e` and thus is a real-denotation.

Although mathematically speaking there is no largest positive integer in $\mathbb{W}$, a computer represents values in a finite number of bits hence the magnitude of an ”integer” is limited. On a 32-bit processor the largest positive integer which can be manipulated by *a68g* is $2^{31} - 1$, and the largest negative integer is $-(2^{31} - 1)$. On a 64-bit processor these values are $2^{63} - 1$ and

$-(2^{63}-1)$ respectively. Algol 68 predefines many values in what is called the standard-prelude (see chapter 11). One predefined identifier in Algol 68 is `max int` that tells you the largest representable integer on the platform on which your program runs.

However, sometimes you want to use integer values larger than `max int`. To that end Algol 68 Genie supports modes `LONG INT` and `LONG LONG INT`. The range of `LONG LONG INT` is default twice the length of `LONG INT` but can be made arbitrary large through the intepreter-option `precision` (see section 10.7.4). Here are the respective maximum values for the three integer lengths available in *a68g*:

identifier	value	
`max int`	$2^{31}-1$	32-bit Pentium
`long max int`	$10^{35}-1$	
`long long max int`	$10^{70}-1$	

In standard Algol 68, a denotation for `LONG INT` must be preceded by the keyword `LONG` and a denotation for `LONG LONG INT` must be preceded by the keyword `LONG LONG`. For instance

```
LONG 34359738368
LONG LONG
   3930061525912861057173624287137506221892737197425280369698987
```

the first value being 2^{35} and the second being 3^{127}.

The *a68g* interpreter relaxes the use of `LONG` prefixes when the context imposes a mode for a denotation, in which case a denotation of a lesser precision is automatically promoted to a denotation of the imposed mode.

The routine `print` will print among others an integer argument, per default to standard output, as in

```
print(max int)
print(LONG 281474976710656)
```

Of course `print` can print multiple arguments in a single call. When printing (or reading) multiple arguments, the arguments have to be enclosed in parentheses, as in

```
print((max int, LONG 281474976710656))
```

The reason for this peculiarity is that `print` takes as argument a row of printable values. In chapter 5 you will see that a row of more than one value is denoted by writing the values as a comma-separated list enclosed by either `(` ... `)` or `BEGIN` ... `END`. The Algol 68 term for an enclosed comma-separated list of units is collateral-clause but when we use it in this manual to denote a row, we will call it a row-display.

Finally, if you want to continue printing a new line, you specify `new line`, as in

```
print((max int, new line, LONG 281474976710656))
```

2.3 Identity-declarations

Suppose you want to use an integer with denotation

```
42
```

in various parts of your program. Algol 68 provides a construct which enables you to declare a synonym for a value (in this case, an integer denotation). It is done by an identity-declaration which is used widely in the language. Similar constructions in other languages are `CONST` declarations in Pascal, `PARAMETER` statements in FORTRAN, or `#define` in the C preprocessor. Here is an identity-declaration for the integer mentioned at the start of this paragraph:

```
INT measurements done = 42
```

Whenever you want to use 42, you write

```
measurements done
```

in your program.

An identity-declaration consists of four parts:

```
formal-declarer identifier = unit
```

You have already met the mode-indicant; the standard mode indicants can be used as a formal-declarer. The formal-declarer cannot be `VOID`. The difference between a formal-declarer and actual-declarer will be explained later on. A unit is an Algol 68 statement yielding a value; you will learn all about them in the course of this text but for now we will limit ourselves to denotations. An identifier is a sequence of one or more characters which starts with a lower-case letter and continues with lower-case letters or digits. It can be broken-up by spaces or tab characters. *a68g* allows underscores in identifiers, but an underscore is part of the identifier unlike white space. Some examples of valid identifiers are:

```
i
rate 2 pay
eigen value 3
```

The following are not identifiers:

```
2pairs
escape.velocity
XConfigureEvent
```

The first starts with a digit, the second contains a character which is neither a letter nor a digit, and the third contains capital letters.

An identifier looks like a name, in the ordinary sense of that word, but we do not use the term ″name″ because it has a special meaning in Algol 68 which will be explained later on.

In Algol 68, no white space is required when no ambiguity exists in the meaning of concatenated terms

```
INTmeasurementsdone=42
```

but it is of course good practice to use white space to improve clarity.

The third part in an identity-declaration is the equals-symbol `=`. The fourth part (the right-hand side of the equals-symbol) is a unit yielding a value. You will see later that the value can be any piece of program which yields a value of the mode specified by the mode-indicant. So far, we have only met integer-denotations.

There are two ways of declaring identifiers for two integers:

```
INT i = 2; INT j = 3
```

The semicolon `;` is called the go-on symbol. We can abbreviate the declarations as follows:

```
INT i = 2, j = 3
```

The comma separates the two declarations, but does not mean that `i` is declared first, followed by `j`. It is up to *a68g* to determine which declaration is elaborated first. They could even be done in parallel. This is known as collateral elaboration, as opposed to sequential elaboration determined by the go-on symbol (the semicolon). We will be meeting collateral elaboration again in later chapters.

2.4 Reals

In Algol 68 ″reals″ are numbers, elements of $\mathbb{R}$ (but not the other way round — not all elements of $\mathbb{R}$ are ″reals″ since a computer is a finite object). Thus numbers which contain fractional parts, such as $1/7$, or 1.5, or numbers expressed in scientific notation, such as 1.38×10^{-23} are values of mode `REAL`. Reals are denoted by digits and at least one of (A) the decimal point which is denoted by a point, or (B) an exponent starting with letter `e`. The `e` is a times-ten-to-the-power-symbol. For example, `e-9` means $\times 10^{-9}$. Just as with integers, there are no denotations for negative reals. The exponent can however be preceded by a sign. In the following `REAL` denotations, the third denotation has the same value as the fourth:

```
4.5
```

```
.9
0.075
7.5e-2
1e6
```

Some identity-declarations for values of mode `REAL` are:

```
REAL e = 2.718 281 828,
     electron charge = 1.6021e-19,
     mils added = 1.0
```

Real values on computers have to be represented with a finite number of bits and therefore have finite precision and finite magnitude. The largest `REAL` which *a68g* can handle is declared in the standard-prelude as identifier `max real`. Also, there is a value `small real` that denotes the smallest value that when added to 1.0, yields a value larger than 1.0, and thus is a measure of precision.

However, sometimes you want to use real values with more decimals than offered by `REAL`. Algol 68 Genie supports modes `LONG REAL` and `LONG LONG REAL`. The range of `LONG LONG REAL` is default circa twice the length of `LONG REAL` but can be made arbitrary large through the intepreter-option `precision`, see section 10.7.4. Here are the respective limiting values for the three reals lengths available in *a68g*, which were chosen under the observation that most multi-precision applications require 20-60 significant digits:

identifier	value	
`max real`	$1.79769313486235 \times 10^{308}$	IEEE-754, Pentium
`long max real`	1×10^{999999}	
`long long max real`	1×10^{999999}	
`small real`	$2.22044604925031 \times 10^{-16}$	IEEE-754, Pentium
`long small real`	1×10^{-28}	
`long long small real`	1×10^{-63}	

In standard Algol 68, a denotation for `LONG REAL` must be preceded by the keyword `LONG` and a denotation for `LONG LONG REAL` must be preceded by the keyword `LONG LONG`. For instance

```
LONG 2.7182818284590452353602874 71
LONG LONG
   0.7071067811865475244008443621048490392848359376884740365883398 69
```

the first value being e and the second being $\frac{1}{2}\sqrt{2}$.

As with integer denotations, the *a68g* interpreter relaxes the use of `LONG` prefixes when the context imposes a mode for a denotation, in which case a denotation of a lesser precision is automatically promoted to a denotation of the imposed mode.

The value of π is declared in the standard-prelude with the identifier `pi` with the three precisions provided:

```
REAL pi = 3.14159265358979
LONG REAL long pi = 3.14159265358979323846264338 3
LONG LONG REAL long long pi =
   3.14159265358979323846264338327950288419716939937510582097494459
```

The routine `print` will print a real argument, per default to standard output, as in

```
print (pi)
print (LONG 1.73205080756887729352744634 2)
```

It was mentioned above that in an identity-declaration, any piece of program yielding a value of the required mode can be used as the value. Here, for example, is an identity-declaration where the value has mode `INT`:

```
REAL a = 3
```

However, the mode required is `REAL`. In certain contexts, a value of one mode can be coerced to a value of another mode. There are five types of context in Algol 68: strong, firm, meek, weak and soft. Depending on the type of context, a value can be changed by a small set of coercions The right-hand side of an identity-declaration is a strong context. In a strong context, the resulting mode is always imposed (in this case by the formal-declarer on the left-hand side). One of the strong coercions is widening that can for instance widen a value of mode `INT` to a value of mode `REAL`.

You can also supply an identifier yielding the required mode on the right-hand side. That is just another example of a unit yielding a value of the required mode. Here are two identity-declarations:

```
REAL one = 1.0;
REAL start = one
```

You cannot combine these two declarations into one with a comma as in

```
REAL one = 1.0, start = one
```

because you do not know which identity-declaration gets elaborated first (because the comma implies collateral elaboration). If *a68g* executes a declaration as the one above, it may or may not end in a runtime error since an uninitialised value may be accessed.

2.5 Booleans

The mode `BOOL` is named after George Boole, the nineteenth century mathematician who developed the system of logic which bears his name. There are only two values of mode `BOOL`, and their denotations are `TRUE` and `FALSE`. The routine `print`, when fed with boolean values prints `T` for `TRUE`, and `F` for `FALSE`, with spaces neither before nor after. Thus

```
BOOL t = TRUE, f = FALSE;
print((t, f));
```

produces `TF` on standard output. Boolean values are also read as `T` and `F` respectively.

2.6 Characters

All single symbols used to compose text are characters. The alphabet consists of the characters `A` to `Z` and `a` to `z`. The digits comprise the characters 0 to 9. The character set recognised by *a68g* is 7-bit ASCII. The mode of a character is `CHAR`. A character is denoted by placing it between quote characters. Thus the denotation of the lower-case a is `"a"`. Here are some character denotations:

```
"a"  "A"  "3"  ";"  "\"  "′"  """"  " "
```

The third denotation is `"3"`. This value has mode `CHAR`. The denotation 3 has mode `INT`: the two values are quite distinct, and one is not a synonym for the other. The last denotation is that of the space character. The quote character `""""` is doubled in its denotation.

Some identity-declarations for values of mode `CHAR` are:

```
CHAR a = "A", zed = "z", tilde = "˜"
```

Note that you cannot write

```
CHAR z = CHAR zed = "z"
```

since an identity-declaration is a not a unit. When a unit is elaborated, it will yield a value. That is, after elaboration, a value will be available for further use if required. Identity-declarations do not yield a value.

Here is a piece of program with identity-declarations for an `INT` and a `CHAR`:

```
INT ninety nine = 99, CHAR x = "X"
```

The interpreter recognises `1 + max abs char` distinct values of mode `CHAR`, some of which cannot be written in denotations (control characters for example). The space is declared as the identifier `blank` in the standard-prelude.

Values of modes `INT`, `REAL`, `BOOL` and `CHAR` are known as plain values.

In chapter 5 you will see that characters can be organised in rows to form texts. For the moment we will only introduce the row-of-character-denotation for denoting texts, that is conventionally delimited by quote characters:

```
"This is a row-of-character-denotation"
```

In Algol 68, two adjacent quote characters are used to introduce a quote character into a row of character denotation. You can of course print row-of-character-denotations, so you can now let your program print descriptive texts:

```
print ("Oops! Too few experiments performed");
```

2.7 Simple read - and write routines

This is a good point to introduce to you four identifiers that are useful even in the simplest of programs:

1. `read int` reads and yields an integer value from standard input (if you did not redirect input, this would be your keyboard),
2. `read real` reads and yields a real value from standard input,
3. `read bool` reads and yields a bool value from standard input,
4. `read char` reads and yields a character value from standard input.

These identifiers come from *ALGOL68C*. In chapter 6 you will see that above identifiers actually are procedures. On a right-hand side of an identity-declaration, the strong context forces these routines to yield a value of `INT`, `REAL`, `BOOL` or `CHAR` respectively by a coercion called deproceduring.

2.8 Program structure

Algol 68 programs are written in free format. The meaning of your program is independent of the layout of the source program. An example of the contrary is old-style FORTRAN where the first five columns in a line are reserved for a numerical label, a non-blank sixth column indicates a continued source line, and the source line itself runs from columns 7-72. Columns 73-80 were ignored by a compiler but were normally used to punch sequence numbers in cards so if you would drop a deck, you could restore it.

For example, you could write a trivial Algol 68 program like this:

```
(print(max int))
```

Note that the pair `BEGIN` and `END` can always be replaced by the pair `(` and `)`. How you lay out your program is up to you, but applying proper indentation will help you write comprehensible programs. Here is another simple example program:

```
BEGIN INT m = read int;
      INT n = read int;
      print("sum is");
      print(m + n)
END
```

Note that there is no semicolon before `END`. Unlike C where a semicolon is a statement terminator, in Algol 68 it is a statement separator. In the above example we have used two `print` statements to print two values. The routine `print` can print an arbitrary number of values in a single statement, but this will be treated in chapter 5 since the orthogonality of Algol 68 plays a role here.

The `print` statement, applied identifiers and denotations are units. Chapter 9 will explain units and coercions in detail. Units and declarations are separated by the go-on symbol, which is a semicolon. A sequence of at least one unit and if needed declarations, separated by semicolons, is called a serial-clause. Since a serial-clause yields a value but a declaration never does, a serial-clause cannot end in a declaration.

An important concept to introduce in this section is that of a range. The construct `BEGIN serial-clause END` syntactically is a closed-clause which is also a range, meaning that any declaration in that range is valid only in the range itself and in embedded ranges, but not in outside ranges. This holds for any declaration. The hierarchy of the range of `BEGIN serial-clause END` is illustrated by:

```
BEGIN  -------
      |       |
      |       |
      |       |
       -------
END
```

2.9 Variables

Previous sections dealt with static values: denotations or values associated with an identifier through an identity-declaration. Once associated, the association between the identifier and a value does not change as long as the identifier is in existence. For most programs it is required that such association can change, specifically, that the value can change. In order to change values, we need some way of referring to them. Algol 68 can generate values that refer to other values. A value that refers to another value is called a name. Algol 68 names are often called ”variables”, which is also the common term in other programming languages.

Variables in computer programming are very different from variables in mathematics and the apparent similarity is a source of much confusion. Variables in most of mathematics (those that are extensional and referentially transparent) are unknown values, while in programming a variable can associate with different values during the time that a program is executed.

In computer programming a variable is a special value (also often called a reference) that has the property of being able to be associated with a value (or no value, this is the name `NIL`, see section 2.14). What is actually ”variable” is the association. Obtaining the value associated with a variable is called dereferencing, and creating or consequently changing the association is called assignation.

We have encountered the procedure `print` that produces output. Of course, there is also a procedure `read` that will input data. Now suppose `read` is presented with the character sequence 123 and is expecting an integer. `read` will convert the text digits into an `INT` value, but *where* must it store this value? To store that value, a name must be passed as argument to `read`.

The mode of a name is called a ”reference mode”, and ”reference” has the Algol 68 keyword `REF`. A name which can refer to a value of mode `INT` is said to have the mode `REF INT`. Likewise, we can create names with modes

```
REF REAL
REF BOOL
REF CHAR
```

`REF` can precede any mode except `VOID`. Since names are values. `REF` can also precede a mode already containing `REF`. Thus it is possible to construct modes such as

```
REF INT
REF REF REAL
REF REF REF BOOL
REF REF REF REF CHAR
```

Although you can write as many REFs as you like, in practice you will not encounter more than four, which are REF-REF-REF variables.

Names are created using generators. There are two kinds of generator: local and global. The extent to which a name is valid is called its scope. The scope of a local name is restricted to the smallest enclosing clause which contains declarations. The scope of a global name extends to the whole program. In general, values have scope, identifiers have range. We will meet global generators in chapters 6 and 11.

The generator `LOC INT` generates a name of mode `REF INT` which can refer to a value of mode `INT`. The `LOC` stands for local. It is not wrong to write

```
read(LOC INT)
```

but the created name is anonymous in the sense that it has no identifier so that once `read` has assigned a value to it, the name is no longer accessible. We need some way of linking an identifier with the generated name so that we can access the name after `read` has finished. This is of course done with an identity-declaration. Here is an identity declaration with a local generator:

```
REF INT a = LOC INT
```

The value identified by `a` has the mode `REF INT` because the generator `LOC INT` generates a name of mode `REF INT`. Thus it is a name, and it can refer to a value (as yet undefined) of mode `INT` (the value referred to always has a mode of one less `REF`). So now, we can write

```
read(a)
```

When `read` finishes, `a` identifies a name which now refers to an integer. Another term for `a` is ”`INT` variable”.

Later on in this chapter you will learn that there is an abbreviated variable-declaration which would let you simply write

```
INT a;
```

but we will use the unabbreviated form for the time being to emphasize what a variable actually is.

Whereas `LOC` will generate local names, `HEAP` will generate global names. The generator `HEAP INT` generates a global name of mode `REF INT` which can refer to a value of mode `INT`. So we could write

```
REF INT a = HEAP INT
```

and here we arrive at an interesting issue. An identity-declaration associates an identifier with a value, in this case a name. This association is a ”constant”. But you should not assume that the name is a *constant address* in memory. You can imagine that during execution of an Algol 68 program the heap may fill up, but the heap will contain much data that is no longer accessible (temporary data, data from ranges that have ceased to exist, et cetera). Algol 68 employs a mechanism called garbage collection that when needed removes inaccessible data from the heap, thereafter compacting the heap, to restore space. Heap compaction means that *addresses* are not constant, though the relation between name and value remains unbroken. This relation is constant, the physical address is not.

Names can also be declared using a previously declared name on the right-hand side of the identity-declaration: using `a`:

```
REF INT b = a
```

In this declaration, `b` has the mode `REF INT` so it identifies a name. `a` also has the mode `REF INT` and therefore also identifies a name. The identity-declaration makes `b` the same name as `a`. This means that if the name identified by `a` refers to a value, then the name identified by `b` will always refer to the same value — both are aliases.

2.10 Assignation

The process of causing a name to refer to a value is called assignation. Using the identifier `a` declared above, we can write

```
a := 3
```

We say ”a becomes 3”. Note that the mode of the name identified by `a` is `REF INT`, and the mode of the denotation 3 is `INT`. After the assignation, the name identified by `a` refers to the value denoted by 3. Look carefully at the assignation. Firstly, an assignation consists of three parts: on the left-hand side is a unit yielding a name, in the middle is the assignation token, and on the right-hand side is a unit yielding a value suitable to that name. The right-hand side of an assignation can be any unit which yields a value whose mode has one less `REF` than the mode of the name on the left-hand side.

When an identifier for a name is declared, the name can be made to refer to a value immediately. This is commonly called ”initialisation”. For example

```
REF REAL x = LOC REAL := pi
```

where `pi` is a value declared in the standard-prelude. `LOC REAL` generates a name of mode `REF REAL`.

The right-hand side of an assignation is a strong context so widening is allowed. Thus we can write

```
x := 3
```

where the 3 is widened to 3.0 before being assigned to `x`.

Here is another identity-declaration with an initial assignation:

```
REF INT c = LOC INT := 5
```

Using the identifier `a` declared earlier, we can write

```
a := c
```

and say ”a becomes c”. The name on the left-hand side of the assignation has mode `REF INT`, so a value which has mode `INT` is required on the right-hand side, but what has been provided is a name with mode `REF INT`. Algol 68 has a coercion which replaces a name with the value to which it refers: this is called dereferencing and is allowed in a strong context. In the above assignation, the name identified by `c` is dereferenced yielding the value five. The left-hand side of an assignation, a name, is in a soft context. In a soft context, dereferencing is not allowed — it is the only context in which dereferencing is not allowed.

Every construct in Algol 68 yields a value except a declaration, that yields no value. We said above that the value of the left-hand side of an assignation is a name. In fact, the value of the whole of the assignation is the value of the left-hand side. Because this is a name, it can be used on the right-hand side of another assignation. For example:

```
a := b := c
```

You should note that an assignation is not an operator. The assignations are performed from *right to left*: firstly, `c` is dereferenced and the resulting value assigned to `b`. Then `b` is dereferenced and the resulting value is assigned to `a`.

One of the properties of assignation is that the elaboration of the two sides, destination and source, is performed collaterally. This means that the order of elaboration is undefined. Take care in complex assignations that the effect of the code does not depend on the order of elaboration.

2.11 Abbreviated variable declarations

When declaring names, the involved mode is repeated on both sides. For example:

```
REF REAL x = LOC REAL
```

Declarations of names are very common in Algol 68 programs and may therefore be abbreviated declarations. The above declaration can be written

```
LOC REAL x
```

or, most commonly

```
REAL x
```

Equivalently

```
REF REAL y = HEAP REAL := 0;
```

can be abbreviated to

```
HEAP REAL y := 0;
```

but the `HEAP` symbol could not be omitted.

Since `REF` cannot precede `VOID` you cannot use `VOID` as actual-declarer in an abbreviated variable declaration. An abbreviated declaration uses the actual-declarer (the right-hand side of an identity-declaration) followed by the identifier; and if the actual-declarer contains the generator `LOC`, you can omit the `LOC`. Generally speaking, actual-declarers are required when space is generated for an object, for instance in a variable-declaration or a generator. For `INT`, `REAL`, `BOOL` and `CHAR` the formal-declarer is the same as the actual-declarer.

It is important to note that identity-declarations cannot be mixed with abbreviated name declarations because the modes are quite different. For example, in

```
REAL a := 0, b = 1
```

the mode of `a` is `REF REAL`, but the mode of `b` is `REAL`. In the abbreviated declaration of a name, the mode given is that of the value to which the name will refer (the actual-declarer).

2.12 Reading variables

Wherever previously we have used a value of mode `INT` with `print`, we can safely use a name with mode `REF INT`, and similarly with all the other modes (such as `[,] REAL`). This is because the parameters for `print` (the identifiers or denotations used for `print`) are in a firm context and so can be dereferenced before being used.

It is now time to examine `read` more closely. Generally speaking, values displayed with `print` can be input with `read`. The main differences are that firstly, the parameters for `read` must be names. For example, we may write

```
REF REAL r = LOC REAL;
read(r)
```

and the program will skip spaces, tabs and end-of-line and new-page characters until it meets an optional sign followed by optional spaces and at least one digit, when it will expect to read a number. If an integer is present, it will be read, converted to the internal representation of an integer and then widened to a real.

Likewise, `read` may be used to read integers. The plus and minus signs (`+` and `-`) can precede integers and reals. Absence of a sign is taken to mean that the number is positive. Any non-digit will terminate the reading of an integer except for a possible sign at the start. Reals can contain `e` as in `31.4e-1`.

For a name of mode `REF CHAR`, a single character will be read, `new line` or `new page` being called if necessary. Boolean values are read as `T` and `F` respectively, with obvious meaning.

Just like `print`, `read` can take more than one parameter by enclosing them in a row-display.

2.13 References to names

Any mode which starts with `REF` is the mode of a name. The value to which a `REF AMODE` name refers has mode `AMODE`. Since names are values in their own right, there is no reason why a name should not refer to a name. For example, suppose we declare

```
INT j, k
```

then the mode of both `j` and `k` is `REF INT`. We could also declare

```
REF INT jj, kk
```

so that `jj` and `kk` both have the mode `REF REF INT`.

Now, according to the definition of an assignation (see section 9.9), it is allowed to write

```
INT j, k
REF INT jj, kk
jj := j
```

because the identifier on the left has mode `REF REF INT` and the identifier on the right has mode `REF INT`.

The potential pitfall in assignations to `REF` variables will be clear after coercions are discussed in a later chapter, but the idea can already be explained here: Algol 68 can adapt the number of `REF`s of the source of an assignation to the number of `REF`s of the destination, but not vice versa. Hence the assignation in

```
INT j;
REF INT ref j = j;
```

```
ref j := 1
```

fails since the `INT` value 1 is not a value for a name of mode `REF REF INT`. A way around this will be discussed later - the cast, that can force coercions where you need them. Above assignation should be written forcing the coercion to `REF INT` for the destination:

```
REF INT (ref j) := 1
```

with the effect that also `j` will be associated with 1!

2.14 The value NIL

Under circumstances you will want that a name refers to no value. This can be arranged by equating the name to `NIL`. `NIL` is the only denotation of a name in Algol 68. The mode of `NIL` is that of the name required by the context. For example, consider

```
REF INT k = NIL
```

then `NIL` has mode `REF INT`. Although `NIL` is a name, you cannot assign to it. An assignation to `k` as declared above

```
k := 0
```

would cause the runtime error

```
2     k := 0
      1
a68g: runtime error: 1: attempt to access NIL name of mode REF INT
(detected in particular-program).
```

The principle application of `NIL` is letting a name refer to no value. We will see in the sections on lists and trees that this is a vital function.

2.15 Comments

When you write a program, it is of course obvious to you how the program works. However, to someone else (or to you if you return to that program after several months) the source code may not at all be self-evident. Therefore you will want to write comments in the source code. Comments can be put almost anywhere, but not in the middle of symbols. A comment is ignored by *a68g*. A comment is surrounded by one of the following pairs:

```
COMMENT ... COMMENT
CO ... CO
# ... #
```

where the ... represent the actual comment. If you start a comment with `COMMENT` then you must also finish it with `COMMENT`, and likewise for the other comment symbols. Here is an example comment describing the purpose of a program:

```
BEGIN
   COMMENT
      The determinant of the square matrix 'a' of order 'n' by the
      method of Crout with row interchanges: 'a' is replaced by its
      triangular decomposition, l * u, with all u[k, k] = 1.
      The vector 'p' gives as output the pivotal row indices; the k-th
      pivot is chosen in the k-th column of T such that
      ABS l[i, k] / row norm is maximal
   COMMENT
   ...
END
```

It is also recommended to indicate limitations of the program

```
BEGIN
   CO This only works for powers of two! CO
   ...
END
```

It is common practice to "comment out" pieces of source-code during the test phase of a program, as in

```
BEGIN INT i = read int;
      COMMENT
      print(i); # trivial check #
      COMMENT
      REAL x := i;
      print(x)
END
```

This is an example of "nested" comments. Since in Algol 68 the symbol starting a comment is equal to the symbol ending a comment, it is not possible to have proper nested comments in Algol 68. To have proper nested comments, the interpreter should be able to count how many comments are open, which is not possible if the embedding symbols are equal. Therefore, if the part of your program that you want to comment out already contains comments, you should ensure that the inner comment symbols are different from those of the outermost comment, because *a68g* only scans the outermost comment and ignores all text until the matching comment symbol.

Compare this to Pascal where comment symbols are { `...` } or C where comment symbols are `/* ... */` making it possible to count how many comments are open; however not all implementations of those languages allow nested comments.

2.16 Pragmats

A `pragmat` is a pragmatic remark that you can write in your source code. The Algol 68 definition leaves it up to the implementation what such pragmatic remark should do, and in *a68g* it lets you insert command-line options in the source.

A pragmat is surrounded by one of the following pairs:

```
PRAGMAT ... PRAGMAT
PR ... PR
```

For instance, if you have a script that needs a lot of heap space you do not want to have to specify an option at the command line. Instead you write in your source code

```
#!/usr/local/a68g

BEGIN PR heap=256M PR
   ...
END
```

The pragmat will execute the option at compile time as if you would have specified `--heap=256M` from the command line.

Chapter 3

Formulas

3.1 Formulas

Formulas — often called *expressions* in other programming languages — consist of operators working on operands. Operators are predeclared pieces of program which compute a value from their operands. Algol 68 provides a rich set of operators in the standard-prelude, described in chapter 11, and you can define as many more as you want. In this chapter, we will examine the operators in the standard-prelude which can take operands of mode `INT`, `REAL`, `BOOL` or `CHAR` . In chapter 6, we will return to operators and look at what they do in more detail, as well as how to define new ones.

Operator symbols are written as a combination of one or more symbols, or in capital letters like a mode indicant. We will meet both kinds in this chapter.

3.2 Monadic operators

Operators come in two types: monadic and dyadic. A monadic operator has only one operand, while a dyadic operator has two operands. A monadic operator is written before its operand. For example, the monadic minus `-` reverses the sign of its operand:

```
-4096
```

As in identifiers, spaces are not significant in formulas as long as there is a unique meaning for a statement. There is, likewise, a monadic `+` operator which does not do anything to its operand. It has been provided for the sake of consistency. You should note that `-4096` is not a denotation, but a formula consisting of a monadic operator operating on an operand which is a denotation. We say that the monadic operator `-` takes an operand of mode `INT` and yields a value of mode `INT` . It can also take an operand of mode `REAL` in which case it will yield a value of mode `REAL` .

A formula can be used as the unit of an identity declaration. Thus the following identity declarations are both valid:

```
INT  minus 2 = -2;
REAL temp below zero = - read real
```

The operator `ABS` takes an operand of mode `INT` and yields the absolute value, again of mode `INT`. For example, `ABS -1` yields 1. Note that when two monadic operators are combined, they are elaborated in right-to-left order, as in the above example. That is, in `ABS -1` `-` acts on the 1 to yield -1, then `ABS` acts on -1 to yield 1. This is just what you might expect. In the standard-prelude is another definition of `ABS` that takes an operand of mode `REAL` yielding a value of mode `REAL`. For example:

```
REAL x = - read real;
REAL y = ABS sin(x)
```

Another monadic operator which takes an `INT` or `REAL` operand is `SIGN` which yields -1 if the operand is negative, 0 if it is zero, and $+1$ if it is positive. Thus you can declare

```
INT res = SIGN i
```

if `i` has been declared.

3.3 Dyadic operators

A dyadic operator takes two operands and is written between them. A basic operator is dyadic `+`, for instance:

```
print (read int + read int)
```

Note that this program only works because $a + b = b + a$; the two `read`s are elaborated collaterally so you would not know which is read first. This `+` operator takes two operands of mode `INT` and yields a result of mode `INT` . It is also defined for two operands of mode `REAL` yielding a result of mode `REAL` :

```
REAL x = read real + offset
```

Before we continue with the other dyadic operators, a word of caution is in order. As we have seen, the maximum integer which the computer can use is `max int` and the maximum real is `max real`. The dyadic `+` operator could give a result which is greater than those two values.

Adding two integers such that the sum exceeds `max int` is said to give ”integer overflow”. Algol 68 leaves such case undefined, meaning that an implementation can choose what to do.

a68g will give a runtime error in case of arithmetic under - or overflow.

The dyadic `-` operator can also take two operands of mode `INT` or two operands of mode `REAL` and will yield an `INT` or `REAL` result respectively:

```
INT difference = a - b,
REAL distance = end - begin
```

Since a formula yields a value of a particular mode, you can use it as an operand for another operator. For example:

```
INT sum = a + b + c
```

the order of elaboration being that operands are elaborated collaterally, and then the operators are applied from left-to-right in this particular example, since the two operators have the same priority.

3.4 Multiplication

The operator `*` performs arithmetic multiplication and takes `INT` operands yielding an `INT` result. For example:

```
INT product = 45 * 36
```

Likewise, `*` is also defined for multiplication of two values of mode `REAL` :

```
REAL pi 2 = 2.0 * pi
```

As with `+` and `-`, a formula can be an operand in another formula:

```
INT factorial 6 = 2 * 3 * 4 * 5 * 6
```

the order of elaboration again being that the operands are elaborated collaterally, and then the operators are applied from left-to-right in this particular example, since the operators have the same priority.

3.5 Order of evaluation

In Algol 68, the common precedence of brackets over division, than multiplication, than addition and subtraction, applies and it is implemented by giving a priority to operators. The priority of multiplication is higher than the priority for addition or subtraction. The priority of the dyadic `+` and `-` operators is 6, and the priority of the `*` operator is 7. For example, the value of the formula $2 + 3 * 4$ is 14. It is possible to change the priority of standard operators, but that makes not much sense — priority-declarations are provided to define the priority of *new* dyadic operators you introduce.

Every dyadic operator has a priority of between 1 and 9 inclusive, and monadic operators bind more tightly than any dyadic operator.

You can of course force priorities by writing sub-expressions in brackets:

```
INT m = 1 + (2 * 3), # 7 #
    n = (1 + 2) * 3; # 9 #
```

Note that the brackets in Algol 68 are a short-hand of `BEGIN` ... `END` and indeed, you could write a clause in brackets:

```
INT one ahead = 1 + (INT k; read(k); k)
```

This is a consequence of Algol 68's famed orthogonality. We will see many more of those examples throughout this manual. Parentheses can of course be nested to any depth as long as *a68g* does not run out of memory.

In principle, any construct yielding a value that is not of mode `VOID` is a potential operand in a formula. Since an assignation yields a value (specifically, a name), it can be used as an operand in a formula. However, an assignation is a unit, and a unit cannot be a direct operand (see chapter 9). The assignation must be "packed" in an enclosed clause using parentheses, or `BEGIN` and `END`; for example

```
2 * (a := a + 1)
```

Remember that the assignation-symbol `:=` is not an operator, which explains why an assignation cannot be a direct operand: if it could be a direct operand, a statement `x := 1 + read int` could mean either `(x := 1) + read int` or `x := (1 + read int)` with different values for `x` after completion of the formula.

Now let us return to order of evaluation. In

```
INT side sq = a * a + b * b
```

the order of elaboration is that both multiplcations are elaborated collaterally. Hence do not write code that yields a result that depends on the order of evaluation, for instance

```
INT poly = a * a + (a := a + 1)
```

which would yield either $a^2 + a + 1$ or $a^2 + 2a + 1$ or $a^2 + 3a + 2$ at the compiler's discretion. Write this strictly as:

```
a := a + 1;
INT poly = a * (a + 1)
```

Remember from chapter 1 that widening is allowed in the context of the right-hand side of an identity declaration, so the following declaration is valid:

```
REAL a = 24 * -36
```

It is important to note that an operand is not in a strong context, but in a firm context. In a firm context no widening is allowed, otherwise we could not decide whether to use integer addition or real addition when we add integer operands! Section 6.15 discusses details. Hence in above example the formula is elaborated first, and the final `INT` result is widened to `REAL`.

3.6 Division

Division poses a dilemma because division of integers can have two different meanings. For example, $3 \div 2$ can be taken to mean 1 or 1.5. Therefore Algol 68 uses two different operator symbols. We have an operator `%` that divides integers to yield and integer result and an operator `/` that divides integers to yield a real result.

Integer division is performed by the operator `%`. It takes operands of mode `INT` and yields a value of mode `INT` . It has the alternative representation `OVER`. The formula `7 % 3` yields the value 2, and the formula `-7 % 3` yields the value -2. The priority of `%` is 7, the same as multiplication.

The modulo operator `MOD` gives the remainder after integer division. It requires two operands of mode `INT` and yields a value also of mode `INT` . Algol 68 defines `a MOD b` as this: let $q \in \mathbb{Z}$ be the quotient of $a \in \mathbb{Z}$ and $b \in \mathbb{Z}, b \neq 0$ and $r \in \mathbb{W}$ the remainder, such that this relation holds:

$$a = q \times b + r; 0 \leq r < |b|$$

then `a MOD b` yields r. Thus `7 MOD 3` yields 1 ($q = 2$), but `-7 MOD 3` yields 2 ($q = -3$). The result of `MOD` is always a non-negative number. `MOD` can also be written as `%*` and its priority is 7.

Division of real numbers is performed by the operator `/`. It takes two operands of mode `REAL` and yields a `REAL` result. Thus the formula `3.0 / 2.0` yields 1.5. Again, `/` can be combined with `*` and the other operators already discussed. It has a priority of 7. As said above, the operator is also defined for integer operands. Thus `3 / 2` also yields 1.5. No widening takes place here since the operator is defined to yield a value of mode `REAL` when its operands have mode `INT` .

3.7 Exponentiation

If you want to compute the value of `x * x * x * x` you can do so using multiplication, but it is clearer if you used the exponentiation operator `**` or its equivalents `^` or `UP`. Its priority is 8. The mode of its left operand can be either `REAL` or `INT` but its right operand, the exponent, must have mode `INT` . If both its operands have the mode `INT`, the yield will have mode `INT` and in this case the right operand must not be negative; if the left operand is real the yield will have mode `REAL`. Thus the formula `3 ** 4` yields 81 and `3.0 ** 4` yields 81.0. In a formula involving exponentiation as well as multiplication or division, exponentiation is elaborated first. For example, the formula `3 * 2 ** 4` yields 48, not 1296.

A common pitfall is the formula `-x ** 2` which yields x^2 in stead of $-(x^2)$. The monadic minus is elaborated first, followed by the exponentiation. This looks straightforward, but even experienced programmers tend to make this mistake every now and then. This particular example is not specific to Algol 68, for instance FORTRAN has the same peculiarity.

3.8 Mixed arithmetic

Up to now, the arithmetic operators `+`, `-` and `*` have had operands of the same modes:

$$\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

In practice, it is quite surprising how often you want to compute something like `2 * 3.0`. The dyadic operators `+`, `-`, `*` and `/` (but not `%`) are also defined for mixed modes. That is, any combination of `REAL` and `INT` can be used:

$$\mathbb{Z} \times \mathbb{R} \rightarrow \mathbb{R}$$

$$\mathbb{R} \times \mathbb{Z} \rightarrow \mathbb{R}$$

With mixed modes the yield is always `REAL`. Thus the following formulas are all valid:

```
small real + 1
2 * pi
```

The priority of the mixed-mode operators is unchanged since as we will see later, the priority relates to the operator symbol rather than modes of operands or result.

3.9 Real to integer conversion

We have seen that in a strong context, a value of mode `INT` can be coerced by widening to a value of mode `REAL`. How do we coerce a value of mode `REAL` to a value of mode `INT` ? This is impossible using (implicit) coercion. The reason behind this is the design choice in Algol 68 that the fractional part cannot be implicitly lost. The programmer has to explicitly state how the conversion should take place.

If you want to convert a `REAL` value to an `INT`, you must use one of the operators `ROUND` or `ENTIER`. The operator `ROUND` takes a single operand of mode `REAL` and yields an `INT` whose value is the operand rounded to the nearest integer. Thus `ROUND 2.7` yields 3, and `ROUND 2.2` yields 2. The same rule applies with negative numbers, thus `ROUND -3.6` yields -4. Essentially, you get an integer result that differs not more than 0.5 from the real value.

The operator `ENTIER` takes a `REAL` operand and likewise yields an `INT` result, but the yield is the largest integer that is not larger than the real operand. Thus `ENTIER 2.2` yields 2, `ENTIER -2.2` yields -3.

3.10 Real functions

Routines are the subject of a later chapter, however we have already met the routines `print`, `read`, `read int`, `read real`, `read bool` and `read char`. At this point we mention that like all other languages, Algol 68 defines various mathematical functions for real values. Next list summarises the routines that are defined in the Algol 68 Genie standard-prelude:

1. `sqrt(x)`
   The square root of `x`.
2. `curt(x)`
   The cube root of `x`.
3. `exp(x)`
   Yields e^x.
4. `ln(x)`
   The natural logarithm of `x`.
5. `log(x)`
   The logarithm of `x` to base 10.
6. `sin(x)`
   The sine of `x`, where `x` is in radians.

7. `arcsin(x)`
   The inverse sine of `x`.
8. `cos(x)`
   The cosine of `x`, where `x` is in radians.
9. `arccos(x)`
   The inverse cosine of `x`.
10. `tan(x)`
    The tangent of `x`, where `x` is in radians.
11. `arctan(x)`
    The inverse tangent of `x`.
12. `arctan2(x, y)`
    The angle whose tangent is y/x. The angle will be in range $[-\pi, \pi]$.
13. `sinh(x)`
    The hyperbolic sine of `x`.
14. `arcsinh(x)`
    The inverse hyperbolic sine of `x`.
15. `cosh(x)`
    The hyperbolic cosine of `x`.
16. `arccosh(x)`
    The inverse hyperbolic cosine of `x`.
17. `tanh(x)`
    The hyperbolic tangent of `x`.
18. `arctanh(x)`
    The inverse hyperbolic tangent of `x`.

Note that a runtime error occurs if either argument or result are out of range. Multi-precision versions of these function are declared and are preceded by either `long` or `long long`, for instance `long sqrt` or `long long ln`.

3.11 Boolean operators

The simplest monadic operator with an operand of mode `BOOL` is `NOT`, with alternative representation `^` or `~`. If its operand is `TRUE`, it yields `FALSE` . Conversely, if its operand is `FALSE`, it yields `TRUE` . The operator `ODD` yields `TRUE` if its integer operand is an odd number and `FALSE` if it is even. The operators can be combined, so

```
NOT ODD 2
```

yields `TRUE` . `ABS` converts its operand of mode `BOOL` and yields an integer result: `ABS TRUE` yields 1 while `ABS FALSE` yields 0.

Dyadic operators with boolean result come in two kinds: those that take operands of mode `BOOL`, yielding `TRUE` or `FALSE`, and those that operate on operands of other modes, which are comparison operators described in section 3.13.

Three dyadic operators are declared in *a68g*'s standard-prelude which take operands of mode `BOOL`. The operator `AND`, with alternative representation `&`, yields `TRUE` only if both its operands yield `TRUE` . The priority of `AND` is 3. The operator `OR` yields `TRUE` if at least one of its operands yields `TRUE` . The priority of `OR` is 2. The operator `XOR` yields `TRUE` if at most one of its operands yields `TRUE` . It has no alternative representation. The priority of `XOR` is 3.

3.12 Character operators

Algol 68 provides operators with operands of mode `CHAR` . The `+` and `*` operators are declared, and we will meet them in chapter 5. As you would guess these two operators involve (repeated) concatenation. There are two monadic operators which involve the mode `CHAR` . The operator `ABS` takes a `CHAR` operand and yields the integer corresponding to that character. For example, `ABS "A"` yields 65 (if your platform uses ASCII encoding). The identifier `max abs char` is declared in the standard-prelude and will give you the maximum number in your encoding. Conversely, we can convert an integer to a character using the monadic operator `REPR`; hence the formula

```
REPR 65
```

yields the value `"A"`. `REPR` accepts an integer in the range 0 to `max abs char`. `REPR` is of particular value in allowing access to control characters (see chapter E).

3.13 Comparison operators

Values of modes `INT`, `REAL`, `BOOL` and `CHAR` can be compared. The boolean-formula

```
3 = 1 + 2
```

yields `TRUE` . Similarly,

```
1 + 1 = 1
```

yields FALSE . The equals-symbol = can also be written EQ. Likewise, the formula

```
35.0 EQ 3.5e1
```

should yield TRUE .

You should be cautious when comparing two REAL values for equality or inequality because of subtle rounding errors and finite precision of an object of mode REAL. For instance, `1.0 + small real / 2 = 1.0` will yield TRUE.

Because the rowing coercion is not allowed in formulas, comparison operators are declared in the standard-prelude for mixed modes (such as REAL and INT).

The negation of = is /= (not equal). So the formula

```
3 /= 2
```

yields TRUE, and

```
TRUE /= TRUE
```

yields FALSE . Alternative representations of /= are   = and NE. The priority of both = and /= is 4. The operands of = and /= can be any combination of values of mode INT and REAL . No widening takes place, the operators being declared for the mixed modes.

The comparison operators <, >, <= and >= can be used to compare values of modes INT, REAL and CHAR in the same way as = and /=. Alternative representations for these operators are LT and GT for < and > and LE and GE for <= and >= respectively. The priority of all these four comparison operators is 5.

If the identifiers b and c are declared as having mode CHAR, then the formula

```
c < b
```

will yield the same value as

```
ABS c < ABS b
```

and similarly for the other comparison operators.

Boolean formulas yielding TRUE or FALSE can of course be combined. For example, here is a formula which tests whether π lies between 3 and 4

```
pi > 3 AND pi < 4
```

which yields TRUE . The priorities of <, > and AND are defined such that parentheses are unnecessary in this case. Likewise, we may write

```
"ab" < "aac" OR 3 < 2
```

which yields `FALSE` . More complicated formulas can be written:

```
cycle < 10 AND cmd /= "s" OR read bool
```

Because the priority of the operator `AND` is higher than the priority of `OR`, the `AND` in the above formula is elaborated first. Legibility can be improved using parentheses.

3.14 Operators combined with assignation

Consider the assignations

```
a := a + 1
```

The right-hand side of the assignation is a formula. The name `a` in the formula is in a firm context. In a firm context, dereferencing is allowed (and also uniting, but that is a subject of chapter 7). The name is dereferenced, the resulting value is added to 1, and then the new value is assigned to the same name.

Assignations of this kind are so common that a special operator has been devised to perform them. The above assignation can be written

```
a +:= 1
```

and is read ”a plus-and-becomes one”. The operator has the alternative representation `PLUSAB`. The left-hand operand must be a name. The right-hand operand must be any unit which yields a value of the appropriate mode in a firm context.

The operator `+:=` is defined for a left-operand of mode `REF INT` or `REF REAL`. The `REF INT` operator expects an integer right-hand operand. There are two `REF REAL` operator expecting a right-hand operand of mode `INT` or `REAL` respectively. No widening occurs in this case, the operator having been declared for operands having these modes. The yield of the operator is the value of the left-hand operand (the name). Because the operator yields a name, that name can be used as the operand for another assigning operator. For example

```
INT a := read int;
INT b := a +:= 1
```

which results in both `a` and `b` referring to the value $a + 1$.

There are other operators like `+:=`. They are `-:=`, `*:=`, `/:=`, `%:=` and `%*:=` with obvious meaning. Their alternative representations are respectively `MINUSAB`, `TIMESAB`, `DIVAB`, `OVERAB` and `MODAB`. The operators `OVERAB` and `MODAB` are only declared for operands with modes `REF`

`INT` and `INT`. The priority of all the operators combined with assignation is 1.

Remember that the assignation operators are operators, not assignation-symbols, so that their application constitutes a formula, not an assignation.

Visual distortion in the night sky if a Schwarzschild black hole were to pass close to the Earth, calculated and drawn with a68g. Note for instance the distorted constellation of Orion to the right to the black hole. The corona of stars around the black hole are secondary images resulting from the hole's gravitation. In Einstein's general relativity, the Schwarzschild metric describes gravitation outside a spherical, non-charged, non-rotating mass. For more details see [Nemiroff 1993].

Chapter 4

Program structure

4.1 Introduction

In this chapter, we will describe program structures which allow an Algol 68 program to choose between alternatives or to perform a repetitive task. The conditional-clause allows a program to elaborate code depending on a boolean value. The case-clause allows a program to elaborate code depending on an integer value. The loop-clause lets you execute code repeatedly.

4.2 The conditional-clause

The part of the enclosed-clause inside parentheses or `BEGIN` and `END` is called a serial-clause because, historically, sequential elaboration used to be called ”serial elaboration”. The value of the serial-clause is the value of its last statement which must be a unit, never a declaration. Also, `DO` . . . `OD` encloses a serial-clause. In this chapter, we will encounter a type of serial-clause that cannot have labels — these are called enquiry-clauses.

The conditional-clause allows a program to elaborate code depending on the value of a boolean-enquiry-clause. Here is a simple example:

```
IF REAL x = read real;
   x > 0
THEN ln(x)
ELSE stop
FI
```

If the `BOOL` enquiry-clause yields `TRUE` , the serial-clause following `THEN` is elaborated, otherwise the serial-clause following `ELSE` is elaborated. The symbol `FI` following the `ELSE` serial-clause is a closing parenthesis to `IF`.

The `ELSE` part of a conditional-clause can be omitted. When the `ELSE` part is omitted, and the conditional-clause is expected to yield a value, an undefined value of the required mode will be yielded if the enquiry-clause yields `FALSE`. Actually, if the `ELSE` part is missing then its serial-clause is regarded as consisting of the single unit `SKIP`.

The use of `IF` with matching `FI` eliminates the dangling-else problem — since `ELSE` is optional, to what `IF` does a nested `ELSE` belong in nested conditional-clauses? In for example Pascal and C such matters are decided by writing extra rules next to the syntax stating that any `ELSE` is related to the closest preceding `IF`. In Algol 68, such rules are not necessary.

The enquiry-clause on line 1 in above example consists of the enquiry-clause

```
REAL x = read real;
x > 0
```

which obviously ends in a unit yielding a value of mode `BOOL`. Two serial-clauses, in this case both containing a single unit, can be elaborated. If the value yielded by `x` is is a positive number, the $ln(x)$ is yielded. Otherwise, the program stops.

An enquiry-clause and a serial-clause may consist of at least one unit and possibly declarations. However, the last statement in an enquiry-clause must be a unit yielding `BOOL`. The hierarchy of ranges in conditional-clauses is illustrated by

```
 -----------IF-----------
|   -THEN-      -ELSE-   |
|  |      |    |      |  |
|   ------      ------   |
 -----------FI-----------
```

The range of any declaration in an enquiry-clause extends to the serial-clauses following `THEN` and `ELSE`. All declarations in the conditional-clause cease to exist when `FI` is encountered. Note that this design of the conditional-clause allows you to write any declaration in the smallest range required, thus shielding them from other parts of the program that have no need for these declarations. This allows for a safe and clean programming style.

The conditional-clause can be written wherever a unit is permitted, so for example

```
IF INT a = read int;
   a < 0
THEN print((a, " is negative"))
ELSE print((a, " is not negative"))
FI
```

can also be written

```
INT a = read int;
print((a, " is", IF a >= 0 THEN " not" FI, " negative"))
```

The value of each of the serial-clauses following `THEN` and `ELSE` in this case is `[] CHAR`.

The conditional-clause can appear as an operand in a formula. A short form for the conditional-clause is often used for this: `IF` and `FI` are replaced by `(` and `)` respectively, and `THEN` and `ELSE` are both replaced by a vertical bar `|`For example, assuming a declaration for `x`:

```
REAL abs = (x < 0.0 | - x | x)
```

which is equivalent to

```
REAL abs = IF x < 0.0 THEN - x ELSE x FI
```

If the `ELSE` part is omitted then its serial-clause is regarded as consisting of a single unit `SKIP`. For instance:

```
REAL quotient = (y /= 0.0 | x / y)
```

where the quotient will have an undefined value in case $y = 0$. In such case, `SKIP` will yield an undefined value of the mode yielded by the `THEN` serial-clause, which is forced by the formal-declarer `REAL`. This is an example of balancing (balancing is explained in chapter 9).

Formally, `SKIP` is an algorithm that performs no action, completes in finite time, and yields some value of the mode required by the context. You will see later on in this manual that `SKIP` can be quite useful.

Since the right-hand side of an identity-declaration is in a strong context, widening is allowed. Thus, in

```
REAL x = (i < j | 3 | 4)
```

whichever value the conditional-clause yielded would be widened to a value of mode `REAL`.

Since the enquiry clause is a serial-clause, it can have any number of statements before the `THEN` . For example:

```
IF INT measurements;
   read (measurements);
   measurements < 10
THEN ...
FI
```

Conditional-clauses can be nested

```
BOOL leap year = IF year MOD 400 = 0
```

```
                 THEN TRUE
                 ELSE IF year MOD 4 = 0
                      THEN year MOD 100 /= 0
                      ELSE FALSE
                      FI
                 FI
```

A construction `ELSE IF ... FI` can be contracted to `ELIF ...` which saves indentation:

```
BOOL leap year = IF year MOD 400 = 0
                 THEN TRUE
                 ELIF year MOD 4 = 0
                 THEN year MOD 100 /= 0
                 ELSE FALSE
                 FI
```

Note that there is no contraction of `THEN IF` since that would leave undefined whether an eventual else-part would be related to the first condition or to the second.

In the abbreviated form `|:` can be used instead of `ELIF` . For example, the above identity-declaration for `leap year` could be written

```
BOOL leap year = (year MOD 400 = 0 | TRUE
   |: year MOD 4 = 0 | year MOD 100 /= 0 | FALSE)
```

4.3 Pseudo-operators

Sometimes it is useful to include a conditional-clause in the `IF` part of a conditional-clause. In other words, a `BOOL` enquiry-clause can be a conditional-clause yielding a value of mode `BOOL`. Here is an example with `a` and `b` declared with mode `BOOL` :

```
IF IF a
   THEN TRUE
   ELSE b
   FI
THEN ...
ELSE ...
FI
```

As was mentioned in chapter 3, the operands of an operator are all elaborated before the operator is elaborated. Sometimes it is useful to refrain from further elaboration when the result of a formula can be determined from the value of a single operand. To that end *a68g* implements the pseudo-operator `THEF` (with synonyms `ANDF` and `ANDTH`) which although it looks like an operator, elaborates its right-hand operand only if its left-hand operand yields `TRUE`. Compare them with the operator `AND`. The statement `p THEF q` is equivalent to

```
IF p THEN q ELSE FALSE FI
```

There is another pseudo-operator `ELSF` (with synonyms `ORF` and `OREL`) which is similar to the operator `OR` except that its right-hand operand is only elaborated if its left-hand operand yields `FALSE`. The statement `p ELSF q` is equivalent to

```
IF p THEN TRUE ELSE q FI
```

Neither `THEF` nor `ELSF` are part of Algol 68. Compare them to `&&` and `||` in C.

4.4 Identity-relations

In the chapter on formulas we did not address any operation on names. Algol 68 does not prescribe the actual implementation of how a name must refer to a value. On a digital computer such implementation will of course involve addresses in memory, but you have already seen that a name generated by `LOC` is different in nature than a name generated by `HEAP` since the latter can be moved through memory by the garbage collector. Since we do not exactly know *what* a name is, in Algol 68 there is no operation on names except comparison for identity: you can check whether two names are the same, and if they are the same, they refer to the same value if the names are not `NIL`.

We cannot use the equals-symbol `=` for this purpose for two reasons. We will in a later chapter see that it is not possible to declare two operators with identical operator-symbol, one operating on a name and the other operating on the related value; this could cause ambiguity in resolving which of the two operators to apply. Also, we can introduce new modes and thus introduce names of these new modes, and we would need to declare an operator `=` comparing these new names, but to compare them we would need a `=` to compare these names, which puts us in an impossible situation.

Therefore, in Algol 68 names are compared in a special construct, the identity-relation. The identity-relation

```
u :=: v
```

yields `TRUE` if u is the same name as v. The alternative representation of `:=:` is `IS`. The identity-relation

```
u :/=: v
```

yields `TRUE` if u is not the same name as v. The alternative representation of `:/=:` is `ISNT`.

There is a potential pitfall in the identity-relation: you must make sure that you compare names at the proper level of dereferencing! Next artificial program demonstrates the potential difficulty:

```
REF INT i := LOC INT, j := LOC INT, k := LOC INT;
IF read bool THEN i := j ELSE i := k FI;
IF i IS j
THEN print ("TRUE entered")
ELSE print ("FALSE entered")
FI
```

The identity-relation `i IS j` always yields `FALSE` because i, j and k are different variables and thus are different names. What you actually want is to compare the *values* of these pointer-variables, which in this case are names themselves. The point is that no automatic dereferencing takes place in an identity relation. To compare the names that both i and j refer to, you should place at least one side in a cast (see chapter 9):

```
REF INT (i) IS j
```

This will ensure that the right-hand side (in this case) is dereferenced to yield a name of the same mode as the left-hand side. The identity-relation is subject to balancing (a subject treated at length in chapter 9): the compiler places one side of the relation is in a soft context and the other side in a strong context in such way that the modes on both sides are unique and matching. Balancing is also needed to use the most common application of the identity-relation: comparison to `NIL` since by balancing `NIL` takes the mode of the other name in the identity-relation. You than also understand that the identity-relation `NIL IS NIL` does not yield `TRUE`, but gives a compiler error since no unique mode can be established for the names, as the next test using *a68g* demonstrates:

```
$ a68g -p "NIL IS NIL"
1     (print ((NIL IS NIL)))
               1
a68g: error: 1: construct has no unique mode (detected in
closed-clause starting at "(" in this line).
```

4.5 The case-clause

Sometimes the number of choices can be quite large or the different choices are related in a simple way. For example, consider the following conditional-clause:

```
IF n = 1
THEN unit 1
```

```
ELIF n = 2
THEN unit 2
ELIF n = 3
THEN unit 3
ELIF n = 4
THEN unit 4
ELSE unit 5
FI
```

This sort of choice can be expressed more concisely using the integer-case-clause in which the boolean enquiry-clause is replaced by an integer enquiry-clause, forming the integer-case-clause:

```
CASE n
IN unit 1, unit 2, unit 3, unit 4
OUT unit 5
ESAC
```

which could be abbreviated as

```
(n | unit 1, unit 2, unit 3, unit 4 | unit 5)
```

Note that units in the `IN` part are separated by commas. If you want more than one statement for each unit, you must make the latter an enclosed-clause. If the `INT` enquiry-clause yields 1, `unit 1` is elaborated; if it yields 2, `unit 2` is elaborated and so on. If the value yielded is negative or zero, or exceeds the number of units in the `IN` part, `unit 5` in the `OUT` part is elaborated. The `OUT` part is a serial-clause so no enclosure is required if there is more than one statement.

Like the conditional-clause, if you omit the `OUT` part, the interpreter assumes that you wrote `OUT SKIP`.

The hierarchy of ranges in case-clauses is illustrated by

```
 ----------CASE-----------
|   --IN--      --OUT--   |
|  |      |    |       |  |
|   ------      -------   |
 ----------ESAC-----------
```

Sometimes the `OUT` part consists of another integer-case-clause. For example,

```
print ((CASE n MOD 4
        IN "case 1", "case 2", "case 3"
        OUT CASE (n - 10) MOD 4
            IN "case 11", "case 12", "case 13"
            OUT "other case"
            ESAC
        ESAC
      ))
```

Just as with `ELIF` in a conditional-clause, `OUT CASE` ... `ESAC ESAC` can be replaced by `OUSE` ... `ESAC` . So the above example can be rewritten

```
print ((CASE n MOD 4
        IN "case 1", "case 2", "case 3"
        OUSE (n - 10) MOD 4
        IN "case 11", "case 12", "case 13"
        OUT "other case"
        ESAC
      ))
```

Calendar computations give examples of integer-case-clauses:

```
INT days =
   CASE month
   IN 31,
      IF year MOD 4 = 0 AND year MOD 100 /= 0 OR year MOD 400 = 0
      THEN 29
      ELSE 28
      FI,
      31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   ESAC
```

where the case-clause has the function of a table.

4.6 The loop-clause

Suppose you wanted to output 4 blank lines. You could write

```
print((new line, new line, new line, new line))
```

A simpler way would be to write:

```
TO 4 DO print(new line) OD
```

The integer following the `TO` can be any unit yielding an integer (not necessarily positive) in a meek context. In a meek context only dereferencing and deproceduring are allowed. If the value yielded is zero or negative, then the ensuing clause enclosed by `DO` and `OD` will not be elaborated at all. The `TO ... OD` construct is a form of the loop-clause. A loop-clause yields no value (cf section 6.8).

The loop-clause can have a loop-identifier counting the iterations, as is shown in the following example:

```
FOR i TO 10
DO print((i, new line))
OD
```

Here i is an identifier whose mode is `INT`. Note that its mode is not `REF INT` so you cannot perform an assignation to i, it is a "counting constant". The example will print the numbers 1 to 10. The range of i is the whole of the loop-clause, but does not include the unit following `TO`. Any identifier may be used in place of i. When the `TO` part is omitted, `TO max int` is taken by default.

It is possible to modify the initial value of the loop-identifier and also its increment per iteration. The simplest way is to define the starting point using the `FROM` construct. Here is an example:

```
FOR n FROM -10 TO 10 DO print((n, blank)) OD
```

This prints the numbers from -10 to $+10$ on standard output. The integer after `FROM` can be any unit which yields a value of mode `INT` in a meek context. When `FROM` is omitted, it is assumed that the initial value of the loop-identifier following `FOR` is 1.

By default, the value of the loop-identifier is always incremented by 1. The increment can be changed using the `BY` construct. The integer after `BY` can be any unit which yields a value of mode `INT` in a meek context. For example, to print the cubes of even numbers up to and including 10, you could write

```
FOR n FROM 0 BY 2 TO 10
DO print((n ** 3, new line))
OD
```

The increment specified using the `BY` construct can be negative, and than the loop will count backwards. Omitting the `BY` construct assumes a default increment of 1.

a68g offers the keyword `DOWNTO` as an alternative to `TO` . `DOWNTO` will decrement, whereas `TO` will increment, the loop identifier by the amount stated by the (implicit) `BY` part. For example:

```
FOR k FROM 10 DOWNTO 1
DO print(k)
OD
```

`DOWNTO` is an *ALGOL68C* extension.

If you omit the `TO` or `DOWNTO` part, the loop will be repeated virtually indefinitely. In that case, you would need some way of terminating the loop. This can be achieved with loop-checks that Algol 68 provides.

Note a peculiarity in the Revised Report: an anonymous loop-counter will be maintained that on an actual computer can give `INT` overflow even if the loop-identifier and loop-intervals are omitted, The only way the Revised Report can give an infinite loop is by specifying `BY 0`.

The loop-check is a syntactic construct which is very useful for controlling the execution of a loop-clause. It is common to execute a loop as long as a particular condition holds. For example, while integers are negative:

```
WHILE REF INT int = LOC INT;
      read(int);
      int < 0
DO print((ABS int,new line))
OD
```

In this example, no loop counter is needed and so the `FOR` part was omitted. The statement following the `WHILE` must be a meek enquiry clause yielding `BOOL`. In this case, an integer is read each time the loop is elaborated until a non-negative integer is read. The range of any declarations in the enquiry-clause extends to the `DO` ... `OD` loop.

The `WHILE` part provides for a pre-checked loop. There are those who regret that Algol 68's definition did not include a post-checked loop. Apparently it was believed that a post-checked loop was not really necessary since it can be programmed as

```
WHILE do what has to be done;
      condition
DO SKIP
OD
```

but many think that this is not elegant. *a68g* extends Algol 68 by offering a post-checked loop. It does so by allowing an optional until-part as final part of the `DO ... OD` loop:

```
FOR id FROM integer-unit BY integer-unit TO integer-unit
WHILE boolean-enquiry-clause
DO serial-clause
   UNTIL boolean-enquiry-clause
OD
```

A trivial example of a post-checked loop is:

```
DO UNTIL read char = "."
OD
```

which will skip characters from standard input until a "." is encountered which will disappear from the input as well.

The hierarchy of ranges in the loop-clause is illustrated by:

```
       --
1 FOR |  | FROM BY TO
 ------  --------------
| 2 ----WHILE--------  |
|  |                 | |
|  | 3 -DO---------  | |
|  |  |            | | |
|  |  | 4 -UNTIL-  | | |
|  |  |  |       | | | |
|  |  |   -------  | | |
|  |   -OD---------  | |
|   -----------------  |
 ----------------------
```

Again, note that the loop-identifier is unknown in the `FROM`-part, `BY`-part and `TO`-part.

In chapter 1, it was mentioned that the basic structure of an Algol 68 program consists of

```
BEGIN statements
END
```

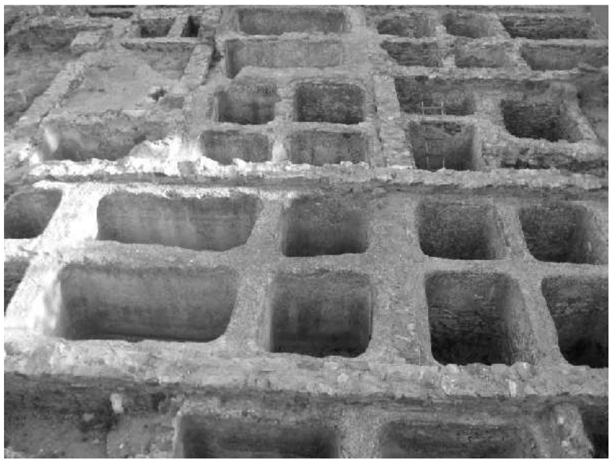
This is not the complete truth. It is quite possible to write a program consisting solely of a conditional-clause, case-clause or loop-clause. For example:

```
FOR i TO 8
DO print((i ** 2, new line))
OD
```

is a perfectly good Algol 68 program. Later on you will learn that a program is in fact an enclosed-clause, such as a closed-clause `BEGIN ... END`, collateral-clause, conditional-clause, case-clause or loop-clause.

Chapter 5

Stowed modes



Ruins of a garum factory in Almuñécar, Andalucía, Spain. Spanish garum was exported to Rome.

Photo © Marcel van der Veer 2008.

5.1 Introduction

”Stowed” is a portmanteau word for **st**ructured or r**owed**. Up to now, we have dealt with plain values, that is, values with modes `INT`, `REAL`, `BOOL` or `CHAR`. In this chapter, we start building more complicated modes: ordered sets of elements. This chapter will introduce ”basic” modes `STRING` and `COMPLEX`, together with the operations that are defined for them. At the end of the chapter also `BITS` and `BYTES` will be introduced.

The simplest example of a compounded mode is the row, which is an ordered set of elements of a same mode. For example, text is a string of characters and many of us use vectors and matrices. In this chapter, we also introduce a mode constructor for making structured modes. A structured mode is an ordered set of objects, not necessarily of a same mode.

5.2 Rows

A row consists of a number of elements, of a same mode which understandably cannot be `VOID`. The mode of a row consists of the mode indicant for each element preceded by brackets, and is said ”row of mode”. For example, here is an identity declaration of a row of `CHAR`:

```
[] CHAR a = "tabula materna combusta est"
```

The left-hand side of the equals-symbol is read ”row of char a”. The statement on the right-hand side of the equals-symbol is the denotation of a value whose mode is `[] CHAR`. Spaces can, of course, appear before, between or after the brackets. Note that we can use a formal-declarer in the identity-relation since `a` will just be an alias of `"tabula materna combusta est"`.

If you want to declare a row variable, you will have to supply bounds. The declarer must be an actual-declarer. For instance, you *must* write

```
[1 : 5] CHAR a := "Algol"
```

even though the bounds are implicit in this specific example. The difference with an identity-declaration is that if a row is the source of an assignation, also when appearing as the initialisation in an variable-declaration, the row is copied into the destination and the bounds must match. The bounds must match since Algol 68 does not re-generate the destination row (unless the row is flexible — see section 5.13 in this chapter). Therefore you will have to supply the bounds. In an identity-declaration you only make an alias for a row-descriptor, which does not involve copying a row.

Rows of mode `[] CHAR` are so common that this denotation was devised as a kind of shorthand. The maximum number of elements in a row is equal to the maximum positive integer (`max int`), although in practice, your program will be limited by the available memory. The denotation of a `[] CHAR` may extend over more than one line. In general, *a68g* will concatenate a line ending in a backslash `\` with the next line. Here is a declaration which exemplifies this:

```
[] CHAR long1 = "The first stage in the develo\
pment of a new program consists of analysing \
the problem that the program must solve.";
```

Note that the second method is neater because you can indent the subsequent parts of the denotation. Everything ″between″ the second and third quote characters and ″between″ the fourth and fifth quote characters is ignored, although you should not put anything other than spaces or tabs and new lines there. If you want to place a quote character `""` in the denotation, you must double it, just as in the character denotation. Here are two `[] CHAR` denotations, each containing two quote characters:

```
[] CHAR rca = """Will you come today?""",
        rcb = "The minority report stated \
that ""in their opinion""";
```

The repeated quote characters are different from the quote characters which chain the two parts of the denotation of `rcb`.

5.3 Row-displays

Rows of other modes cannot be denoted as shown above, but are denoted by a construct called a row-display. A row-display consists of none or two or more units separated by commas and enclosed by parentheses (or `BEGIN` and `END`). Here is the identity declaration for `a` written using a row-display:

```
[] CHAR a = ("a", "b", "c", "d")
```

It is important to note that the units in the row-display could be quite complicated. For example, here is another declaration for a row with mode `[] CHAR`:

```
[] CHAR b = ("a", "P", REPR 36, """")
```

In each of these two declarations, the number of elements is 4.

Here are identity declarations for a row of mode `[] INT` and a row of mode `[] REAL`:

```
[] INT  c = (1, 2 + 3, -2 ** 4, 7, -11, 2, 1);
[] REAL d = (1.0, -2.9, 3e4, (x > 0 | x | -x), -5)
```

In a row of mode `[] INT`, the individual elements can have any value of mode `INT`: that is, any unit yielding an integer. In `d`, the unit yielding the last element is written as a formula yielding a value of mode `INT`. Since a row-display is only allowed in a strong context, such as the right-hand side of an identity declaration, its constituent units are also in a strong context. Thus, the context of the formula is also strong, and so the value yielded by the formula is widened.

An empty row-display can be used to yield a flat row (a row with no elements). For example, here is an identity declaration using an empty row-display:

```
[] REAL empty = ()
```

The denotation for a flat `[] CHAR` is used in the identity declaration

```
[] CHAR none = ""
```

A row can also have a single element. However, a row-display cannot have a single unit (because it would be a closed-clause, which is a different construct). In this case and in a strong context, we use a unit for the only element; the value of that unit is coerced to a row with a single element using the rowing coercion. For example,

```
[] INT ri = 0
```

yields a row with one element. A closed-clause can be used instead:

```
[] INT ri1 = (4)
```

since a closed-clause is also a unit (see section 9.4).

Rowing can only occur in strong contexts. For instance, the right-hand side of an identity declaration is a strong context:

```
[] CHAR rc = "p"
```

A row-display can only be used in a strong context. Because the context of an operand is firm, a row-display cannot appear in a formula (but there is a way round this using a cast, see section 9.6). The shorthand denotation for a `[] CHAR` is not a row-display and so does not suffer from this limitation.

5.4 Dimensions

One of the properties of a row is its number of dimensions. All the rows declared so far have one dimension. The number of dimensions affects the mode. A two-dimensional row of integers has the mode

```
[, ] INT
```

(said ″row-row-of-int″), while a 3-dimensional row of reals (real numbers) has the mode

```
[, , ] REAL
```

Note that the number of commas is always one less than the number of dimensions. In Algol 68, rows of any number of dimensions can be declared.

To cater for more than one dimension, each of the units of a row-display can also be a row-display. For example, the row-display for a row with mode `[, ] INT` could be

```
((1, 2, 3), (4, 5, 6))
```

The fact that this is the row-display for a 2-dimensional row would be clearer if it were written

```
((1, 2, 3),
 (4, 5, 6))
```

For two dimensions, it is convenient to talk of ″rows″ and ″columns″. Here is an identity declaration using the previous row-display:

```
[, ] INT e = ((1, 2, 3),
              (4, 5, 6))
```

The first ″row″ of `e` is yielded by the row-display $(1, 2, 3)$ and the second ″row″ is yielded by $(4, 5, 6)$. The first ″column″ of `e` is yielded by the row-display $(1, 4)$, the second ″column″ by $(2, 5)$ and the third ″column″ by $(3, 6)$. Note that the number of elements in each ″row″ is the same, and the number of elements in each ″column″ is also the same, but that the number of ″rows″ and ″columns″ differ. We say that `e` is a rectangular row. If the number of ″rows″ and ″columns″ are the same, the row is said to be square. Here is an identity declaration for a square row:

```
[, ] CHAR f = (("a", "b", "c"),
               ("A", "B", "C"),
               ("1", "2", "3"))
```

All square rows are also rectangular, but the converse is not true. Note that in the row-display for a multi-dimensional row of characters, it is not possible to use the special denotation for a `[] CHAR`.

The base mode of a row can be any mode, including another row mode. For example:

```
[][] CHAR days = ("Monday","Tuesday","Wednesday",
                  "Thursday","Friday","Saturday",
                  "Sunday")
```

The mode is said "row of row of CHAR". Note that days is one-dimensional, each element consisting of a one-dimensional [] CHAR. The shorthand denotation for a [] CHAR can be used in this case. Because the base mode is [] CHAR, the individual [] CHARs can have different lengths. Here is another example using integers:

```
[][] INT trapezium = ((1, 2), (1, 2, 3), (1, 2, 3, 4))
```

5.5 Subscripts and bounds

Each element of a row has one integer associated with it for each dimension. These integers increase by 1 from the first to the last element in each dimension. For example, in the declaration

```
[] INT r1 = (90, 95, 98)
```

the integers associated with the elements are [1], [2] and [3] (see the next section for an explanation of why the integers are written like this). Remember that the first element in a row-display always has an associated integer of [1]. These integers are known as subscripts or indexers. Thus the subscript of 98 in r1 is [3]. In the two-dimensional row

```
[, ] INT r2 = ((-40, -30, -20),
               (100, 130, 160))
```

the subscripts for -40 are [1, 1] and the subscripts for 160 are [2, 3].

We say that the lower bound of r1 is 1, and its upper bound is 3. The row r2 has a lower bound of 1 for both the first and second dimensions, an upper bound of 2 for the first dimension (2 "rows") and an upper bound of 3 for the second dimension (3 "columns"). We will write the bounds of r1 and r2 as [1 : 3] and [1 : 2, 1 : 3] respectively. The bounds of a flat row, unless specified otherwise (see the section on trimming), are [1 : 0].

The bounds of a row can be interrogated using the operators LWB for the lower bound, and UPB for the upper bound. The bounds of the first, or only, dimension can be interrogated using the monadic form of these operators. For example, using days defined above, LWB days yields 1, and UPB days yields 7. Where the row is multi-dimensional, the bounds are interrogated using the dyadic form of LWB and UPB: the left operand is the dimension while the right operand is the identifier of the row. For example, 1 UPB r2 yields 2 and 2 UPB r2 yields 3. The priority of the dyadic operators is 8.

An extension of Algol 68 provided by *a68g* are the monadic- and dyadic operator ELEMS that operate on any row. The dyadic version gives the number of elements in the specified dimension of a row, and the monadic version yields the total number of elements of a row. For example:

```
[1 : 10, -5 : 5] INT r;
```

```
print (1 ELEMS r);  # prints +10  #
print (2 ELEMS r);  # prints +11  #
print (ELEMS r)     # prints +110 #
```

The monadic operator returns the total number of elements while the dyadic operator returns the number of elements in the specified dimension, if this is a valid dimension.

The loop-clause will be used frequently to operate on elements of rows:

```
REAL abs := 0;
FOR k FROM LWB vector TO UPB vector
DO abs +:= vector[k] ^ 2
OD
```

Note the use of the `LWB` and `UPB` operators. If you try to access an element whose subscript is greater than the upper bound or less than the lower bound, the program will fail at run-time with an appropriate error message. Using the bounds interrogation operators `LWB`, `UPB` and `ELEMS` is useful because

1. it ensures that your program does not try to use a subscript outside the bounds of the row.
2. if the bounds are changed when the program is being maintained, the loop-clause can remain unchanged. This simplifies the maintenance of Algol 68 programs.
3. a compiler could forego bounds checking. For large rows, this can speed up processing considerably.

5.6 Slicing

In the previous section, it was mentioned that a subscript is associated with every element in a row. The lower-bound of the row for a dimension determines the minimum subscript for that dimension and the upper-bound for that dimension determines the maximum subscript. Thus there is a set of subscripts for each dimension. The individual elements can be accessed by quoting all the subscripts for that element. For example, the elements of the row

```
[] INT odds = (1, 3, 5)
```

can be accessed as `odds[1]`, `odds[2]` and `odds[3]`.

For one-dimensional rows, *a68g* offers the operator `ELEM` which provides compatibility with the vintage *ALGOL68C* compiler. This is an example of a subscript using `ELEM`:

```
[] INT first primes = (1, 3, 5, 7, 11);
```

```
print (first primes[1]);
print (1 ELEM first primes); # prints two times +1 #
```

In a multi-dimensional row, two or more subscripts are required to access a single element, the subscripts being separated by commas. For example, in the row

```
[, ] REAL rs = ((1.0, 2.0, 3.0),
                (4.0, 5.0, 6.0))
```

`rs[1, 2]` yields 2.0. Similarly, `rs[2, 3]` yields 6.0. Thus one can declare

```
REAL rs12 = rs[1, 2],
     rs23 = rs[2, 3]
```

In the example above a slice selects a single element from a row. A slice can also select a sub-row from a row. For example, using `rs` declared above, we can write

```
[] REAL srs = rs[1, ]
```

which yields the row denoted by $(1.0, 2.0, 3.0)$. The comma must be present in the slice on the right-hand side otherwise the interpreter will report an error. Note that the absence of an indexer implicitly selects all elements for that dimension.

Vertical slicing is also possible. The statement `rs[, 2]` yields the row `(2.0, 5.0)`. In the context of the declaration

```
[, ] CHAR rs2 = (("a", "b", "c", "d"),
                 ("e", "f", "g", "h"),
                 ("i", "j", "k", "l"))
```

the slice `rs2[, 3]` yields the value `"cgk"` with a mode of `[] CHAR`. Note, however, that vertical slicing is only possible for rows with at least two dimensions. The row `days`, declared in the previous section, is one-dimensional and so cannot be sliced vertically.

In a 3-dimensional row, both 2-dimensional and 1-dimensional slices can be produced. Here are some examples:

```
[, , ] INT r3 = (((1, 2), (3, 4), ((5, 6), (7, 8)));
[, ] INT r31 = r3[1, , ],
         r32 = r3[, 2, ],
         r33 = r3[, , 3];
[] INT r312 = r31[2, ], r4 = r31[, 2]
```

Example

Given the declaration

```
[, ] INT r = ((1, 2, 3, 4),
              (5, 6, 7, 8),
              (9, 10, 11, 12),
              (13, 14, 15, 16))
```

these are the yields of different slices:

1. `r[2, 2]` yields 6
2. `r[3, ]` yields $(9, 10, 11, 12)$
3. `r[, 2 UPB r]` yields $(4, 8, 12, 16)$
4. 10 is yielded by `r[3, 2]`
5. $(5, 6, 7, 8)$ is yielded by `r[2, ]`
6. $(3, 7, 11, 15)$ is yielded by `r[, 3]`

If you slice a name, you of course want the sliced element to be a name as well, otherwise you could not assign to an element of the row. The important rule in Algol 68 is that if you slice an object of mode `REF [...] MODE`, the yield will be of mode `REF MODE`. So if you write

```
[1 : products] REAL price;
```

you can write

```
price[1] := 0;
```

since slicing a `[] REAL` variable yields a `REAL` variable.

Hence, if you slice an object of mode `[...] MODE`, the yield will be of mode `MODE`. If you slice an object of mode `REF [...] MODE`, the yield will be of mode `REF MODE`. But if you slice an object of mode `REF REF [...] MODE`, the yield will still be of mode `REF MODE`. This coercion is called weak-dereferencing and is explained in section 9.3.3.

5.7 Trimming

The bounds of a slice can be changed using the `@` construction. For example, in the declaration

```
[] CHAR digits = "0123456789"[@0]
```

the bounds of `digits` are `[0 : 9]`. Bounds do not have to be non-negative. For example,

```
[, ] INT ii = ((1, 2, 3), (4, 5, 6));
[, ] INT jj = ii[@-3, @-50]
```

whence the bounds of `jj` are `[-3 : -4, -50 : -48]`. Note that you cannot change the bounds of a row-display (except by using a cast — see section 9.6). For now, declare an identifier for the display, and then alter the bounds. The bounds of a slice can be changed:

```
[,] INT ij = ((1, 3, 5), (7, 9, 11), (13, 15, 17));
 [] INT ij2 = ij[2, ][@0]
```

The declaration for `ij2` could also be written

```
[] INT ij2 = ij[2, @0]
```

`@` can also be written `AT`.

Wherever an integer is required in the above, any unit yielding an integer will do. Thus it is quite in order to use the formula

```
(a + b) UPB r
```

where the parentheses are necessary if `a + b` is expected to yield the dimension of `r` under consideration (because the priority of `UPB` is greater than the priority of `+`).

A trimmer uses the `:` construction. In the context of the declaration of `digits` above, the statement `digits[1 : 3]` yields the value `"123"` with mode `[] CHAR`.

Trimming is particularly useful with values of mode `[] CHAR`. Given the declaration

```
[] CHAR quote = "Habent sua fata libelli"
```

(the quotation at the start of the acknowledgements in the ”Revised Report”),

```
quote[: 6]
quote[8 : 10]
quote[12 : 15]
```

yield the first three words. Note that when the first subscript in a trimmer is omitted, the lower bound for that dimension is assumed, while omission of the second subscript assumes the corresponding upper bound. Again, any unit yielding `INT` may be used for the trimmers. The context for a trimmer or a subscript is meek.

Omission of both subscripts yields the whole slice with a lower bound of 1. So, the upper bound of the statement `digits[ : ]` is 10 which is equivalent to `digits[@1]`.

The lower bound of a `trimmer` is, by default, 1, but may be changed by the use of `@`. For example, `digits[3 : 6]` has bounds `[1 : 4]`, but `digits[3 : 6@2]` has bounds `[2 : 5]`.

The bounds of `quote[17 :]` mentioned above are `[1 : 7]`.

a68g allows you to replace the colon `:` by `..` in bounds and trimmers, which is the Pascal style. Hence in *a68g* next trims are identical:

```
quote[: 6]     and quote[.. 6]
quote[8 : 10]  and quote[8 .. 10]
quote[12 : 15] and quote[12 .. 15]
```

5.8 Reading and printing rows

We have already used `print` to convert plain values to characters displayed on your screen. Actually, `print` takes a row of values to be output, so it is quite valid to write

```
[] INT i1 = (2, 3, 5, 7, 11, 13);
print(i1);
print(new line)
```

or equivalently, using a row-display

```
print((2, 3, 5, 7, 11, 13, new line))
```

The doubled parentheses are necessary: the outer pair are needed by `print`, and the inner pair are part of the row-display. Note that the modes of the elements of the row-display are quite different. We will learn in chapter 7 how that can be so.

With respect to rows, `read` behaves just like `print` in that a whole row can be read at one go. The only difference between the way `read` is used and the way `print` is used is that the values for `read` must be names (or identifiers of names) whereas `print` can use denotations or identifiers of names or identifiers which are not names. In case of multi-dimensional rows, the elements are transput in row-order, that is, the rightmost subscript varies most frequently.

5.9 Operators with rows

Rows of `CHAR` occur so often, that two dyadic operators are available for them.

The operator `+` is defined for all combinations of `CHAR` and `[] CHAR`. Thus, the formula

```
"abc" + "d"
```

yields the value denoted by `"abcd"`. With these operands, `+` acts as a concatenation operator. The operator has a priority of 6 as before.

Multiplication of values of mode CHAR or [] CHAR is defined using the operator *. The other operand has mode INT and the yield has mode [] CHAR. For example, in the declaration

```
[] CHAR repetitions = "ab" * 3
```

repetitions identifies "ababab". The formula could have been written with the integer as the left operand. In both cases, the operator only makes sense with a positive integer.

The operators = and /= are also defined for operands of mode [] CHAR. The two rows must have the same number of elements, and corresponding elements must be equal if the operator is to yield TRUE. Thus

```
"a" = "abc"
```

yields FALSE. Note that the bounds do not have to be the same. So a and b declared as

```
[] CHAR a = "Dodo"[@0], b = "Dodo"
```

yield TRUE when compared with the equals operator. Because the rowing coercion is not allowed in formulas, the operator is declared in the standard-prelude for mixed modes (such as CHAR and [] CHAR).

The inverse of = is /= (not equal). So the formula

```
"r" /= "r"
```

yields FALSE. Alternative representations of /= are =, ^= and NE.

The ordering operators <, >, <= and >= can be used to compare values of mode [] CHAR in the same way as = and /=.

For values of mode [] CHAR, ordering is alphabetic. The formula

```
"abcd" > "abcc"
```

yields TRUE. Two values of mode [] CHAR of different length can be compared. For example, both

```
"aaa" <= "aaab"
```

and

```
"aaa" <= "aaaa"
```

yield TRUE. Alternative representations for these operators are LT and GT for < and > and LE and GE for <= and >= respectively.

Note that apart from values of mode [] CHAR, no operators are defined in the Revised Report for rows.

5.10 Names of rows

Here is an identity declaration for a name referring to a name:

```
REF [] INT i7 = LOC [1 : 7] INT
```

which can be abbreviated to

```
[1 : 7] INT i7
```

There are two things to notice about the first declaration. Firstly, the mode on the left-hand side is known as a formal-declarer. It says what the mode of the name is, but it says nothing about how many elements there will be in any row to be assigned, nor what its bounds will be. In fact, only formal-declarers are used on the left-hand side of *any* identity-declaration.

Secondly, the generator on the right-hand side is an actual-declarer. It specifies how many elements can be assigned. In fact, the trimmer represents the bounds of the row which can be assigned. If the lower bound is 1 it may be omitted, so the above declaration could well have been written

```
REF [] INT i7 = LOC [7] INT
```

which can be read as ”ref row of int i7 equals loc row of seven int”. This declaration can be abbreviated to

```
[7] INT i7
```

The bounds of a row do not have to start from 1. In this identity declaration

```
REF [] INT i7 at 0 = HEAP [0 : 6] INT
```

or, equivalently,

```
HEAP [0 : 6] INT i7
```

the bounds of the row will be `[0 : 6]`.

5.11 Dynamic names

Up to now, all the names which can refer to rows were declared with bounds whose values were given by integer denotations. In fact, the bounds given on the right-hand side of the identity declaration can be any unit which yields an integer in a meek context. So it is quite reasonable to write

```
INT size;
read(size);
REF [] INT a = LOC [1 : size] INT
```

or even

```
REF [] INT r = LOC [1 : (INT size; read(size); size)] INT
```

or the equivalent but short form

```
[1 : read int] INT r
```

since an enclosed serial-clause has the value of its last unit. The value of the clause in the parentheses is a name of mode `REF INT` and since the context of the clause is meek, dereferencing is allowed. The context is passed on to the last unit in the clause. Thus the integer read by `read` will be passed to the generator.

A dynamic name is one which can refer to a row whose bounds are determined at the time the program is elaborated. It means that you can declare names referring to rows of the size you actually require, rather than the maximum size that you might ever need.

5.12 Assigning to row names

We can assign values to the elements of a row either individually or collectively.

5.12.1 Individual assignation

We can access an individual element of a row by subscripting that element. For example, suppose that we wish to access the third element of `i7` as declared in the last section. The rules of the language state that a subscripted element of a row name is itself a name. In fact, the elaboration of a slice of a row name creates a new name. Thus the mode of `i7[3]` is `REF INT`. We can assign a value to `i7[3]` by placing the element on the left-hand side of an assignation:

```
i7[3] := 4
```

Unless you define a new identifier for the new name, it will cease to exist after the above assignation has been elaborated (see below for examples of this).

Since each element of `i7` has an associated name (created by slicing) of mode `REF INT`, it can be used in a formula:

```
i7[2] := 3 * i7[i7[1]] + 1
```

As you can see, an element was used to compute a subscript. It has been presumed that the value obtained after dereferencing lies between 1 and 7 inclusive. If this were not so, a run-time error would be generated. In the above assignation, all elements on the right-hand side of the assignation would be dereferenced before being used in the formula. Note that subscripting (or slicing or trimming) binds more tightly than any operator.

5.12.2 Collective assignation

There are two ways of assigning values collectively. Firstly, it can be done with a row-display or a `[] CHAR` denotation. For example, using the declaration of `i7` above:

```
i7 := (4, -8, 11, ABS "K", ABS TRUE, 0, ROUND 3.4)
```

Note that the bounds of both `i7` and the row-display are `[1 : 7]`. In the assignation of a row, the bounds of the row on the right-hand side must match the bounds of the row name on the left-hand side. If they differ, a fault is generated. If the bounds are known at compile-time, the compiler will generate an error message. If the bounds are only known at run-time (see section 5.11 on dynamic names), a run-time error will be generated. The bounds can be changed using a trimmer or the `@` symbol (or `AT`).

The second way of assigning to the elements of a row collectively is to use an identifier of a row, or any unit yielding a row of the correct mode for that matter, with the required bounds. For example:

```
[] INT i3 = (1, 2, 3);
REF [] INT k = LOC [1 : 3] INT := i3
```

The right-hand side has been assigned to the row name `k`.

As mentioned above, parts of a row can be assigned using slicing or trimming. For example, given the declarations

```
REF [, ] REAL x = LOC [1 : 3,1 : 3] REAL,
              y = LOC [0 : 2, 0 : 2] REAL
```

and the assignation

```
x := ((1, 2, 3),
      (4, 5, 6),
      (7, 8, 9))
```

we can write

```
y[2, 0] := x[3, 2]
```

The row name `y` is sliced yielding a name of mode `REF INT`. Then the row name `x` is sliced also yielding a name of mode `REF INT` which is then dereferenced yielding a new instance of the value to which it refers (8) which is then assigned to the new name on the left hand side of the assignation. To save execution time when you repeatedly slice the same name, you could store the new name by means of an identity-declaration:

```
REF INT y20 = y[2, 0];
y20 := x[3, 2]
```

Here are some examples of slicing with (implied) row assignations:

```
y := x[@0, @0];
y[2, ] := x[1, @0];
y[, 1] := x[2, @0]
```

In the first example, the right-hand side is a slice of a name whose mode is `REF [, ] REAL`. Because the slice has no trimmers its mode is also `REF [, ] REAL`. Using the `@` symbol, the lower bounds of both dimensions are changed to 0, ensuring that the bounds of the row name thus created match the bounds of the row name `y` on the left. After the assignation (and the dereferencing), `y` will refer to a copy of the row `x` and the name created by the slicing will no longer exist.

In the second assignation, the row `x` has been sliced yielding a name whose mode is `REF [] REAL`. It refers, in fact, to the first ″row″ of `x`. The `@0` ensures that the lower bound of the second dimension of `x` is 0. The left-hand side yields a name of mode `REF [] REAL` which refers to the last ″row″ of the row `y`. The name on the right-hand side is dereferenced. After the assignation `y[2,]` will refer to a copy of the first ″row″ of `x` and the name produced by the slicing will no longer exist.

In the third assignation, the second ″row″ of `x` is assigned to the second ″column″ of `y`. Again, the `@0` construction ensures that the lower bound of the second dimension of `x` is zero. After the assignation, the name created by the slicing will no longer exist.

Note how the two declarations for `x` and `y` have a common formal-declarer on the left-hand side, with a comma between the two declarations. This is a common abbreviation. The comma means that the two declarations are elaborated collaterally (and on a parallel processing computer, possibly in parallel).

At the end of this section we demonstrate how row-variables and trimming can be used in a practical situation:

```
PR echo "Circular assignment of a string" PR

BEGIN STRING u := "Barbershop";
      TO UPB u + 1
      DO print((u, new line));
         CHAR v = u[UPB u]; STRING w = u[1 : UPB u - 1];
         u[2 : UPB u] := w;
         u[1] := v
      OD
END
```

which produces

```
$ a68g barbershop.a68
Circular assignment of a string
Barbershop
pBarbersho
opBarbersh
hopBarbers
shopBarber
rshopBarbe
ershopBarb
bershopBar
rbershopBa
arbershopB
Barbershop
```

5.13 Flexible names

In the previous section, we declared row names. The bounds of the row to which the name can refer are included in the generator. In subsequent assignations, the bounds of the new row to be assigned must be the same as the bounds given in the generator. In Algol 68, it is possible to declare names which can refer to a row of any number of elements (including none) and, at a later time, can refer to a different number of elements. These are called flexible names. Here is an identity declaration for a flexible name:

```
REF FLEX [] INT fn = LOC FLEX [1 : 0] INT
```

or, abbreviated

```
FLEX [1 : 0] INT fn
```

There are several things to note about this declaration. Firstly, the mode of the name is not `REF [] INT`, but `REF FLEX [] INT`. `FLEX` means that the bounds of the row to which the name can refer can differ from one assignation to the next. Secondly, the bounds of the name generated at the time of the declaration are `[1 : 0]`. Since the upper bound is less than the lower bound, the row is said to be flat; in other words, it has no elements at the time of its declaration. Thirdly, `FLEX` is present on both sides of the identity declaration (but in the last section of this chapter we will see a way round that).

We can now assign rows of integers to `fn`:

```
fn := (1, 2, 3, 4)
```

The bounds of the row to which `fn` now refers are `[1 : 4]`. Again, we can write

```
fn := (2, 3, 4)
```

Now the bounds of the row to which `fn` refers are `[1 : 3]`. We can even write

```
fn := 7
```

in which the right-hand side will be rowed to yield a one-dimensional row with bounds `[1 : 1]`, and

```
fn := ()
```

giving bounds of `[1 : 0]`.

In the original declaration of `fn` the bounds were `[1 : 0]`. The compiler will not ignore any bounds other than `[1 : 0]`, but will generate a name whose initial bounds are those given. So the declaration

```
REF FLEX [] INT fn1 = LOC FLEX [1 : 4] INT
```

will cause `fn1` to have the bounds `[1 : 4]` instead of `[1 : 0]`.

The lower bound does not have to be 1. In this example,

```
REF [] INT m1 = LOC [-1 : 1] INT;
...
REF FLEX [] INT f1 = LOC FLEX [1 : 0] INT := m1
```

the bounds of `f1` after the initial assignation are `[-1 : 1]`.

If a flexible name is sliced or trimmed, the resulting name is called a transient name because it can only exist so long as the flexible name stays the same size. Such names have a restricted use to avoid the production of names which could refer to nothing. For example, consider the declaration and assignation

```
REF FLEX [] CHAR c1 = LOC FLEX [1 : 0] INT;
c1 := "abcdef";
```

Suppose now we have the declaration

```
REF [] CHAR lc1 = c1[2 : 4]; #WRONG#
```

followed by this assignation:

```
c1 := "z";
```

It is clear that `lc1` no longer refers to anything meaningful. Thus transient names cannot be assigned without being dereferenced, nor given identifiers, nor used as parameters for a routine (whether operator or procedure). However there is nothing to prevent them being used in an assignation. For example,

```
REF FLEX [] CHAR s = LOC [1 : 0] CHAR := "abcdefghijklmnopqrstuvwxyz";
s[2 : 7] := s[9 : 14]
```

where the name yielded by `s[9 : 14]` is immediately dereferenced. Note that the bounds of a trim are fixed even if the value trimmed is a flexible name. So the assignation

```
s[2 : 7] := "abc"
```

would produce a run-time fault.

What has not been made apparent up to now is a problem arising from the ability that we can have rows of any mode except `VOID`, so also rows of rows, et cetera. Now consider the declaration

```
FLEX [1 : 0][1 : 3] INT flexfix
```

Because the mode of `flexfix` is `REF FLEX [][] INT`, when it is subscripted, the mode of each element is `REF [] INT` with bounds of `[1 : 3]`. Clearly, after the declaration, `flexfix` has no elements, so how would we know about dimensions of the `REF [] INT` sub-row of `flexfix`? According to the Revised Report a row must have a ”ghost” element, inaccessible by subscripting, to preserve information on bounds in case no elements are present. This ghost element prohibits writing

```
flexfix := LOC [1 : 4][1 : 4] REAL
```

although *a68g* will allow this last assignation as it does not implement ghost elements.

5.14 The mode STRING

The mode `STRING` is defined in the standard-prelude as having the same mode as the expression `FLEX [1 : 0] CHAR`. That is, the identity declaration

```
REF STRING s = LOC STRING
```

has exactly the same effect as the declaration

```
REF FLEX [] CHAR s = LOC FLEX [1 : 0] CHAR
```

You will notice that although the mode indicant `STRING` appears on both sides of the identity declaration for `s`, in the second declaration the bounds are omitted on the left-hand side (the mode is a formal-declarer) and kept on the right-hand side (the actual-declarer). Without getting into abstruse grammatical explanations, just accept that if you define a mode like `STRING`, whenever it is used on the left-hand side of an identity declaration the compiler will ignore the bounds inherent in its definition.

We can now write

```
s := "String"
```

which gives bounds of `[1:6]` to `s`. We can slice that row to get a value with mode `REF CHAR` which can be used in a formula. If we want to change the bounds of `s`, we must assign a value which yields a value of mode `[] CHAR` to the whole of `s` as in `s := "Another string"` or `s := s[2 : 4]`

Often, where `[] CHAR` appears, it may be safely replaced by `STRING`. This is because it is only names which are flexible so the flexibility of `STRING` is only available in `REF STRING` declarations.

There are two operators defined in the standard-prelude which use an operand of mode `REF STRING`: `PLUSAB`, whose left operand has mode `REF STRING` and whose right operand has mode `STRING` or `CHAR`, and `PLUSTO`, whose left operand has mode `STRING` or `CHAR` and whose right operand has mode `REF STRING`. Using the concatenation operator `+`, their actions can be summarised as follows:

1. `a PLUSAB b` means `a := a + b`
2. `a PLUSTO b` means `b := b + a`

Thus `PLUSAB` concatenates `b` onto the end of `a`, and `PLUSTO` concatenates `a` to the beginning of `b`. Their alternative representations are `+:=` and `+=:` respectively. For example, if `a` refers to `"abc"` and `b` refers to `"def"`, after `a PLUSAB b`, `a` refers to `"abcdef"`, and after `a PLUSTO b`, `b` refers to `"abcdefdef"` (assuming the `PLUSAB` was elaborated first).

5.15 Vectors, matrices and tensors

People interested in mathematical calculations will use rows to represent vectors, matrices and tensors. Algol 68 Genie has extensions to support these objects when they are represented as rows. Next section shows how to extract a transpose, or a diagonal from a matrix. The library-prelude (see section 11.12) provides a simple vector and matrix interface to Algol 68 rows of mode:

```
[] REAL       # vector #
[, ] REAL     # matrix #
[] COMPLEX    # complex vector #
[, ] COMPLEX  # complex matrix #
```

These routines require the GNU Scientific Library.

5.16 Torrix extensions

a68g implements pseudo-operators `TRNSP`, `DIAG`, `COL` and `ROW` as described by [Torrix 1977]. These are of particular interest to vector - and matrix algebra. Original Torrix code implements these symbols as operators on one- and two-dimensional rows of real and complex numbers. The pseudo-operator implementation offered by *a68g* is more general as it works on one- and two-dimensional rows of any mode. The syntactic position of these pseudo-operator expressions is at the same level as a formula, which is a `tertiary` as we will see in a forthcoming chapter.

Next list compiles the definitions of these pseudo-operators. Note that all yield a descriptor. This means that the yield of the pseudo-operator refers to the same elements as the row operated on, only the indices are mapped. In the list below `a` is a two-dimensional row,

$$a = \begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

`u` is a one-dimensional row,

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \cdots \end{pmatrix}$$

and `i`, `j` and `k` are integers.

1. `TRNSP` constructs, without copying, a descriptor such that

   ```
   (TRNSP a)[j, i] = a[i, j]
   ```

for valid `i` and `j`.

2. `DIAG` constructs, without copying, a descriptor such that

   ```
   (k DIAG a)[i] = a[i, i + k]
   ```

   for valid `i` and `k`. The monadic form of `DIAG` is equivalent to `0 DIAG ...` .

3. `COL` constructs, without copying, a descriptor such that

   ```
   (k COL u)[i, k] = u[i]
   ```

   for valid `i` and `k`. The monadic form of `COL` is equivalent to `1 COL ...` .

4. `ROW` constructs, without copying, a descriptor such that

   ```
   (k ROW u)[k, i] = u[i]
   ```

   for valid `i` and `k`. The monadic form of `ROW` is equivalent to `1 ROW ...` .

These pseudo-operators yield a new descriptor, but do no copy data. They give a new way to address the elements of an already existing one - or two dimensional row. For example, next code sets the diagonal elements of a matrix:

```
[3, 3] REAL matrix;
DIAG matrix := (1, 1, 1);
print(DIAG matrix)
```

This will print three zeroes as the result of the assignation. These pseudo-operators delivering new descriptors to existing data cannot be coded in standard Algol 68.

There is an analogy with slicing a name. If you apply an above pseudo-operator to an object of mode `[...] MODE`, the yield will be of mode `[...] MODE`, but the number of dimension will of course change. If you operate on an object of mode `REF [...] MODE`, the yield will be of mode `REF [...] MODE`. But if you operate on an object of mode `REF REF [...] MODE`, the yield will still be of mode `REF [...] MODE`. This coercion is called weak-dereferencing and is explained in section 9.3.3.

5.17 Reading rows

If `read` is used to read a `[] CHAR` with fixed bounds as in

```
REF [] CHAR sf = LOC [80] CHAR;
read(sf)
```

then the number of characters specified by the bounds will be read, `new line` and `new page` being called as needed. You can call `new line` and `new page` explicitly to ensure that the next value to be input will start at the beginning of the next line or page.

The only flexible name for which `read` can be used is `REF STRING`. When reading values for `REF STRING`, the reading pointer will not go past the end of the current line.[1] If the reading position is already at the end of the line, the row will have no elements. When reading a `STRING`, `new line` must be called explicitly for transput to continue. The characters read are assigned to the name.

5.18 Structured modes

We have already seen how a number of individual values can be collected together to form a row whose mode was expressed as ”row of” mode. The principal characteristic of rows is that all the elements have the same mode. Often the value of an object cannot be represented by a value of a single object of a standard mode. Think for instance of a book, that has an author, a title, year of publication, ISBN, et cetera. A structure is another way of grouping data elements where the individual parts may be of different modes. In general, accessing the elements of a row is determined at run-time by the elaboration of a slice. In a structure, access to the individual parts, called fields, is determined at compile time. In some programming languages, structured modes are called *records*.

The mode constructor `STRUCT` is used to create structured modes. Here is a simple identity declaration of a structure:

```
STRUCT (INT index, STRING title) s = (1, "De bello gallico")
```

The mode of the structure is

```
STRUCT (INT index, STRING title)
```

and its identifier is `s`. The `index` and the `title` are called field-selectors and are part of the mode. They are not identifiers, even though the rule for identifier construction applies to them, because they are not values in themselves. You cannot say that `index` has mode `INT` because `index` cannot stand by itself. We will see in the next section how they are used.

The expression to the right of the equals-symbol is called a structure-display. Like row-displays, structure-displays can only appear in a strong context. In a strong context, *a68g* can determine which mode is required and so it can tell whether a row-display or a structure-display has been provided. We could now declare another such structure:

```
STRUCT (INT index, STRING title) t = s
```

[1]See section 8.5 for details of string terminators.

and `t` would have the same value as `s`.

Here is a structure declaration with different field-selectors

```
STRUCT (INT count, STRING title) ss =
   (1, "Reflexions sur la puissance motrice du feu")
```

which looks almost exactly like the first structure declaration above, except that the field-selector `index` has been replaced with `count`. The structure `ss` has a different mode from `s` because not only must the constituent modes be the same, but the field-selectors must also be identical.

Structure names can be declared:

```
REF STRUCT (INT index, STRING title) sn =
   LOC STRUCT (INT index, STRING title)
```

Because the field-selectors are part of the mode, they appear on both sides of the declaration. The abbreviated form is

```
STRUCT (INT index, STRING title) sn
```

We could then write

```
sn := s
```

in the usual way, but not

```
sn := ss
```

The modes of the fields can be any mode except `VOID`. For example, we can declare

```
STRUCT (REAL x, REAL y, REAL z) vector
```

which can be abbreviated to

```
STRUCT (REAL x, y, z) vector
```

and here is a possible assignation:

```
vector := (-1.0, 0, 1.0)
```

where the value 0 would be widened to `0.0`.

A structure can also contain another structure:

```
STRUCT (STRING c, STRUCT (REAL x, y) point) ori = ("O", (0, 0))
```

In this case, the inner structure has the field-selector `point` and *its* field-selectors are `x` and `y`.

The mode of a field-selector cannot be `VOID`. If size of rows is relevant, as in generators and variable-declarations, the mode of a field-selector is an actual-declarer. Otherwise, as in an identity-declaration, the mode is a formal-declarer.

5.19 Field selection

The field-selectors of a structured-mode are used to ”extract” the individual fields of a structure. For example, given this declaration for the structure `s`:

```
STRUCT (INT index, STRING title) s = (1, "De bello gallico")
```

we can select the first field of `s` using the selection

```
index OF s
```

The mode of the selection is `INT` and its value is 1. Note that the construct `OF` is not an operator. The second field of `s` can be selected using the selection

```
title OF s
```

whose mode is `STRING` with value `"De bello gallico"`. The field-selectors cannot be used on their own: they can only be used in a selection.

A selection can be used as an operand. Consider the formula

```
index OF s + 1
```

The two fields of the structure

```
STRUCT (STRING c, STRUCT (REAL x, y) point) ori
```

can be selected by writing

```
c OF ori
point OF ori
```

and their modes are `STRING` and `STRUCT (REAL x, y)` respectively. Now the fields of the inner structure `point` of `ori` can be selected by writing

```
x OF point OF ori
y OF point OF ori
```

and both selections have mode `REAL`.

Now consider the structure name `sn` declared by

```
STRUCT (INT index, STRING title) sn;
```

The mode of `sn` is

```
REF STRUCT (INT index, STRING title)
```

This means that the mode of the selection

```
index OF sn
```

must be `REF INT`, and the mode of the selection

```
title OF sn
```

must be `REF STRING`. That is, the modes of the fields of a structure name get preceded by `REF`. Otherwise you would not be able to assign to a single field in a structured object.

The important general rule is that if you select a field with mode `MODE` from an object with mode `STRUCT (...)`, than the yield will be of mode `MODE` as well. If you select a field with mode `MODE` from an object with mode `REF STRUCT (...)`, than the yield will be of mode `REF MODE`. But if you select a field with mode `MODE` from an object with mode `REF REF STRUCT (...)`, than the yield will still be of mode `REF MODE`. This coercion is called weak-dereferencing and is explained in section 9.3.3.

Thus, instead of assigning a complete structure using a structure-display, you can assign values to individual fields. That is, the assignation

```
sn := (2, "The republic")
```

is equivalent to the assignations

```
index OF sn := 2;
title OF sn := "The republic"
```

except that the two units in the structure-display are separated by a comma and hence are elaborated collaterally.

Given the declaration

```
STRUCT (CHAR mark, STRUCT (REAL x, y) point) ori;
```

the selection

```
point OF ori
```

has the mode `REF STRUCT (REAL x, y)`, and so you could assign directly to it:

```
point OF ori := (0, 0)
```

as well as to its fields:

```
x OF point OF ori := 0;
y OF point OF ori := 0
```

5.20 Mode-declarations

Structure declarations are very common in Algol 68 programs because they are a convenient way of grouping disparate data elements, but writing out their modes every time a name needs declaring is clumsy and error-prone. Using the mode-declaration, a new mode indicant can be declared to act as an abbreviation.

An indicant can be a tag as `RECORD` or `VECTOR`. *a68g* accepts a tag that starts with an upper-case letter, optionally followed by upper-case letters or underscores. Since spaces are not allowed in an upper-case tag to avoid ambiguity, underscores can be used to improve legibility. The use of underscores in tags is not allowed in standard Algol 68.

For example, the mode-declaration

```
MODE VECTOR = STRUCT (REAL x, y, z)
```

makes `VECTOR` synonymous for the mode specification on the right-hand side of the equals-symbol and new objects using `VECTOR` can be declared in the ordinary way:

```
VECTOR vec = (1, 2, 3);
VECTOR vn := vec;
[10] VECTOR va;
MODE TENSOR = STRUCT (VECTOR x, y, z)
```

Suppose you want a mode which refers to another mode which has not yet been declared before, and a second mode that refers back to the first mode, for example:

```
MODE LIST_A = STRUCT (STRING title, REF LIST_B next),
     LIST_B = STRUCT (STRING name, REF LIST_A next)
```

This can for instance not be written in Pascal or C without using some sort of ”forward declaration” (Pascal) or ”incomplete type” (C). In Algol 68, tags do not have to be declared before they are applied, so you can straightforwardly declare the two modes as listed above.

A mode in Algol 68 cannot give rise to (1) an infinitely large object or (2) endless coercion. Using a mode-declaration, you might be tempted to declare a mode such as

```
MODE CIRCULAR = STRUCT (INT i, CIRCULAR c) CO wrong! CO
```

but this is not allowed since a declaration

```
CIRCULAR z;
```

would quickly consume all memory on your system and then complain that memory is exhausted. Next declaration will also give an infinitely large object:

```
MODE BINARY = [1 : 2] BINARY
```

and is therefore not allowed.

However, there is nothing wrong with modes as

```
MODE LIST = STRUCT (STRING s, REF NODE next)
```

because only a reference to itself is declared within the structure.

Mode-declarations are not confined to structures. For example, the mode `STRING` is declared in the standard-prelude as

```
MODE STRING = FLEX [1 : 0] CHAR
```

and you can write declarations like

```
MODE INDEX = INT, POINT = COMPLEX, MATRIX = [n, n] REAL
```

Note that the mode-declarations have been abbreviated (by omitting `MODE` each time and using commas). In the last declaration involving `MATRIX`, the bounds will be elaborated at the time of the declaration of any name of mode `MATRIX`. When declaring a value, the declarer can be formal, and bounds are ignored (and not evaluated). Here, for example, is a small program using `MATRIX`:

```
BEGIN INT n;
      MODE MATRIX = [n, n] REAL;
      WHILE print((new line, "Give the dimension (n): "));
            read(n);
            n > 0
      DO MATRIX m;
         FOR i TO 1 UPB m
         DO FOR j TO 2 UPB m
            DO m[i, j] := 1 / (i + j - 1)
            OD
         OD;
```

```
        print(("Hilbert matrix:", new line, m))
      OD
END
```

5.21 Recursion in mode-declarations

A curiosity of Algol 68 that does not receive a lot of attention is that since a tag can be applied before it is declared, application in actual-bounds of a rowed mode while it is being declared can make a mode-declaration recursive. This may lead to fuzzy code like this example:

```
MODE FAC = [1 : (n > 0 | m *:= n; n -:= 1; LOC FAC; n | 0)] INT;
INT m := 1, n := 10;
LOC FAC;
print ((m, n))
```

Such code actually runs under *a68g*:

```
$ ./a68g faculty.a68
1     MODE FAC = [1 : (n > 0 | m *:= n; n -:= 1; LOC FAC; n | 0)] INT;
                                                 1
a68g: warning: 1: value of REF [] INT generator will be voided
(detected in conditional-clause starting at "(" in this line).
3     LOC FAC;
      1
a68g: warning: 1: value of REF [] INT generator will be voided
(detected in particular-program).
   +3628800         +0
```

Note that two warnings are issued on a value being voided; *a68g* draws your attention to the fact that a name is generated on two occasions that is not stored. Using this type of recursion will be regarded as amusing by some, but most will agree that this is not a recommended programming style.

5.22 Complex numbers

The standard-prelude contains the mode declaration

```
MODE COMPL = STRUCT (REAL re, im)
```

and *a68g* also defines

```
MODE COMPLEX = STRUCT (REAL re, im)
```

Multiprecision declarations exist in *a68g*, just as for REAL:

```
MODE LONG COMPL = STRUCT (LONG REAL re, im);
MODE LONG COMPLEX = STRUCT (LONG REAL re, im)
```

and

```
MODE LONG LONG COMPL = STRUCT (LONG LONG REAL re, im);
MODE LONG LONG COMPLEX = STRUCT (LONG LONG REAL re, im)
```

You can use values with these modes to perform complex arithmetic. Here are example declarations for values of modes COMPL and REF COMPL respectively:

```
COMPL z1 = (2.4, -4.6);
COMPL z2 := z1
```

Most of the operators you need to manipulate complex numbers have been declared in the standard-prelude. You can use the monadic operators + and - which have also been declared for values of mode COMPL. The dyadic operator ** has been declared for a left-operand of mode COMPL and a right-operand of mode INT. The dyadic operators + - * / have been declared for all combinations of complex numbers, real numbers and integers, as are the boolean operators = and /= [2]. The assignation operators TIMESAB, DIVAB, PLUSAB, and MINUSAB all take a left operand of mode REF COMPL and a right-operand of modes INT, REAL or COMPL.

In a strong context, a real number will be widened to a complex number. So, for example, in the following identity declaration

```
COMPL z3 = -3.4
```

z3 will have the same value as if it had been declared by

```
COMPL z3 = (-3.4, 0)
```

The dyadic operator I takes left- and right-operands of any combination of REAL and INT and yields a complex number. It has a priority of 9. For example, in a formula, the context of operands is firm and so widening is not allowed. Nevertheless, the yield of this formula is COMPL:

```
2 * 3 I 4
```

[2]Complex numbers can only be compared for equality. One cannot speak of a complex number being ”larger” than another.

Some operators act only on complex numbers. The monadic operator `RE` takes a `COMPL` operand and yields its `re` field with mode `REAL`. Likewise, the monadic operator `IM` takes an operand of mode `COMPL` and yields its `im` field with mode `REAL`. For example, given the declaration above of `z3`, `RE z3` would yield `-3.4`, and `IM z3` would yield `0.0`. Given a complex number `z` then `CONJ z` yields `RE z I - IM z`. The operator `ARG` gives the argument of its operand in the interval $< -\pi, \pi]$. The monadic operator `ABS` for a complex number is defined as

```
OP ABS = (COMPL z) REAL: sqrt(RE z ** 2 + IM z ** 2)
```

Remember that in the formula `RE z ** 2`, the operator `RE` is monadic and so is elaborated first.

a68g implements routines for complex arithmetic that circumvent unnecessary overflow when large real or imaginary values are used. Naive implementation of division, `sqrt` or `ABS` can overflow while the result is perfectly representable.

As described in the previous section, the indicant `COMPL` can be used wherever a mode is required.

From the section on field selection, it is clear that in the declarations

```
COMPL z = (2.0, 3.0);
COMPL w := z
```

the selection

```
re OF z
```

has mode `REAL` (and value `2.0`), while the selection

```
re OF w
```

has mode `REF REAL` (and its value is a name). However, the formula

```
RE w
```

still yields a value of mode `REAL` because `RE` is an operator whose single operand has mode `COMPL`. In the above expression, `w` will be dereferenced before `RE` is elaborated. Thus it is quite valid to write

```
im OF w := RE w
```

or

```
im OF w := re OF w
```

in which case the right-hand side of the assignation will be dereferenced before a copy is assigned.

5.23 Complex functions

Routines are the subject of a later chapter, however we have already met the mathematical functions for real values. Algol 68 Genie extends the Revised Report requirements for the standard-prelude by also defining mathematical functions for mode `COMPLEX`:

1. `complex sqrt(z)`
   The square root of `z`.
2. `complex exp(z)`
   Yields e^x.
3. `complex ln(z)`
   The natural logarithm of `z`.
4. `complex log(z)`
   The logarithm of `z` to base 10.
5. `complex sin(z)`
   The sine of `z`, where `z` is in radians.
6. `complex arcsin(z)`
   The inverse sine of `z`.
7. `complex cos(z)`
   The cosine of `z`, where `z` is in radians.
8. `complex arccos(z)`
   The inverse cosine of `z`.
9. `complex tan(z)`
   The tangent of `z`, where `z` is in radians.
10. `complex arctan(z)`
    The inverse tangent of `z`.

Note that a runtime error occurs if either argument or result are out of range. Multi-precision versions of these function are declared and are preceded by either `long` or `long long`, for instance `long complex sqrt` or `long long complex ln`.

5.24 Rows in structures

If rows are required in a structure, the structure declaration should only contain the required bounds if it is an actual-declarer. For example, we could declare

```
STRUCT ([] CHAR forename, surname, title)
   lecturer = ("Albert","Einstein","Dr")
```

where the mode on the left is a formal-declarer (remember that the mode on the left-hand side of an identity declaration is always a formal-declarer).

When declaring a name, because the mode preceding the name identifier is an actual-declarer (in an abbreviated declaration), the bounds of the required rows must be included. A suitable declaration for a name which could refer to `lecturer` would be

```
STRUCT ([7] CHAR forename, [6] CHAR surname, [3] CHAR title)
        new lecturer;
```

but we cannot assign `lecturer` to it. A better declaration would use `STRING`:

```
STRUCT (STRING forename, surname, title) person
```

in which case we could now write

```
person := lecturer
```

Using field selection, we can write

```
title OF person
```

which would have mode `REF STRING`. Thus, using field selection, we can assign to the individual fields of `person`:

```
surname OF person := "Schweitzer"
```

When slicing a field which is a row, it is necessary to remember that slicing binds more tightly than selecting (see chapter 9 for a detailed explanation). Thus the first character of the surname of `person` would be accessed by writing

```
(surname OF person)[1]
```

which would have mode `REF CHAR`. The parentheses ensure that the selection is elaborated before the slicing. Similarly, the first five characters of the forename of `person` would be accessed as

```
(forename OF person)[: 5]
```

with mode `REF [] CHAR`.

5.25 Multiple selection

In the last section, we considered rows in structures. What happens if we have a row each of whose elements is a structure? If we had declared

```
[10] COMPL z
```

then the selection `re OF z` would yield a name with mode `REF [] REAL` and bounds `[1 : 10]`. It is possible, because it is a name, to assign to it:

```
re OF z := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

This "extraction of a row of fields" from a row of a structured value is called multiple selection. A new descriptor is yielded in multiple selection, so one could write aliases for the fields:

```
[10] COMPL z;
[] REAL x = re OF z, y = im OF z
```

but note that both `x` and `y` are aliases, so any assignation to either one would also affect `z`. To avoid this side effect, the rows should be copied by a variable declaration:

```
[10] COMPL z;
[10] REAL x := re OF z, y = im OF z
```

after which both `x` and `y` contain the actual values of `z` but have no connection to `z` anymore.

Selecting the field of a sliced row of structured elements is straightforward. Since the row is sliced before the field is selected, no parentheses are necessary. Thus the real part of the third `COMPL` of `z` above is given by the expression

```
re OF z[3]
```

5.26 Transput of structures

Structures are transput field-by-field from left to right if the modes of every field can be transput. For example, the following program fragment will print a complex number:

```
print((complex sqrt (-1), new line))
```

For details of how this works, see the remarks on "straightening" in section 8.8.

5.27 A compact [] BOOL — mode BITS

The mode BITS is a compact representation for a [] BOOL. The BOOL values are thus represented as single bits. The advantage of using the mode BITS is efficiency: compact storage but also parallel operation on bits for a number of operators.

Algol 68 implements strong-widening from a BITS value to a [] BOOL value. Default formatting in transput of a value of mode BITS is as a row-of-bool.

Normally a BITS value would be stored in a machine word. The number of bits in one machine word is given by the environment enquiry (see section 11.4) bits width; on modern hardware this value on *a68g* will be either 32 or 64.

A BITS value can be denoted in four different ways using denotations written with radices of 2 (binary), 4, 8 (octal) or 16 (hexadecimal). Thus the declarations

```
BITS a = 2r 0000 0000 0000 0000 0000 0010 1110 1101,
     b = 4r 0000 0000 0002 3231,
     c = 8r 000 0000 1355,
     d = 16r 0000 02ed
```

are all equivalent because they all denote the same value. Note that the radix precedes the r and is written in decimal. Note also that the numbers can be written with spaces, or new lines, in the middle of the number. However, you cannot put a comment in the middle of the number. It is common practice to omit digits on the left of the denotation whose value is zero. Thus the declaration above could have been written

```
BITS a = 2r1011101101,
     b = 4r23231,
     c = 8r1355,
     d = 16r2ed
```

In standard Algol 68, a denotation for LONG BITS must be preceded by the keyword LONG and a denotation for LONG LONG BITS must be preceded by the keyword LONG LONG. As with integer - and real denotations, the *a68g* interpreter relaxes the use of prefixes when the context imposes a mode for a denotation, in which case a denotation of a lesser precision is automatically promoted to a denotation of the imposed mode.

It is important to note that in Algol 68 the most significant bit in a BITS value is bit 1 and the least significant bit is bit 32 (or bit 64). Nowadays this seems counter-intuitive, but when Algol 68 was designed this is the way mainframes as the IBM 360 or PDP 10 stored data - the sign bit was bit 0, the most significant bit was bit 1, and the least significant bit was either bit 31 (32-bit machines) or bit 35 (36 bit machines).

5.27.1 Monadic operators for BITS

There are many operators for BITS values. Firstly, the monadic operator BIN takes an INT operand and yields the equivalent value with mode BITS. The operator ABS converts a BITS value to its equivalent with mode INT. The NOT operator which you first met in chapter 3 (section 3.11) takes a BITS operand and yields a BITS value where every bit in the operand is reversed. Thus

```
   NOT 2r 1000 1110 0110 0101
          0010 1111 0010 1101
```

yields

```
       2r 0111 0001 1001 1010
          1101 0000 1101 0010
```

Note that spaces have been used to make these binary denotations more comprehensible. NOT is said to be a bit-wise operator because its action on each bit is independent of the value of other bits.

5.27.2 Dyadic operators for BITS

AND, OR (both of which you also met in chapter 3) and XOR both take two BITS operands and yield a BITS value. They are bit-wise operators and their actions are summarised as follows:

Bit 1	Bit 2	AND	OR	XOR
F	F	F	F	F
F	T	F	T	T
T	F	F	T	T
T	T	T	T	F

For OR, the yield of

```
   2r 100110 OR 2r 10101
```

is $2r110111$. The priority of AND and XOR is 3 and the priority of OR is 2.

The AND operator is particularly useful for extracting parts of a machine word. For example, suppose you have a BITS value where the least-significant 8 bits are equivalent to a character. You could write

```
CHAR c = REPR ABS (b AND 16rff)
```

Here, the operators REPR and ABS do not generate machine-code instructions, but merely satisfy the compiler that the modes are correct. This sort of formula is, in fact, very efficient in

Algol 68.

It is possible to extract a single bit from a word using the operator ELEM which has the header

```
(INT n, BITS t) BOOL
```

For example, given the declaration

```
BITS bi = 16r 394a 2716
```

then each hexadecimal digit represents 4 bits: the 3 occupies bit positions 1–4, the 9 occupies bit positions 5–8, the 4, bit positions 9–12, and so on. Suppose we want the third bit (the leftmost bit is bit-1). The following declaration is valid:

```
BOOL bit3 = 3 ELEM bi
```

Thus, if the third bit is a 1, the declaration will give the value TRUE for bit 3. In fact, 3 written in binary is 0011_2, so bit 3 is 1. Thus

```
2 ELEM bi
```

would yield FALSE. The priority of ELEM is 7.

Operators SET and CLEAR yield a BITS value with the bit indicated by the left operand set or cleared in the BITS value yielded by the right operand. For instance, bits width SET 2r100 yields 2r101. The priority of SET and CLEAR is 7.

The dyadic operators SHL and SHR shift a word to the left or to the right respectively by the number of bits specified by their right operand. When shifting left (SHL), bits shifted beyond the most significant part of the word are lost. New bits shifted in from the right are always zero. When shifting right (SHR), the reverse happens. Note that the number of bits shifted should be in the range $[-32, +32]$. For SHL, if the number of bits to be shifted is negative, the BITS value is shifted to the right and likewise for SHR. The header for SHL is

```
OP SHL = (BITS b, INT i) BITS
```

and correspondingly for SHR. The value b is the value to be shifted and the integer i is the number of bits to shift. UP and DOWN are synonyms for SHL and SHR respectively. The priorities of SHL and SHR are both 8.

As well as the operators = and /= (which have the usual meaning), the operators <= and >= are also defined for mode BITS. The formula

```
s >= t
```

yields TRUE only if for all bits in t that are 1, the corresponding bits in s are also 1. This is sometimes regarded as "s implies t". Contrariwise, the formula

```
s <= t
```

yields `TRUE` only if for all bits in `t` which are 0, the corresponding bits in `s` are also 0. Likewise, this is sometimes regarded as ”`NOT t` implies `s`”.

Sometimes you want to use bits values with more bits than offered by `BITS`. Algol 68 Genie supports modes `LONG BITS` and `LONG LONG BITS`. The range of `LONG LONG BITS` is default circa twice the length of `LONG BITS` but can be made arbitrary large through the intepreter-option `precision`, see section 10.7.4. Here are the respective bits widths for the three lengths available in *a68g*:

identifier	value
`bits width`	32
`long bits width`	116
`long long bits width`	232

5.28 A compact `[] CHAR` — mode `BYTES`

The mode `BYTES` is a compact representation of `[] CHAR`. It is a structure with an inaccessible field that stores a row of characters of fixed length in a compact way. A fixed-size object of mode `BYTES` may serve particular purposes (such as filenames on a particular file system) but the mode appears of limited use - it was useful in a time when memories were small. Unlike `BITS`, an object of mode `BYTES` does not have the advantage of parallel operation on its elements. Therefore, on modern hardware there is no reason to use an object of mode `BYTES` in stead of a `[] CHAR`. For reasons of compatibility with old programs, `a68` implements modes `BYTES` and `LONG BYTES`. Since these modes are of little practical use they are not extensively treated here and you are referred to chapter 11 that lists available operators and procedures for these modes.

Chapter 6

Routines

6.1 Routines

A routine is a set of encapsulated actions which can be elaborated in other parts of the program. A routine has a value with a well-defined mode. The value of a routine is expressed as a routine-text. Here is an example of a routine-text:

```
([] INT a) INT:
   BEGIN INT sum := 0;
         FOR i FROM LWB a TO UPB a
         DO sum +:= a[i] OD;
         sum
   END
```

In this example, the header of the routine is given in line 1:

`([] INT a) INT`

which could be read as ”with row-of-`INT`-parameter yielding `INT`”. The mode of the routine is given by the header, less any identifiers. So the mode of the above routine-text is

`PROC ([] INT) INT`

We say that the routine takes one parameter of mode `[] INT` and yields a value of mode `INT`. A routine-text is a unit. A routine-text less its header is sometimes called a ”body”. The body of the above routine-text is:

```
BEGIN INT sum := 0;
      FOR i FROM LWB a TO UPB a
      DO sum +:= a[i] OD;
      sum
END
```

The routine-text yields the sum of the individual elements of the parameter a. The body of a routine-text is a unit. In this case, the body is an closed-clause.

Since a routine-text is a value it can be associated with an identifier by means of an identity-relation:

```
PROC ([] INT) INT sum = ([] INT a) INT:
   BEGIN INT sum := 0;
         FOR i FROM LWB a TO UPB a
         DO sum +:= a[i] OD;
         sum
   END
```

Since in such a identity-relation the mode is stated on both sides of the equals-symbol, Algol 68 allows an abbreviation:

```
PROC sum = ([] INT a) INT:
   BEGIN INT sum := 0;
         FOR i FROM LWB a TO UPB a
         DO sum +:= a[i] OD;
         sum
   END
```

In the header of the above routine, a is declared as a formal parameter. The mode of a is [] INT. At the time the routine is declared, a does not identify a value and size of a row parameter is irrelevant. That is why it is called a formal-parameter. It is only when the routine is used (called) that a will identify a value.

Now, according to the header of the routine, the routine must yield a value of mode INT. The context of the body of a routine is strong. The last statement of the serial-clause will be coerced. In this case, we have a value of mode REF INT, sum, which, in a strong context, can be coerced to a value of mode INT by dereferencing.

The procedure sum declared above can be invoked by a call:

```
print ((sum ((1, 2, 3, 4, 5, 6, 7, 8, 9, 10)), new line))
```

will produce 55 on standard output.

Grouping your program into procedures helps to keep the logic simple at each level. Since procedure - and operator-declarations are declarations like other declarations, they can appear within other procedure - and operator-declarations. This is called nesting. Nesting procedures makes sense when the nested procedures are used only within the outer procedures.

6.2 Routine modes

In general, a routine may have any number of parameters, including none, as we will see. The mode of the parameters may be any mode except `VOID` and the value yielded may be any mode including `VOID`. The modes written for the parameters and the yield are always formal-declarers, so no bounds are specified if the modes of the parameters or yield involve rows.

Here is a possible header of a more complicated routine:

```
(REAL x, REAL y, REAL z) BOOL:
```

A minor abbreviation would be possible in this case since

```
REAL x, REAL y, REAL z
```

could be contracted to

```
REAL x, y, z
```

The mode of the routine above is

```
PROC (REAL, REAL, REAL) BOOL
```

Of course you can declare a structure with a procedure field:

```
STRUCT (PROC (REAL) REAL f, CHAR name, REAL arg) method =
   (sin, "sin", pi / 6)
```

Here is a name referring to such a structure:

```
STRUCT (PROC (REAL) REAL f, CHAR name, REAL arg) method2 = method
```

In the structure `method`, the procedure in the structure can be selected by

```
f OF method
```

which has the mode `PROC (REAL) REAL`. For reasons which will be clarified in chapter 9, if you want to call this procedure, you must enclose the selection in parentheses:

```
(f OF method)(pi / n OF method)
```

In the chapter on structures it was remarked that a mode-declaration cannot lead to infinitely large objects or endless coercion. It was demonstrated that `REF` shields a mode-indicant from its declaration through a `STRUCT`. `PROC` also shields a mode-indicant from its declaration through a `STRUCT`. A parameter-pack shields a mode-indicant from its declaration, also without being embedded in a structure. It is therefore possible to declare:

```
MODE NODE = (STRING info, PROC NODE process),
     P = PROC (P) P # rather academic, but ok #
```

6.3 Routines yielding VOID

A routine must yield a value of some mode, but it is possible to discard that value using the voiding coercion. The mode VOID has a single value denoted by EMPTY. VOID cannot be a formal-declarer (except as a component in a union, see chapter 7) or actual-declarer, so it is not allowed to write

```
VOID z = EMPTY
```

or

```
PROC (VOID) VOID # which is allowed in C #
```

though the effect of above (invalid) procedure can be obtained in Algol 68 by writing

```
PROC VOID # a parameter-less procedure yielding VOID #
```

In practice, because the context of the yield of a routine is strong, using EMPTY is usually unnecessary (but see section 7.2).

A FOR loop always yields EMPTY and a semicolon voidens the unit that terminates the loop. Declarations yield no value, not even EMPTY, as demonstrated by this *a68g* example:

```
$ a68g -e "PROC init = VOID: (INT i := 0)"
1     (PROC init = VOID: (INT i := 0))
      2                 1
a68g: syntax error: 1: clause does not yield a value.
a68g: syntax error: 2: clause does not yield a value.
```

6.4 Parameter-less procedures

Procedures can have no parameters at all; for example consider:

```
PROC report = VOID: print(("Now at point ", counter +:= 1));
```

A procedure can be invoked or called by writing its identifier as a statement. For example, the procedure above would be called by

```
INT counter := 0;
...
report;
...
report;
...
```

We have already seen the mechanism that calls a parameter-less procedure: it is by the deproceduring coercion. This coercion is available in every context (even soft).

6.5 Procedures with parameters

Parameters of procedures can have any mode, including procedures, except `VOID`. Unlike operators, procedures can have any number of parameters. The parameters are written as a parameter **list** consisting of parameters separated by commas.

A procedure with parameters is not implicitly called by deproceduring, but by writing a call. An example of a call is writing the identifier of a procedure followed by an argument-list. The argument list consists of one unit for every parameter, in a strong position, separated by commas. For example:

```
print(("The angle is ", atan2(x, y)));
```

Note that Algol 68's orthogonality allows you to write, instead of a procedure identifier, any primary (see chapter 9) that yields a procedure; for example:

```
r * (on x axis | cos | sin)(angle)
```

calls either `sin(angle)` or `cos(angle)` depending on the `BOOL` value of `on x axis`.

In standard Algol 68, a call of a routine must supply the same number of actual-parameters, and in the same order, as there are formal parameters in the procedure declaration. However, *a68g* offers an extension called partial parametrisation, which is treated in section 6.19.

6.6 Rows as parameters

Recall that in Algol 68, bounds are not part of a row mode. Since a formal-parameter which is a row has no bounds written in it, any row having that mode could be used as the actual-parameter. This means that if you need to know the bounds of the actual row, you will need to use operators interrogating bounds as `LWB` or `UPB`. An example is a routine-text which finds the smallest element in its row parameter `a`:

```
([] INT a) INT:
   (INT min := a[LWB a];
    FOR i FROM LWB a + 1 TO UPB a
    DO (a[i] < min | min := a[i])
    OD;
    min
   )
```

6.7 Names as parameters

When a parameter is a name, the body of the routine can have an assignation which makes the name refer to a new value. For example, here is a routine-text which assigns a value to its parameter:

```
(REF INT a) INT: a := 0
```

Note that the unit in this case is a single unit and so does not need to be enclosed.

If a flexible name is used as an actual-parameter, then of course the mode of the formal-parameter must include the mode constructor `FLEX`. For example,

```
(REF FLEX [] CHAR s) INT:
```

Of course, in this example, the mode of `s` could equivalently have been given as `REF STRING`.

You will have to be particularly careful when a formal-parameter of a procedure is a flexible name. A mechanism is in place in *a68g* to ensure that one cannot alter the bounds of a non-flexible row by aliasing it to a flexible row. This is particularly the case when passing names as parameters to procedures. For example, in a range that holds next declarations:

```
PROC x = (REF STRING s) VOID: ...,
PROC y = (REF [] CHAR c) VOID: ...;
```

these problems could occur:

```
x(LOC STRING);    # OK #
x(LOC [10] CHAR); # Not OK, suppose x changes the bounds of s! #
y(LOC STRING);    # OK #
y(LOC [10] CHAR); # OK #
```

a68g issues an error if it encounters a construct that might alter the bounds of a non-flexible row.

6.8 Routines yielding names

Since the yield of a routine can be a value of any mode, a routine can yield a name, but there is a restriction: the name yielded must have a scope larger than the body of the routine. This means that any names declared to be *local*, cannot be yielded by the routine. The reason for this is simple: when the routine terminates, it discards the local objects it generated, so any reference to those local object would point at something that no longer exists.

We have seen that a new name can be generated using the generator `LOC`. This is a routine which should yield a name generated within its body:

```
(INT a) REF INT: LOC INT := a # wrong #
```

This routine is wrong because the scope of the name generated by `LOC INT` is limited to the body of the routine. a68g provides both compile-time and run-time scope checking, and will flag this error.

There is a way of yielding a name declared in a routine. This is achieved using a global generator:

```
(INT a) REF INT: HEAP INT := a
```

6.9 Procedures as parameters

Parameters can be of any mode but `VOID`, so it is possible to pass procedures as parameters. Here is a procedure which takes a procedure as a parameter:

```
PROC series sum = (INT n, PROC (INT) REAL func) REAL:
   (REAL s := 0;
    FOR i TO n
    DO s +:= func(i)
    OD;
    s
   )
```

Note that the mode of the procedure parameter is a formal mode so no identifier is required for its `INT` parameter in the header of the procedure `sum`. In the loop-clause, the procedure is called with an actual-parameter.

When a parameter must be a procedure, any unit yielding a routine-text can be supplied. For instance, a predeclared procedure identifier can be supplied, as in

```
PROC rec = (INT a) REAL: 1 / a;
```

```
series sum(1000, rec)
```

or a routine-text:

```
series sum(1000, (INT a) REAL: 1 / a)
```

In this case, the routine text has the mode `PROC (INT) REAL`, so it can be used in the call of `series sum`. Note also that, because the routine text is an actual-parameter, its header includes the identifier `a`. In fact, routine-texts can be used wherever a procedure is required, as long as the routine-text has the required mode. The routine-text given in the call is on the right-hand side of the implied identity declaration of the elaboration of the parameter.

6.10 Routines and scope

We have seen that this routine is not correct:

```
(INT a) REF INT: LOC INT := a
```

This routine is wrong because the scope of the name generated by `LOC INT` is limited to the body of the routine. This is a scope error. The difference between range and scope is that identifiers have range, but values have scope. Denotations of primitive modes have global scope. A routine-text also is a value, but a potential scope problem arises when you write a routine that yields another routine. There is a danger that a routine gets exported to ranges where the symbols that the exported routine applies, no longer exist. For example, next declaration is wrong:

```
MODE FUN = PROC (REAL) REAL;
MODE OPERATOR = PROC (FUN) FUN;
OPERATOR deriv = (FUN f) FUN: (REAL x) REAL: f(x) - f(x - 1);
```

the danger is that `f` may no longer exist when the routine yielded by `deriv` gets called, and *a68g* will warn you for the potential danger:

```
$ a68g examples/scope.a68
3     OPERATOR deriv = (FUN f) FUN: (REAL x) REAL: f(x) - f(x - 1);
                                    1
a68g: warning: 1: PROC (REAL) REAL value from  routine-text could
be exported out of its scope (detected in particular-program).
```

For this reason you cannot export a routine out of the ranges that hold all declarations (identifiers, operators, modes) that the exported routine applies. A routine is said to have thus a necessary environment outside of which the routine is meaningless.

6.11 Miscellaneous features of procedures

Since a procedure is a value, it is possible to declare values whose modes include a procedure mode. For example, here is a row of procedures:

```
[] PROC (REAL) REAL pr = (sin, cos, tan)
```

and here is a possible call:

```
pr[2](pi / 2)
```

We could also declare a procedure which could be called with the expression

```
pr(2)[2]
```

but this is left as an exercise.

Similarly, names of procedures can be declared and can be quite useful. Instead of declaring

```
PROC pc = (INT i) PROC (REAL) REAL: pr[i]
```

using `pr` declared above, with a possible call of `pc(2)` we could write

```
PROC (REAL) REAL pn := pr[i]
```

and then use `pn` instead of `pc`. The advantage of this would be that `pr` would be subscripted only once instead of whenever `pc` is elaborated. Furthermore, another procedure could be assigned to `pn` and the procedure it refers to again called. Using `pn` would usually involve dereferencing.

There are times when `SKIP` is useful in a procedure declaration:

```
PROC sqrtf = (REAL x) REAL:
   IF x >= 0
   THEN sqrt(x)
   ELSE print("Negative parameter");
        SKIP
   FI
```

The yield of the procedure is `REAL`, so each part of the conditional clause must yield a value of mode `REAL`. `SKIP` will yield an undefined value of the mode required by the context. In this case, `SKIP` yields some value of mode `REAL`.

6.12 Operators

An operator is called monadic when it takes one operand, or dyadic when it takes two operands in which case it also needs a priority. Monadic-operators have priority over any dyadic-operator.

Operators are declared by means of an identity declaration:

```
OP (INT) INT ABS = (INT a) INT: (a >= 0 | a | -a);
```

There are several points to note:

1. The mode of the operator is `PROC (INT) INT`. That is, it takes a single operand of mode `INT` and yields a value of mode `INT`.
2. The right-hand side of the identity declaration is a routine text. Since the routine-text forces a mode for the operator, an abbreviated declaration can be used for operators:

   ```
   OP ABS = (INT a) INT: (a >= 0 | a | -a);
   ```

6.13 Operator-symbols

An operator-symbol can be a tag as `ABS` or `REPR`. *a68g* accepts a tag that starts with an upper-case letter, optionally followed by upper-case letters or underscores. Since spaces are not allowed in an upper-case tag to avoid ambiguity, underscores can be used to improve legibility. The use of underscores is not allowed in standard Algol 68.

Many of the operators described up to now are not words but composed of mathematical symbols as `+` or `**`. Since spaces have no meaning in between operator-symbols composed of mathematical symbols, rules apply to guarantee that any sequence of symbols has one unique meaning.

To avoid ambiguity in parsing operator-symbol sequences, operator characters are divided into monads and nomads, the latter group being inherently dyadic. Next rules force that for instance `1++-1` can only mean `(1) + (+(-2))`, and `1+>2` can only mean `(1) +> (2)` and not `(1) + (>2)`.

1. monads are `+`, `-`, `!`, `?`, `%`, `^`, `&`, `~`.
2. (inherently dyadic) nomads are `<`, `>`, `/`, `=`, `*`.
3. a monadic-operator-symbol is a monad, optionally followed by a nomad.
4. a dyadic-operator-symbol is either a monad or a nomad, optionally followed by a nomad.

5. an operator-symbol consisting of monads or nomads may be followed by either `:=` or `=:`.

Note that Algol 68 forbids operator-symbols that start with a double monad, such as `++`, `--` or `&&`, although dyadic-operator-symbols starting with a double nomad (`**`, `>>`, et cetera) are allowed.

Some declarations of operators using above rules are:

```
OP ^^ = (INT a, b) INT: 2 ^ (a ^ b);
OP % = (BOOL b) BOOL: b AND read int > 0;
OP -:= = (REF CHAR c, d) INT: c := REPR (ABS c - ABS d)
```

6.14 Dyadic-operators

The difference between monadic and dyadic operators is that the latter have a priority and take two operands in stead of one. Therefore the routine-text used for a dyadic-operator has two formal-parameters. The priority of a dyadic-operator is declared using the keyword `PRIO`:

```
PRIO ** = 9, +=: = 1
```

The declaration of the priority of the operator uses a digit in the range 1 to 9 on the right-hand side.

Consecutive priority declarations do not need to repeat the keyword `PRIO`, but can be abbreviated in the usual way. The priority declaration relates to the operator symbol, not to the modes of operands or result. Hence the same operator cannot have two different priorities in the same range, but there is no reason why an operator cannot have different priorities in different ranges.

If an existing operator-symbol is used in a new declaration, the priority of the new operator must be the same as the old if it is in the same range, so the priority declaration should be omitted.

6.15 Identification of operators

An obvious consequence of being able to declare operators with common operator-symbols as `+` or `ELEM` is that there can be more than one declaration of the same operator-symbol. Think for instance of a `+` for integers, for reals, for complex values, et cetera. This is called ”operator overloading”.

How does the interpreter identify the operator to use when various definitions exist? As in identifying any declaration, the interpreter will first search the range in which it finds the application of that declaration (in this case, an operator-symbol). If no such declaration is in that range, it will search the embedding range, and so on outwards until finally the standard-prelude is searched.

Suppose the interpreter finds a declaration with matching operator-symbol and matching number of operands. How is it identified as the operator to be actually used? To answer that question we must know the coercions that apply to an operand. The syntactic position of an operand is firm. In a firm context we can have (repeated) dereferencing and deproceduring, followed by uniting. Dereferencing is already discussed, uniting will be discussed in chapter 9, and this is a good place to explain deproceduring.

Deproceduring is a forced call of a parameter-less procedure. You have seen various applications of deproceduring already, for instance in

```
INT k := read int
```

where we see that the `PROC INT` must somehow be coerced to an `INT` that is expected in the strong context that is the source of the destination. The solution is trivial — call the `PROC INT` procedure to produce a value of mode `INT`.

In determining which operator to use, the interpreter finds, in the smallest range enclosing the formula, that operator-declaration with correct number of operands, whose modes can be obtained from the operands in question using coercions allowed in a firm context.

This way of identifying operators has an important consequence: if in a same range there could exist two operator declarations taking the same number of operands of modes that can be coerced to each other by firm coercions (in which case we say that these modes are firmly related), the interpreter would have no way to choose between the two. This condition is therefore forbidden, it is not valid Algol 68. You cannot in the same range declare multiple operators whose operands are firmly related. For instance you could not declare in one range

```
OP ? = (INT k) INT: k OVER 2;
...
OP ? = (REF INT k) BOOL: k IS NIL
```

since this would lead to ambiguous identification:

```
? 9
```

may identify the first declaration but

```
INT k := read int;
? k
```

is ambiguous.

The identification of dyadic operators proceeds exactly as for monadic-operators except that the most recently declared priority in the same range is used to determine the order of elaboration of operators in a formula.

6.16 Standard operators and procedures

The standard-prelude contains the declarations of numerous operators procedures, most of them concerned with transput (see chapter nine). A number of procedures, all having the mode

```
PROC (REAL) REAL
```

are declared in the standard-prelude and yield the values of common mathematical functions. These are `sqrt`, `exp`, `ln`, `cos`, `sin`, `tan`, `arctan`, `arcsin` and `arccos` et cetera. Naturally, you must be careful to ensure that the actual-parameter for `sqrt` is non-negative, and that the actual-parameter for `ln` is greater than zero. The procedures `cos`, `sin` and `tan` expect their `REAL` parameter to be in radians.

New procedures using these predeclared procedures can be declared:

```
PROC sec = (REAL x) REAL: 1 / sin(x);
OP (REAL) REAL COS = cos
```

6.17 Recursion

One of the elegant properties of procedures and operators is that they can call themselves. This is known as recursion. For example, here is a paradigm of recursion — the Ackermann function which is a well-known recursive definition in mathematics

```
PROC ack = (INT m, n) INT:
  IF m = 0
  THEN n + 1
  ELSE (n = 0 | ack(m - 1, 1) | ack(m - 1, ack(m, n - 1)))
  FI;
```

Though this is a function only of interest in number theory, it does find practical application in testing compilers for their ability to perform deep recursion well.

Next example is another illustration of a recursive procedure that calculates the number of ways to split an amount of money in coins of 2 euro, 1 euro, 50 ct, 20 ct, 10 ct and 5 ct. It

Primitive recursive functions (computable functions) are defined using primitive recursion and composition as central operations. Many of the functions normally studied in number theory, and approximations to real-valued functions, are primitive recursive such as addition, division, factorial, exponential, finding the n^{th} prime, et cetera. It is difficult to devise a function that is not primitive recursive, although some are known, for example the Ackermann function.

applies back-tracking, which is constructing a tree of all possible combinations and cutting off impossible branches as early as possible:

```
PROC count = (INT rest, max) INT:
   IF rest = 0
   THEN 1 # Just right, valid combination found #
   ELIF rest < 0
   THEN 0 # Invalid combination, subtracted too much #
   ELSE [] INT values = (5, 10, 20, 50, 100, 200);
        INT combinations := 0;
        FOR i TO UPB values
        WHILE values[i] <= max
        DO combinations +:= count (rest - values[i], values[i])
        OD;
        combinations
   FI;

INT amount = 500 # cts #;
print (count (amount, amount))
```

6.17.1 Fast Fourier Transform

Some more points of interest to recursion will be illustrated by a non-trivial example which is a workhorse from numerical analysis: the Fast Fourier Transform or abbreviated, FFT. It can be easily proven that the $k-th$ element of a discrete Fourier transform

$$F_k = \sum_{j=0}^{N-1} e^{2\pi ik/N} f_j$$

can be rewritten as the sum of two sub-transforms

$$F_k = F_{k,even} + e^{2\pi ik/N} F_{k,odd}$$

where $F_{k,even}$ is the transform of the even-numbered elements and $F_{k,odd}$ is the transform of the odd-numbered elements. This splitting of the transform can be applied recursively, meaning that transform $F_{k,even}$ and $F_{k,odd}$ can be obtained by calculating their sub-transforms et cetera, up until the point where the transform becomes trivial when there is just one element:

$$F_1 = f_1; N = 1$$

which is an identity. The advantage of binary splitting is that the complexity of a Fourier transform can be greatly reduced. A Fourier transform of length N requires N^2 operations. The binary splitting reduces that to $N \log_2 N$ operations. That is an immense improvement for large N, and many practical datasets are large.

Note that this simple form of FFT involves a binary split in every sub-transform, and that therefore the number of elements in F must be a power of 2. Algorithms that lift this limitation of FFT are out of the scope of this text.

Here is a naive but instructive implementation of the FFT:

```
OP FFT = (REF [] COMPLEX f) VOID:
   IF # Unnormalised Fast Fourier Transform in recursive form:
        ELEMS f must be a power of 2 and LWB f must be zero. #
      INT length = ELEMS f;
      length = 1
   THEN # When the length is 1, the transform is an identity #
        SKIP
   ELSE INT middle = length % 2;
        # Calculate sub-transforms recursively #
        [0 .. middle - 1] COMPLEX f even, f odd;
        FOR i FROM 0 TO middle - 1
        DO f even[i] := f[2 * i];
           f odd[i] := f[2 * i + 1]
        OD;
        (FFT f even, FFT f odd);
        # Calculate transform at this level #
        FOR k FROM 0 TO middle - 1
        DO REAL phi = 2 * pi * k / length;
           COMPLEX w = cos(phi) I sin(phi);
           f[k] := f even[k] + w * f odd[k];
           f[k + middle] := f even[k] - w * f odd[k]
        OD
   FI;
```

Note that the routine does assignations to elements of its arguments that must therefore be of mode `REF [] COMPLEX`. When the parameter `f` holds a single element, the transform is an identity and the routine just returns. When the length of the parameter row is a power of two, the transform is is calculated by making a binary split into even and odd elements, which are than transformed recursively.

Obviously, when the transformation is in progress, various calls to `FFT` are made, and every instant of `FFT` — which is called an incarnation of the routine — must wait for the sub-transform to finish, before it can continue. Hence various incarnations of a routine can exist at the same time; only the deepest (youngest) one being active, and the older ones awaiting completion of the younger ones.

Algol 68 elaborates every new incarnation of a routine as if the called routine's body were *textually* and if applicable *recursively* inserted into the source code. Algol 68 is defined such that no incarnation can have access to locally defined tags in another incarnation. If such access is needed, such tag must be passed along as a parameter when calling a deeper incarnation. Hence every declaration in a routine is private to an incarnation; for instance, every incarnation of FFT has its own `f even` and `f odd` but also `length` et cetera.

6.17.2 Space-filling curves

A beautiful demonstration of the elegance of recursion is the drawing of so-called space-filling curves.

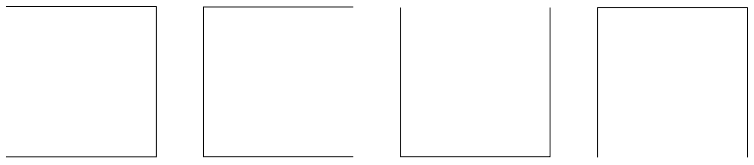
The mathematician Giuseppe Peano studied among many other subjects the projection of a square onto a line [1]. He considered a set of curves, constructed from straight line segments. The $n - th$ member of the set is a continuous curve that passes all points in the square at a distance of at most 2^{-n}, a so-called Peano curve. There are various such sets and here we will demonstrate an algorithm due to A. van Wijngaarden that draws a particular solution that is known as a Hilbert-curve. Drawing the Hilbert-curve actually was an exercise in Algol 68 programming classes.

Let us first define:

```
REAL d = 2.0 ^ (- n); # grid resolution #
MODE POINT = STRUCT (REAL x, y);
POINT origin := (d / 2, d / 2);
```

a68g gives you some standard routines that allow you to plot in a plane, see section 11.10. Here we will skip the details of plotting, but the figures that follow have been plotted with *a68g*. For the moment it suffices to comment that we can move the pen to an initial position `origin` and that we have a procedure `line to` that will draw a line to a specified point, and than set `origin` to this point.

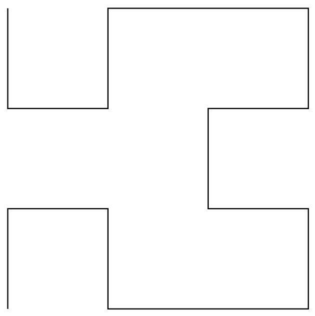
The $0 - th$ member of the curve is trivial, it is a point in the centre of the plane. The first member $n = 1$ starts at the bottom of the figure and end at its top, we call this orientation ”up”. By changing the orientation we also get ”down”, ”left” and ”right”: The divide-and-conquer


(a) up (b) down (c) left (d) right

strategy in this problem is to subdivide the square into four equal squares, and combine the four orientations which are chosen in the particular way depicted here to allow drawing a

[1]G. Peano. Sur une courbe, qui remplit toute une aire plaine. Math. Annln. (36) 157-160 [1890]

continuous curve by recursively applying our divide-and-conquer strategy. Before we write a routine to draw the orientation "up" we construct the second member $n = 2$ in a convenient way:



The figures for $n = 1$ and $n = 2$ are drawn using next procedures:

```
PROC draw up = (INT n) VOID:
     IF n > 0
     THEN draw right (n - 1);
          line to ((x OF origin + d, y OF origin));
          draw up (n - 1);
          line to ((x OF origin, y OF origin + d));
          draw up (n - 1);
          line to ((x OF origin - d, y OF origin));
          draw left (n - 1)
     FI;

PROC draw down = (INT n) VOID:
     IF n > 0
     THEN draw left (n - 1);
          line to ((x OF origin - d, y OF origin));
          draw down (n - 1);
          line to ((x OF origin, y OF origin - d));
          draw down (n - 1);
          line to ((x OF origin + d, y OF origin));
          draw right (n - 1)
     FI;

PROC draw left = (INT n) VOID:
     IF n > 0
     THEN draw down (n - 1);
          line to ((x OF origin, y OF origin - d));
          draw left (n - 1);
          line to ((x OF origin - d, y OF origin));
          draw left (n - 1);
          line to ((x OF origin, y OF origin + d));
          draw up (n - 1)
     FI;

PROC draw right = (INT n) VOID:
     IF n > 0
```

```
    THEN draw up (n - 1);
         line to ((x OF origin, y OF origin + d));
         draw right (n - 1);
         line to ((x OF origin + d, y OF origin));
         draw right (n - 1);
         line to ((x OF origin, y OF origin - d));
         draw down (n - 1)
    FI;
```

For example, the closing illustration to this chapter is the $n = 6$ member of the family drawn with the routines given here by calling `go right(6)`: it is left as an exercise to show that the Euclidean length of the Hilbert curve, member n, reads

$$2^n - \frac{1}{2^n}$$

that is, grows exponentially.

6.17.3 Recursion versus iteration

In the factorial example an iterative routine is likely to be slightly faster in practice than the recursive one. This result is typical, because iterative functions do not have the "multiple call overhead" of recursive functions, and that overhead is relatively high in many languages. Note that an even faster implementation for the factorial function on small integers is to use a lookup table.

There are other types of problems whose solutions are inherently recursive, because they need to keep track of prior state. One example is tree traversal; others include the Ackermann function and divide-and-conquer algorithms such as Quicksort. All of these algorithms can be implemented iteratively with the help of a stack, but the need for the stack arguably nullifies the advantages of the iterative solution.

Another possible reason for choosing an iterative rather than a recursive algorithm is that in today's programming languages, the stack space available to a thread is often much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms.

6.18 Recursion and data structures

In mathematics and computer science, graph theory is the study of mathematical structures used to model pairwise relations between objects from a certain collection. A "graph" in this context refers to a collection of vertices or 'nodes' and a collection of edges that connect pairs of vertices. This is not the place to discuss topics in graph theory but it sets the background for this section. We will demonstrate how to handle data structures where nodes of a same mode refer to each other.

6.18.1 Linear lists

A linked list is a fundamental data structure, and can be used to implement other data structures. A linked list is a self-referring data structure because each node contains a pointer to another node of the same type. Practical examples are process queues or lists of free blocks in heap management.

Linked lists permit insertion and removal of nodes at any point in the list, but do not allow random access. A linear list is a finite set in which the elements have the relation *is followed by*. We call it linear because each element has one successor. We could of course try to represent it by a row of elements

```
MODE LIST = FLEX [1 : 0] ELEMENT
```

The principal benefit of a linked list over a row is that the order of the linked items may be different from the order that the data items are stored, allowing the list of items to be traversed in a different order. Also, inserting an element in a row is an expensive operation.

An elegant representation is a self-referring data structure:

```
# Data structure and primitive operations: #

MODE NODE = STRUCT (ELEMENT info, LIST successor),
     LIST = REF NODE;

LIST empty list = NIL;

OP ELEM = (LIST list) REF ELEMENT: info OF list,
OP REST = (LIST list) REF LIST: successor OF list

OP EMPTY = (LIST list) BOOL: list IS empty list

OP ELEM = (INT n, LIST list) REF ELEMENT:
   IF list IS empty list OR n < 1
   THEN empty list # thus we mark an error condition #
   ELSE (n = 1 | ELEM list | (n - 1) ELEM list)
   FI;

PRIO ELEM = 9

# A routine to have a denotation for lists: #

PROC make list = ([] ELEMENT row) LIST:
   IF ELEMS row = 0
   THEN empty list
   ELSE HEAP NODE := (row[i], make list(row[LWB row + 1 : UPB row]))
   FI

# Note that REST yields a REF REF NODE, a pointer-variable #

# Operators to add an element to a list: #

# Add at the head #
OP + = (ELEMENT elem, LIST list) VOID:
   HEAP LIST := (elem, list)

# Add at the tail #
OP +:= = (REF LIST list, ELEMENT elem) VOID:
   IF EMPTY list
   THEN list := HEAP LIST := (elem, empty list)
   ELSE REST list +:= elem
   FI
```

Several different types of linked list exist:

1. singly-linked lists as discussed above, where each node points at its successor,
2. doubly-linked lists in which each node points to both predecessor and successor,
3. circularly-linked lists in which the last node points back at the first one which is especially useful when implementing buffers.

Linked lists sometimes have a special dummy or sentinel node at the beginning or at the end of the list, which is not used to store data. Its purpose is to simplify some operations, by ensuring that every node always has a previous - and next node, and that every list (even an empty one) always has a first and last node. Lisp has such a design - the special value nil is used to mark the end of a proper singly-linked list; a list does not have to end in nil, but a list that did not would be termed improper.

6.18.2 Trees

A binary tree is a data structure in which each node refers to two other nodes. Binary trees are commonly used to implement binary search trees and binary heaps. Next is the paradigm quick-sort routine in Algol 68:

```
MODE NODE = STRUCT (INT k, TREE smaller, larger),
     TREE = REF NODE;
TREE empty tree = NIL;

PROC sort = (REF TREE root, INT k) VOID:
   IF root IS empty tree
   THEN root := HEAP NODE := (k, empty tree, empty tree)
   ELSE sort((k < k OF root | smaller OF root | larger OF root), k)
   FI;

PROC write = (TREE root) VOID:
   IF root ISNT NIL
   THEN write(smaller OF root);
        print((whole (k OF root, 0), " "));
        write(larger OF root)
   FI;

TREE root := empty tree;
WHILE INT n = read int;
      n > 0
DO sort(root, n)
OD;
write(root)
```

Compare the version above with next iterative version, which is the so-called ”triple-REF trick”:

```
MODE NODE = STRUCT (INT v, REF NODE less, more);
REF NODE empty tree = NIL;

PROC sort = (INT v) VOID:
   BEGIN
      REF REF REF NODE place = LOC REF REF NODE := root;
      WHILE place ISNT empty tree
      DO place := (v < v OF place | less OF place | more OF place)
      OD;
      REF REF NODE (place) := HEAP NODE := (v, empty tree, empty tree)
   END;

PROC write = (TREE root) VOID:
   IF root ISNT NIL
   THEN write(smaller OF root);
        print((whole (k OF root, 0), " "));
        write(larger OF root)
   FI;

REF REF NODE root = LOC REF NODE := empty tree;
WHILE INT n = read int;
      n > 0
DO sort(n)
OD;
write(root)
```

This is an example of an application of a `REF REF REF` object, that is, a pointer-to-pointer variable.

Next example demonstrates buidling a decision tree in Algol 68. It applies `UNION`s which are the subject of chapter 7. Suffice here to say that a union can hold in one object values of different modes, but only one value at a time, and the mode of the currently contained value can be interrogated using a special case-clause. Next program builds a library of objects you type (with a question to distinguish the object) while it tries to guess an object you have in mind (This was a popular type of program among students. They wrote programs like this for *ALGOL68C* in the 1980's with filenames as `ANIMALS ALGOL68`):

```
# Q&A game #

ENTRY library := get reply("give an initial answer");

DO guess object(library);
   UNTIL ~ ask("again")
OD;

# Data structure #

MODE ENTRY = UNION (STRING, FORK),
     FORK = STRUCT (STRING text, REF ENTRY has, hasnt);

OP TEXT  = (FORK d) STRING: text OF d,
   HAS   = (FORK d) REF ENTRY: has OF d,
   HASNT = (FORK d) REF ENTRY: hasnt OF d;

PROC new fork = (STRING text, ENTRY has, hasnt) FORK:
     (HEAP STRING := text, HEAP ENTRY := has, HEAP ENTRY := hasnt);

# Guessing and extending library #

PROC guess object = (REF ENTRY sub lib) VOID:
     # How to guess an object #
     CASE sub lib
     IN (STRING s): (ask(s) | ~ | sub lib := learn(s)),
        (FORK d):   guess object((ask(TEXT d) | HAS d | HASNT d))
     ESAC;

PROC learn = (STRING guess) ENTRY:
     # Introduce new entry in library #
     IF STRING answer = get reply("what is the answer"),
               question = get reply("give a question to distinguish "
                                    + answer);
        ask("does " + question + " apply to " + answer)
     THEN new fork(question, answer, guess)
     ELSE new fork(question, guess, answer)
     FI;

# Interaction #

PROC get reply = (STRING prompt) STRING:
     BEGIN STRING s;
           printf(($gl$, prompt));
```

```
            readf(($gl$, s));
            s
     END;

PROC ask = (STRING question) BOOL:
     IF STRING s = get reply(question);
        UPB s > 0
     THEN s[1] = "y" ORF s[1] = "Y"
     ELSE ask (question)
     FI;

SKIP
```

6.19 Partial parametrisation and currying

a68g implements C.H. Lindsey's proposal for partial parametrisation [Lindsey AB39.3.1] giving the imperative language Algol 68 a functional sub language [Koster 1996]. A formal description of this proposal can be found in the appendix to the Algol 68 Revised Report that is distributed with Algol 68 Genie.

With *a68g* a call does not require that all arguments be specified. In case not all of the actual-parameters are specified, a call yields a routine (with identical body but with already specified arguments stored) that requires the unspecified actual-parameters. When specification of all required actual-parameters is complete (complete closure) a procedure will actually be evaluated to yield its result.

⚠ Partial parametrisation is closely related to currying, a transformation named after Haskell Curry. Currying is transforming a function
$f : (x \times y) \rightarrow z$ into $f : (x) \rightarrow (y \rightarrow z)$.
This transformation is for instance used in λ-calculus.

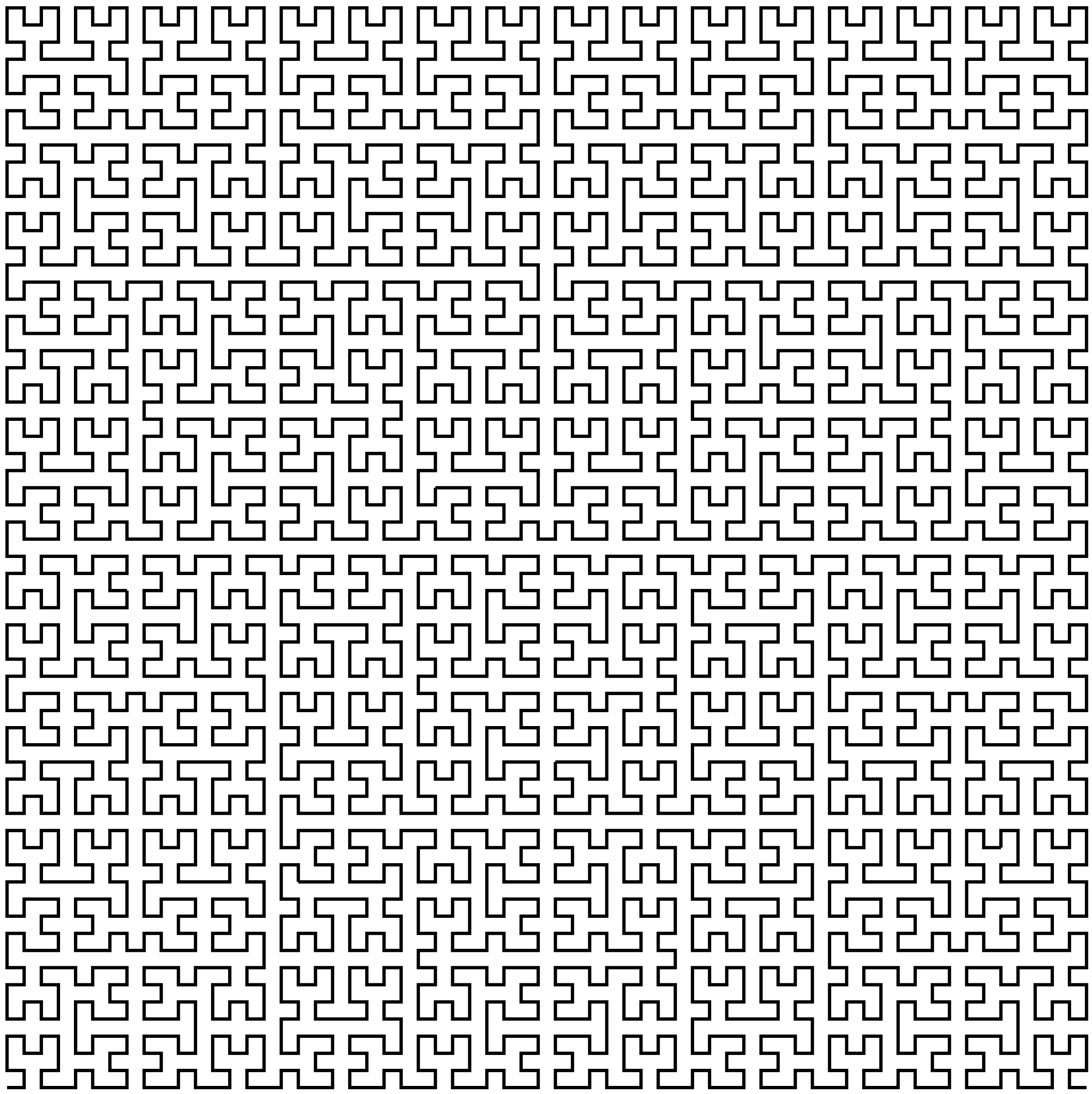
a68g does not save copies of the stack upon partial parametrisation, as happens for example in Lisp; the yield of a partially parametrised call in an environ *E*, cannot be newer in scope than *E*. Therefore stored actual-parameters cannot refer to objects in stack frames that no longer exist, and dynamic scope checking extends to stored actual-parameters.

A routine may be parametrised in several stages. Upon each stage the yields of the new actual-parameters are stored inside the routine's locale and the scope of the routine becomes the newest of its original scope and the scopes of those yields.

Here is an example of partial parametrisation:

```
# Raising a routine to a power #

MODE FUN = PROC (REAL) REAL;
```

```
PROC pow = (FUN f, INT n, REAL x) REAL: f(x) ** n;
OP ** = (FUN f, INT n) FUN: pow (f, n, );

# Example: sin (3 x) = 3 sin (x) - 4 sinˆ3 (x)
  (follows from DeMoivre’s theorem) #

REAL x = read real;
print ((new line, sin (3 * x),  3 *  sin (x) - 4 * (sin ** 3) (x)))
```



The $n = 6$ Hilbert-curve, a continuous space-filling curve drawn with a68g using a recursive algorithm of A. van Wijngaarden, explained in this chapter.

Figure © Marcel van der Veer 2008.

Chapter 7

United modes

7.1 United mode declarations

`UNION` is used to create a united mode. For example `UNION (INT, STRING)` can either hold either an `INT` or a `STRING`. Experience shows that united-modes are not used very often in actual programs. One way of using unions is to save a little memory space when two values can be mapped onto each other, but this argument hardly holds anymore considering the specifications of modern hardware. A good way to approach unions is

1. to use them as an abstraction tool, when you want to explicitly express that an object can be of one of several modes. Transput is a good example of this, for instance

   ```
   MODE NUMBER = UNION (INT, REAL,
                        LONG INT, LONG REAL,
                        LONG LONG INT, LONG LONG REAL);
   ```

   or think of a Lisp interpreter in Algol 68 where you could declare for example this:

   ```
   MODE VALUE = UNION (ATOM, LIST),
        ATOM  = STRING,
        LIST  = REF NODE,
        NODE  = STRUCT (VALUE car, cdr);
   ```

   Also here, the union serves to provide abstraction.

2. to otherwise use a structure with a field for every mode you need.

The first thing to notice is that, unlike structures, there are no field selectors. This is because a united mode does not consist of constituent parts. The second thing to notice is that `UNION (INT, STRING)` could well have been written

```
UNION (STRING, INT)
```

The order of the modes in the union is irrelevant, they are associative. A constituent mode can be repeated, as in

```
UNION (STRING, INT, STRING)
```

This is equivalent to the previous declarations, identical modes are ”absorbed”.

In computer science, a union is a data structure that stores one of several types of data at a single location. Algol 68 implements tagged unions. A tagged union, also called a variant, variant record, discriminated union, or disjoint union, is a data structure used to hold a value that could take on several different, but fixed types. Only one of the types can be in use at any one time, and a tag field explicitly indicates which one is in use.

Like structured modes, united modes are often declared with the mode declaration. Here is a suitable declaration of a united mode containing the constituent modes `STRING` and `INT`:

```
MODE CARDINAL = UNION (STRING, INT)
```

We could create another mode `NUMERAL` in two ways:

```
MODE NUMERAL = UNION (CARDINAL, REAL)
```

or

```
MODE NUMERAL = UNION(STRING, INT, REAL)
```

Using the above declaration for `CARDINAL`, we could declare `u` by

```
CARDINAL u = (roman mood | "MMVIII" | 2008)
```

In this identity declaration, the mode yielded by the right-hand side is either `INT` or `STRING`, but the mode required is `UNION(STRING,INT)`. The value on the right-hand side is coerced to the required mode by uniting.

The united mode `CARDINAL` is a mode whose values either have mode `INT` or mode `STRING`. Any value of a united mode actually has a mode which is one of the constituent modes of the union. So there are no new values for a united mode. Because a united mode does not introduce new values, there are no denotations for united modes. Identifier `u` as declared above identifies a value which is either an `INT` or a `STRING`. We will see later how to determine the mode of the value currently contained in a united object.

Almost any mode can be a constituent of a united mode. For example, here is a united mode containing a procedure mode and `VOID`:

```
MODE PROID = UNION (PROC (REAL) REAL, VOID)
```

and a declaration applying it:

```
PROID pd = EMPTY
```

The only limitation on united modes is that none of the constituent modes may be firmly related (see the section 6.15) and a united mode cannot appear in its own declaration. The following declaration is wrong because a value of one of the constituent modes can be deprocedured in a firm context to yield a value of the united mode:

```
MODE WRONG = UNION (PROC WRONG, INT)
```

Names for values with united modes are declared in exactly the same way as before. Here is a declaration for such a name using a local generator:

```
REF UNION (BOOL, INT) un = LOC UNION (BOOL, INT);
```

The abbreviated declaration reads

```
UNION (BOOL, INT) un;
```

Likewise, we could declare a name for the mode CARDINAL:

```
CARDINAL sn;
```

In other words, objects of united modes can be declared in the same way as other objects.

7.2 United modes in procedures

We can now partly address the problem of the parameters for print and read. It should be possible to construct a united mode which will accept all the modes accepted by those two procedures. In fact, the united modes used are almost the same as the two following declarations:

```
MODE SIMPLOUT =  UNION (
    INT, REAL, BOOL, CHAR,
    [] INT, [] REAL, [] BOOL, [] CHAR,
    [, ] INT, [, ] REAL, [, ] BOOL, [, ] CHAR,
    ...
   ),

PROC print = ([] SIMPLOUT) VOID;

MODE SIMPLIN = UNION (
    REF INT, REF REAL, REF BOOL, REF CHAR,
    REF [] INT, REF [] REAL, REF [] BOOL, REF [] CHAR,
    REF [, ] INT, REF [, ] REAL, REF [, ] BOOL, REF [, ] CHAR,
    ...
    ),

PROC read = ([] SIMPLIN) VOID;
```

As you can see, the mode SIMPLIN used for read is united from modes of names. The modes SIMPLOUT and SIMPLIN are more complicated than this (see chapters 9 and 11), but you now have the basic idea.

The uniting coercion is available in a firm context. This means that operators which accept operands with united modes will also accept operands whose modes are any of the constituent modes. We will return to this in the next section.

Here is an example of the uniting coercion in a call of the procedure print. If a has mode REF INT, b has mode [] CHAR and c has mode PROC REAL, then the call

```
print((a,b,c))
```

causes the following to happen:

1. a is dereferenced to mode INT and then united to mode SIMPLOUT.
2. b is united to mode SIMPLOUT.
3. c is deprocedured to produce a value of mode REAL and then united to mode SIMPLOUT.
4. The three elements are regarded as a row-display for a []SIMPLOUT.
5. print is called with its single parameter.

print uses a united-case-clause (see next section) to extract the actual value from each element in the row.

7.3 United-case-clauses

In the last section, we discussed the consequences of the uniting coercion; that is, how values of various modes can be united to values of united modes. This raises the question of how a value of a united mode can be extracted since its constituent mode cannot be determined at compile-time. There is no ”de-uniting” coercion in Algol 68. The constituent mode of the value can be determined using a variant of the integer-case-clause discussed in chapter 4 (see section 4.5). It is called a united-case-clause or conformity-clause. For our discussion, we will use the declaration of u in section 7.1 (u has mode CARDINAL).

The constituent mode of u can be determined by the following:

```
CASE u
IN (INT): print("u is an integer"),
   (STRING): print("u is a string")
ESAC
```

If the constituent mode of u is INT, the first case will be selected. Note that the mode selector is enclosed in parentheses and followed by a colon. Otherwise, the united-case-clause is just like the integer-case-clause (in fact, it is sometimes called a conformity-case-clause). This particular example could also have been written

```
CASE u
IN(STRING): print("u is a string")
OUT print("u is an integer")
ESAC
```

This is the only circumstance when a CASE clause can have one choice. Usually, however, we want to extract the value. A slight modification is required:

```
CASE u
IN (INT i): print(("u is an integer", i)),
   (STRING s): print(("u is a string", s))
ESAC
```

In this example, the declarer and identifier, which is called the specifier, act as the left-hand side of an identity declaration. The identifier can be used in the following unit (or enclosed-clause).

A specifier can also select a sub-set of a united mode. For example:

```
MODE ICS = UNION (INT, CHAR, STRING);

# Now define multiplication with an INT: #

OP * = (ICS a, INT b) ICS:
   CASE a
   IN (UNION (STRING, CHAR) ic):
         (ic | (CHAR c): c * b, (STRING s): s * b),
      (INT n): n * b
   ESAC
```

Note that united-case-clauses do not usually have an OUT clause; since they must be used to extract values, one usually defines a specific action by a united-case-clause on all possible modes at once. See the standard-prelude for more examples of united-case-clauses.

Although `read` and `print` use united modes in their call, you cannot read a value of a united mode or print a value of a united mode (remember that united modes do not introduce new values). You have to read a value of a constituent mode and then unite it, or extract the value of a constituent mode and print it.

7.4 Well-formed modes

It was already pointed out that not all possible mode declarations are allowed. Now that the description of mode constructing elements is complete, we can give the rules for determining whether a mode declaration is well-formed.

There are two reasons why a mode might not be well-formed:

1. the elaboration of a declaration using that mode would generate an infinitely large object, for instance

   ```
   MODE FRACTAL = [100] FRACTAL,
        LIST = STRUCT (INT index, LIST next)
   ```

2. coercion of that mode leads to an endless or ambiguous sequence of coercions.

   ```
   MODE POINTER = REF POINTER,
        CELL = UNION (STRING content, REF CELL next)
   ```

Let us look at some examples of modes which are not well-formed. Firstly, in the mode-declaration

```
MODE LIST = STRUCT (INT index, LIST next)
```

the `LIST` within the `STRUCT` would expand to a further `STRUCT` and so on *ad infinitum*. However, if the mode within the `STRUCT` is shielded by `REF` or `PROC`, then the mode-declaration is legal:

```
MODE LIST = STRUCT(INDEX index, REF LIST a);
```

In the declaration

```
LIST node = (1, LOC LIST)
```

the second field of the structure is a name referring to a structure which is quite different from generating the structure itself. Likewise, the declaration

```
MODE LIST = STRUCT (INT index, PROC LIST action)
```

is well-formed because in any declaration, the second field is a procedure (or a name referring to such a procedure) which is not the original structure and so does not require an infinite amount of storage.

It should be noted, however, that a `UNION` does not shield the mode as does `STRUCT`. The mode-declarations

```
MODE MWA = UNION (INT, REF MWA);
MODE MWB = STRUCT (UNION (CHAR, MWB) u, CHAR c)
```

are not well-formed. If we would have an object of mode `REF MWA` that would need coercion to `MWA` in a meek or strong context, the coercion could either be derefencing or uniting, which is ambiguous. `MWB` again leads to objects of infinite size since `MWB` must be expanded to know the size of the union.

All the above declarations have been recursive, but not mutually recursive. recursive. Is it possible to declare

```
MODE WA = STRUCT (WB wb, INT i),
     WB = STRUCT (WA wa, CHAR c)
```

Again, the elaboration of declarations using either mode would require an infinite amount of storage, so the modes are not well-formed. The following pair of mode-declarations are well-formed:

```
MODE RA = STRUCT (REF RB rb, INT i),
     RB = STRUCT (PROC RA pra, CHAR c)
```

All non-recursive mode-declarations are well-formed. It is only in recursive and mutually-recursive modes that we have to apply a test for well-formedness.

7.4.1 Determination of well-formedness

In any mutually-recursive mode-declarations, or any recursive mode declaration, to get from a particular mode on the left-hand side of a mode-declaration to the same mode indicant written on the right-hand side of a mode-declaration, it is necessary to traverse various mode constructors. Above each `STRUCT` or `PROC` with parameters parameters, write *yang*. Above each `REF` or parameter-less `PROC` write *yin*. A `UNION` gets neither *yin* nor *yang*. Now trace the path from the mode in question on the left-hand side of the mode-declaration until you arrive at the same mode indicant on the right-hand side. If you have at least one *yin* and at least one *yang*, the mode is well-formed — we say that the definition is properly shielded from its application.

Let us try this method on the recursive mode-declarations given in this section. In the mode-declaration

```
MODE LIST = STRUCT (INT index, LIST next)
```

write *yang* above the STRUCT. Thus to get from LIST on the left to LIST on the right, a single *yang* is traversed. Thus LIST is not well-formed. In the mode-declaration

```
MODE LIST = STRUCT(INDEX index, REF LIST next);
```

we have to traverse a *yang* (STRUCT) and a *yin* (REF), so LIST is well-formed.

Conversely, in

```
MODE MWA = UNION (INT, REF MWA);
MODE MWB = STRUCT (UNION (CHAR, MWB) u, CHAR c)
```

to get from MWA to MWA requires only *yin* so MWA is not well-formed. To get from MWB to MWB, only a STRUCT is traversed (*yang*) so MWB is also not well-formed.

Now consider the mutually-recursive mode-declarations of WA and WB:

```
MODE WA = STRUCT (WB wb, INT i),
     WB = STRUCT (WA wa, CHAR c)
```

At whichever mode we start, getting back to that mode means traversing two *yangs* (both STRUCT). Two *yangs* are all right, but there should be at least one *yin*, so the modes are not well-formed. On the other hand, in

```
MODE RA = STRUCT (REF RB rb, INT i),
     RB = STRUCT (PROC RA pra, CHAR c)
```

going from RA to RA traverses a STRUCT and a REF and, via RB, a STRUCT and a PROC giving *yang-yin-yang-yin*, so both RA and RB are well-formed.

Example

1. `MODE MA = INT` is well-formed.
2. `MODE MB = PROC (MB) VOID` is well-formed.
3. `MODE MB = PROC (MB) MB` is well-formed.
4. `MODE MC = [3, 2] MC` is not well-formed.
5. `MODE MD = STRUCT (BOOL p, MD m)` is not well-formed.
6. `MODE ME = STRUCT (STRING s, REF ME m)` is well-formed.

7.
```
MODE MFA = STRUCT (REF MFB f),
     MFB = PROC (INT) MFA
```
is well-formed.

8.
```
MODE MGA = PROC (MGB) VOID,
     MGB = STRUCT (MGA a)
```
is well-formed.

9.
```
MODE A = PROC (B) A,
MODE B = STRUCT (PROC C c, STRUCT (B b,INT i) d),
MODE C = UNION (A, B)
```
is not well-formed.

7.5 Equivalence of modes

In Algol 68, modes are equivalent if they have an equivalent structure. This principle is called structural equivalence. Compare this for example with Pascal where two objects are only of the same mode (type, in that language) when they are declared with the same type-identifier. The rules for structural equivalence in Algol 68 are intuitive:

1. rows are of equivalent mode when the elements have equivalent modes. Bounds are not part of a mode.
2. structures are equivalent when fields in a same position have equivalent modes and identical field names. Field names are part of a mode.
3. procedures are equivalent when parameters in a same position have equivalent modes, and the result modes are equivalent,
4. unions are equivalent when for every constituent mode in one there is an equivalent constituent mode in the other.

For instance, these two modes are equivalent:

```
MODE COMPLEX = STRUCT (REAL re, im), PERPLEX = STRUCT (REAL re, im)
```

and these two are not equivalent:

```
MODE COMPLEX = STRUCT (REAL re, im), POLAR = STRUCT (REAL r, theta)
```

but these are:

```
MODE FUNC = UNION (COMPLEX, PROC (COMPLEX) PERPLEX),
     GUNC = UNION (PERPLEX, PROC (PERPLEX) COMPLEX)
```

As you are well aware by now, mode-declarations can get quite entangled. The algorithm that *a68g* uses to determine the equivalence of two modes, is to prove equivalence assuming that the two are equivalent. The latter assumption is necessary since in many cases a proof of equivalence eventually comes down to ”circular” sub-proofs as $A = B$ *if and only if* $A = B$, and mentioned assumption resolves these situations.

Structural equivalence has an important consequence when defining modes for unrelated quantities. For instance, one could write:

```
MODE WEIGHT = REAL, DISTANCE = REAL;
WEIGHT w = read real, DISTANCE r = read real;
print (w + r) # ? #
```

It makes of course no sense to add a `WEIGHT` to a `DISTANCE`, but since both are structurally equivalent, a `REAL`, it is impossible to set them apart in Algol 68. We could pack them in a structure

```
MODE WEIGHT = STRUCT (REAL u), DISTANCE = STRUCT (REAL v);
```

but this no more than an inelegant trick. In Algol 68 there is no simple way to abstract from a mode’s representation, which is considered a weakness of the language.

Chapter 8

Transput

{"Purgamentum init, exit purgamentum"
Algol 68 Genie User Manual. Marcel van der Veer. }

8.1 Transput

Transput is a portmanteau word for input or output. At various points you have been reading data from standard input and writing data to standard output. These normally are your keyboard and screen, respectively. This chapter addresses the means whereby an Algol 68 program can obtain data from other devices and send data to devices other than the screen. Data transfer is called input-output in many other languages, but in Algol 68 it is called transput. We will use this term as a verb: we will use "to transput" meaning "to perform input or output".

This chapter describes transput as implemented by *a68g*. Though *a68g* transput preserves many characteristics of Algol 68 transput, there are differences.

Formally, Algol 68 discriminates files and books. A book was a datastructure having pages and lines (containing characters). At the time when Algol 68 was presented it was convenient to consider a deck of punch cards as a book of as many lines as there are cards, or to consider a line printer as a book with pages and lines, and it was not uncommon for a disk file to be organised as pages containing lines (of fixed or variable width).

This way of viewing files has changed over time. One of the important concepts from Linux is that a generalised mechanism exists for accessing a wide range of resources that are called "files": documents, directories, hard disks, CD/DVD drives, modems, keyboards, printers, monitors, terminals and even process - and network communications. This fundamental concept actually is twofold:

1. everything is a stream of bytes
2. the file system is a universal name-space

For this reason we will not talk about books with pages and lines. In *a68g*, we have ”files” that are sequences of bytes. An object of mode `FILE` is a file descriptor as in many other contemporary programming languages.

Algol 68 transput is ”event-driven”. You can at forehand specify actions to be taken when certain events occur during transput, and the transput routines will execute your preferred a ctions once an event occurs. This relieves you of explicitly having to check all kind of conditions yourself when reading or writing (such as: did you already encounter end of file, can you actually write to this file, did a conversion error take place, et cetera). We shall be examining later the kinds of event which can affect your programs if they read or write data.

8.2 Channels and files

Files have various properties. They usually have an identification string, the simplest example of which is a filename. Some files are read-only, some files are write-only and others permit both reading and writing. Some files allow you to browse: that is, they allow you to start anywhere and read (or write) from that point on. If browsing is allowed, you can restart at the beginning. Some files allow you to store data in text form (human-readable form) only, while others allow you to store values in a compact internal form known as binary. In the latter form, values are stored more or less in the same form as they are held in the program. The values will not usually be human-readable, being more suited to fast access by computer programs.

A file is associated with a channel. In the early days of Algol 68 channels could be considered as a description of a device-driver, but nowadays these details are abstracted in the kernel. They are maintained in *a68g* since they also specify access rights to a file. In *a68g*, five principal channels are provided in the standard-prelude: `stand in channel`, `stand out channel`, `stand error channel`, `stand back channel` and `stand draw channel`. The first is used for read-only files, the second and third for write-only files, the fourth for files which permit both reading and writing. The fifth does not allow for reading and writing, but specifies that a file is used as a plotting device; it comes close to the device-driver specification mentioned earlier.

This classification is a little over-simplified. Many files permit both reading and writing, but you may only want your program to read it. The first four standard channels mentioned are all buffered. This means that when you, for example, write data to a file, the data is collected in memory until a fixed amount has been transput, when the collection is written to the file in its entirety. The mode of a channel is `CHANNEL` and it is declared in the standard-prelude.

a68g keeps track of where you are in a file, which file is being accessed and whether you have come to the end of the file by means of a special structure which has the mode `FILE`. This is a complicated structure declared in the standard-prelude. The internals of values of mode `FILE` are likely to change from time to time, but the methods of using them will remain the same.

8.3 Reading files

Before you can read the contents of an existing file, you need to connect the file to your program. The procedure `open` with the header

```
PROC open = (REF FILE f, STRING idf, CHANNEL chan)INT:
```

performs that function. `open` yields zero if the connection is established and non-zero otherwise. Here is a program fragment which establishes communication with a read-only file whose identification is `testdata`:

```
FILE inf;

IF open(inf, "testdata", stand in channel) /= 0
THEN print(("Cannot open testdata", new line))
FI
```

The program displays a short message on the screen if for any reason the file cannot be opened.

After a file has been opened, data can be read from the file using the procedure `get`. It has the header

```
PROC get = (REF FILE f,
            [] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID
```

The mode `INTYPE` is declared by the Revised Report as an actual-declarer specifying a mode united from 'reference to flexible row of character' together with a sufficient set of modes each of which is 'reference to' followed by a mode which does not contain 'flexible', 'reference to', 'procedure' or 'union of'. `INTYPE` is processed by a process called straightening that converts any structure or row into a row of values of the constituent fields or elements. Allowed modes for latter elements in *a68g* are specified by the mode `SIMPLIN`:

```
MODE SIMPLIN = UNION (
   REF INT, REF REAL, REF COMPLEX,
   REF LONG INT, REF LONG REAL, REF LONG COMPLEX,
   REF LONG LONG INT, REF LONG LONG REAL, REF LONG LONG COMPLEX,
   REF BOOL,
   REF BITS, REF LONG BITS, REF LONG LONG BITS,
   REF CHAR, REF [] CHAR
)
```

This means that `get` can read directly any structure or row (or even rows of structures or rows).

The mode `BITS` is covered in chapter 11 together with `LONG` modes.

The `PROC` mode in

```
[] UNION (INTYPE, PROC (REF FILE) VOID) items
```

lets you use routines like `new page` and `new line` as parameters. The header of `new line` is

```
PROC new line = (REF FILE f) VOID
```

and you can call it from `get` if you want. On input, the rest of the current line is skipped and a new line started. The position in the file is at the start of the new line, just before the first character of that line.

Here is a program fragment which opens a file and then reads the first line and makes a name of mode `REF STRING` to refer to it. After reading the string, `new line` is called explicitly:

```
FILE inf;
open(inf, "file", stand in channel);
STRING line;
get(inf, line);
new line(inf)
```

This could equally well have been written

```
FILE inf;
open(inf, "file", stand in channel);
STRING line;
get(inf,(line, new line))
```

There is no reason why you should not declare your own procedures with mode `PROC (REF FILE) VOID`.

There are four names of mode `REF FILE` declared in the standard prelude. One of these is identified by `stand in`. The procedure `read` which you have already used is declared as

```
PROC read = ([] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID:
   get(stand in, items)
```

in the standard-prelude. As you can see, it gets data from `stand in`. If you want to, you can use `get` with `stand in` instead of `read`. The file `stand in` is already open when your program starts and should not be closed — unless you know what you are doing!.

When you have finished reading data from a file, you should sever the connection between the file and your program by calling the procedure `close`. This closes the file. Its header is

```
PROC close = (REF FILE f) VOID
```

8.4 Writing to files

You should use the `establish` procedure to create a new file. Here is a program fragment which creates a new file called `results`:

```
FILE outf;

IF establish(outf, "results", stand out channel) /= 0
THEN print(("Cannot establish file results", new line));
     stop
FI
```

As you can see, `establish` has a similar header to `open`. The header for `establish` is

```
PROC establish = (REF FILE f, STRING idf, CHANNEL chann) INT:
```

The procedure used to write data to a file is `put`. Its header is

```
PROC put =
(REF FILE f, [] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID:
```

The mode `OUTTYPE` is declared by the Revised Report as an actual-declarer specifying a mode united from a sufficient set of modes none of which is 'void' or contains 'flexible', 'reference to', 'procedure' or 'union of'. `OUTTYPE` is also processed by straightening that converts any structure or row into a row of values of the constituent fields or elements. Allowed modes for latter elements in *a68g* are specified by the mode `SIMPLOUT`:

```
MODE SIMPLIN = UNION (
   INT, REAL, COMPLEX,
   LONG INT, LONG REAL, LONG COMPLEX,
   LONG LONG INT, LONG LONG REAL, LONG LONG COMPLEX,
   BOOL,
   BITS, LONG BITS, LONG LONG BITS,
   CHAR, [] CHAR
)
```

Again, `new line` and `new page` can be used independently of `put` as in the following fragment:

```
FILE outf;
IF establish(outf, "newfile", stand out channel=) /= 0
THEN put(stand error, ("Cannot establish newfile"));
     stop
ELSE put(outf, ("Data for newfile", new line));
     close(outf)
FI
```

On output, the new line character is written to the file.

`new page` behaves just like `new line` except that a form feed character is searched for on input, and written on output.

The procedure `establish` can fail if the disk you are writing to is full or you do not have write access (in a network, for example) in which case it will return a non-zero value.

When you have completed writing data to a file, you must close it with the `close` procedure. This is particularly important with files you write to because the channel is buffered as explained above. Using `close` ensures that any remaining data in the buffer is flushed to the file.

The procedure `print` uses the `REF FILE` name `stand out`. So

```
print(("Your name", new line))
```

is equivalent to

```
put(stand out, ("Your name", new line))
```

Again, `stand out` is open when your program is started and it should not be closed. You cannot read from `stand out`, nor write to `stand in`. The procedure `write` is synonymous with `print`.

8.5 String terminators

One of the really useful facilities available for reading data from files is that of being able to specify when the reading of a string should terminate. Usually, this is set as the end of the line only. However, using the procedure `make term`, the string terminator can be a single character or any one of a set of characters. The header of `make term` is

```
PROC make term = (REF FILE f, STRING term) VOID
```

so if you want to read a line word by word, defining a word as any sequence of non-space characters, you can make the string terminator a space by writing

```
make term(inf, blank)
```

because `blank` (synonymous with `"␣"`) is rowed in the strong context of a parameter to `[]CHAR`. This will not remove the end-of-line as a terminator because the character `lf` is always added whatever characters you specify. You should remember that when a string is read, the string terminator is available for the next read — it has *not* been read by the previous read.

Note that in *a68g* it is possible to inquire about the terminator string of a file by means of the routine

```
PROC term = (REF FILE f) STRING
```

8.6 Events

Algol 68 transput is characterised by its use of events. *a68g* detects these events:

1. Reaching end of file.
2. Reaching end of line.
3. Reaching end of page.
4. Reaching end of format.
5. A file open error.
6. A value error.
7. A format error.
8. A transput error not caught by other events.

The default action when an event occurs depends on the event. However, the default action can be replaced by a programmer-defined action using one of the ”on”-procedures.

8.6.1 File end

When the end of a file is detected, the default action is to generate a runtime error. A programmer-supplied action must be a procedure with mode `PROC (REF FILE) BOOL`. The yield should be `TRUE` if some action has been taken to remedy the end of the file, in which case transput is resumed, or `FALSE`, when the default action will be taken.

The procedure `on logical file end` has the header

```
PROC on logical file end =
(REF FILE f, PROC (REF FILE) BOOL p) VOID
```

In *a68g*, `on file end`, `on logical file end` and `on physical file end` are the same procedures.

An example will help explain its use. Next program will display the contents of its text input file and print a message at its end:

```
IF FILE inf; [] CHAR infn = "file";
   open(inf, infn, stand in channel) /= 0
THEN put(stand error, ("Cannot open ", infn, new line));
     stop
ELSE on logical file end
        (inf, (REF FILE f) BOOL:
                 (write(("End of ", idf(f), new line));
                  close(f);
                  stop
                 )
        );
   STRING line;
   DO get(inf, (line, new line));
      write((line, new line))
   OD
FI
```

The anonymous procedure provided as the second parameter to `on logical file end` prints an informative message and closes the file. Note also that the `DO` loop simply repeats the reading of a line until the `logical file end` procedure is called by `get`. The procedure `idf` is described in section 8.13.

You should be careful if you do any transput on the parameter `f` in the anonymous routine otherwise you could get an infinite loop. Also, because the `on logical file end` procedure assigns its procedure parameter to its `REF FILE` parameter, you should be cautious when using `on logical file end` in limited ranges — a scope error may result.

Any piece of program which will yield an object of mode `PROC (REF FILE) BOOL` in a strong context is suitable as the second parameter of `on logical file end`. If you want to reset the action to the default action, use the statement

```
on logical file end(f, (REF FILE) BOOL: FALSE)
```

8.6.2 Line end and page end

These events are caused when while reading, the end of line or end of page is encountered. These are events so you can provide a routine that for instance automatically counts the number of lines or pages read. The procedures `on line end` and `on page end` let the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`.

If the programmer-supplied routine yields `TRUE`, transput continues, otherwise a `new line` in case of end of line, or `new page` in case of end of page, is executed and than transput resumes.

Be careful when reading strings, since end of line and end of page are string terminators! If you provide a routine that mends the line - or page end, be sure to call `new line` or `new page` before returning `TRUE`. In case of a default action, `new line` or `new page` must be called explicitly, for instance in the `read` statement, otherwise you will read nothing but empty strings as you do not eliminate the terminator.

8.6.3 Format end

This event is caused when in formatted transput, the format gets exhausted. The procedure `on format end` lets you provide a procedure of mode `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, transput simply continues, otherwise the format that just ended is restarted and transput resumes.

8.6.4 Open error

This event is caused when a file cannot be opened as required. For instance, you want to read a file that does not exist, or write to a read-only file. The procedure `on open error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the program continues, otherwise a runtime error occurs.

8.6.5 Value error

This event is caused when transputting a value that is not a valid representation of the mode of the object being transput. The procedure `on value error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If you do transput on the file within the procedure ensure that a value error will not occur again! If the programmer-supplied routine yields `TRUE`, transput continues, otherwise a runtime error occurs.

8.6.6 Format error

This event is caused when an error occurs in a format, typically when patterns are provided without objects to transput. The procedure `on format error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the program continues, otherwise a runtime error occurs.

8.6.7 Transput error

This event is caused when an error occurs in transput that is not covered by the other events, typically conversion errors (value out of range et cetera). The procedure `on transput error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the program continues, otherwise a runtime error occurs.

8.7 Formatting routines

One of the problems of using the rather primitive facilities given so far for the output of real and integer numbers is that although they allow numbers to be printed in columns, the column widths are fixed. You might not always want a fixed number of decimal places. It is necessary to have some means of controlling the size of the resulting strings. The procedures `whole`, `fixed`, `float` and `real` provide this capability. Transput of data in Algol 68 is organised such that values written by an Algol 68 program can be read by another (or the same) program.

The procedure `whole` has the header

```
PROC whole = (NUMBER v, INT width) STRING
```

and takes two parameters. The first is a real or integer value (modes `REAL` or `INT`) and the second is an integer which tells `whole` the field width of the output number (the space in your output file required to accommodate a value is often called a field). If $width = 0$, then the number is printed with the minimum possible width and this will be wider for larger numbers. A positive value for `width` will give numbers preceded by a "+" if positive and a "-" if negative and the number output right-justified within the field with spaces before the sign. A negative value for `width` will provide a minus sign for negative numbers but no plus sign for positive numbers and the width will be `ABS width`.

Note that where the integer is wider than the available space, the output field is filled with the character denoted by `error char` (which is declared in the standard-prelude as the asterisk (`*`) with mode `CHAR`), so it is wise to ensure that your output field is large enough to accommodate the largest number you might want to print.

If you pass a real number to `whole`, it calls the procedure `fixed` with parameters `width` and 0.

The procedure `fixed` has the header

```
PROC fixed = (NUMBER v, INT width, after) STRING
```

and takes three parameters. The first two are the same as those for `whole` and the third specifies the number of decimal places required. The rules for `width` are the same as the rules for `width` for `whole`.

When you want to print numbers in scientific format (that is, with an exponent), you should use `float` which takes four parameters and has the header

```
PROC float = (NUMBER v, INT width, after, exp) STRING
```

The first three are the same as the parameters for `fixed`, while the fourth is the width of the exponent field. The version of `float` supplied in the transput library uses `e` to separate the exponent from the rest of the number. Thus the call

```
print(float(pi * 10, 8, 2, -2)
```

produces the output `+3.14e 1`. The parameter `exp` obeys the same rules (applied to the exponent) as `width`.

a68g implements a further routine `real` for formatting reals in a way closely related to `float`, declared as

```
PROC real =
(NUMBER x, INT width, after, exp width, modifier) STRING:
```

It converts `x` to a `STRING` representation. If `modifier` is a positive number, the resulting string will present `x` with its exponent a multiple of `modifier`. If $modifier = 1$, the returned string is identical to that returned by `float`. A common choice for `modifier` is 3 which returns the so-called engineers notation of `x`. If `modifier` is zero or negative, the resulting string will present `x` with `ABS modifier` digits before the decimal point and its exponent adjusted accordingly; compare this to FORTRAN nP syntax.

8.8 Straightening

The term straightening is used in Algol 68 to mean the process whereby a compounded mode is separated into its constituent modes, which are themselves straightened if required. . For example, the mode

```
STRUCT (INT a, CHAR b, UNION (REAL, STRING) u)
```

would be straightened into values of the following modes:

1. `INT`
2. `CHAR`
3. `UNION(REAL, STRING)`

The mode

```
REF STRUCT (INT a, CHAR b, UNION (REAL, STRING) u)
```

would be straightened into a number names having the modes

1. `REF INT`
2. `REF CHAR`
3. `REF UNION (REAL, STRING)`

However, a value of mode `COMPLEX` is not straightened into two values both of mode `REAL`.

Also, any row is separated into its constituent elements. For example

```
[1 : 3] COMPLEX
```

will be straightened into three values of mode `COMPLEX`.

8.9 Default-format transput

In default-format transput, each primitive mode is written as follows:

1. `CHAR`
   Output a character to the next logical position in the file.
2. `[] CHAR`
   Output all the characters on the current line.
3. `BOOL`
   Output `flip` or `flop` for `TRUE` or `FALSE` respectively. For `[] BOOL`, output `flip` or `flop` for each `BOOL`.
4. `L BITS`
   Output `flip` for each bit equal to one and `flop` for each bit equal to zero. `l bits width` characters are output in all. No new line or new page is output. For `[] L BITS`, each `BITS` value is output as above with no intervening spaces.

5. `L INT`
   Output the integer using the call

   ```
   whole(i, L int width + 1)
   ```

   which will right-justify the integer in

   ```
   L int width + 1
   ```

   positions with a preceding sign. For `[] L INT`, each integer is output as described above, preceded by a space if it is not at the beginning of the line. No new lines or new pages are output.

6. `L REAL`
   Output the real using the call

   ```
   float (r, L real width + L exp width + 4,
             L real width - 1, L  exp width + 1)
   ```

   which will right-justify the real in

   ```
   L real width + L exp width + 4
   ```

   positions preceded by a sign.

7. `L COMPLEX`
   The complex value is output as two real numbers in floating-point format. For `[] L COMPLEX`, each complex value is output as described above.

8. `PROC (REF FILE) VOID`
   An `lf` character is output if the routine is `newline` and an `ff` character if the routine is `new page`. User-defined routines with this mode can be used.

In default-format transput, each primitive mode is read as follows:

1. `REF CHAR`
   Any characters `c` where `c < blank` are skipped and the next character is assigned to the name. If a `REF [] CHAR` is given, then the above action occurs for each of the required characters of the row.

2. `REF STRING`
   All characters, including any control characters, are assigned to the name until any character in the character set specified by the `string term` field of `f` is read. The file is then backspaced so that the string terminator will be available for the next `get`.

3. `REF BOOL`
   The next non-space character is read and, if it is neither `flip` nor `flop`, the char error procedure is called with `flop` as the suggestion. For `REF [] BOOL`, each `REF BOOL` name is assigned a value as described above.

4. `REF L BITS`
   The action for `REF BOOL` is repeated for each bit in the name. For `REF[]L BITS`, each `REF L BITS` name is assigned a value as described above.

5. `REF L INT`
   If the next non-control character (*ie*, a character which is neither a space, a tab character, a new line or new page character or other control character) is not a decimal digit, then the char error procedure is called with `"0"` as the suggestion. Reading of decimal digits continues until a character which is not a decimal digit is encountered when the file is backspaced. If during the reading of decimal digits, the value of `l max int` would be exceeded, reading continues, but the input value is not increased. For `REF [] L INT`, each integer is read as described above.

6. `REF L REAL`
   A real number consists of 3 parts:

   (a) an optional sign possibly followed by spaces

   (b) an optional integral part

   (c) a `"."` followed by any number of control characters (such as new line or tab characters) and the fractional part

   (d) an optional exponent preceded by a character in the set `"Ee\"`. The exponent may have a sign. Absence of a sign is taken to mean a positive exponent

   The number may be preceded by any number of control characters or spaces. For `REF [] L REAL`, each `REAL` value is read as described above.

7. `REF L COMPLEX`
   Two real numbers separated by `"i"` are read from the file. Newlines or new pages or other control characters can precede each real. The first number is regarded as the real part and the second the imaginary part. For `REF [] L COMPLEX`, each `REF L COMPLEX` is read as described above.

8.10 Formatted transput

Formatted transput gives the programmer a high degree of control over the transput of values. If you program in FORTRAN or in C then the concept of formatted transput is not strange to you. Algol 68 implements a similar mechanism but in its own orthogonal way.

A format is a description of how values should be transput. Formats actually are objects of mode `FORMAT`. The behaviour of an object of mode `FORMAT` is like that of a routine; they have a ”denotation” that is called a format-text, you can pass formats to routines, have them yielded by routines, assign them, et cetera. Like routines there are no procedures or operators on formats in the standard-prelude, but if you can think of some useful application then by all means, declare them in your program.

Formats are associated with files by including them in the parameter list of routines `putf` and `getf`. There are (of course) related routines `printf` and `readf` that perform `putf` on `standout`, or `getf` on `standin`, respectively. These routines have respective headers:

```
PROC putf = (REF FILE f, [] UNION (OUTTYPE,
             PROC (REF FILE) VOID, FORMAT) items) VOID

PROC getf = (REF FILE f, [] UNION (INTYPE,
             PROC (REF FILE) VOID, FORMAT) items) VOID

PROC printf =
([] UNION (OUTTYPE, PROC (REF FILE) VOID, FORMAT) items) VOID

PROC readf =
([] UNION (INTYPE, PROC (REF FILE) VOID, FORMAT) items) VOID
```

8.10.1 Format texts — insertions and patterns

format-texts In this section we will introduce formatted transput by describing the constituting constructs.

In FORTRAN, a format is defined by a format statement like

```
      WRITE (6, 10) N, RESULT
   10 FORMAT (/1X, 'RESULT', 1X, I3, '=', 1X, F9.5)
```

A format in Algol 68 is a value of mode `FORMAT`. The standard-prelude defines the mode `FORMAT` as a structure the fields of which are inaccessible to you. You can have format identities, format variables, procedures yielding formats et cetera. There are however no predefined operators for objects of mode `FORMAT`. You can define them yourself but since the fields are inaccessible to you there is little sense in passing a format as operand. In *a68g*, formats behave like anonymous routines and follow the scope rules of routine-texts.

A format has a ”denotation” that is called a format-text, for instance:

```
$l, 1x, "Result", 1x, 2zd, "=", 1x, g(9.5)$
```

and you could write

```
FORMAT my format = $l, 1x, "Result", 1x, 2zd, "=", 1x, g(9.5)$;
FORMAT variable format := my format
```

but typically a format-text is used as a parameter to transput routines:

```
printf(($l, 1x, "Result", 1x, 2zd, "=", 1x, g(9.5)$, n, result))
```

A format-text has this general syntax:

```
format-text:
   '$' picture-list '$'.
```

which introduces a meaning for the dollar-sign in Algol 68: they are delimiters for format texts. A format-text is a list of pictures; each picture representing an individual item to be transput:

```
picture:
   insertion,
   pattern,
   collection,
   replicator collection.
```

Before describing other elements, a replicator is a static or dynamic specification of how often a subsequent construct should be repeated:

```
replicator:
   integer-denotation, 'n' meek-integer-enclosed-clause.
```

Examples of replicators are

```
6             # Repeat six times   #
n(k + 1)      # Repeat k + 1 times #
n(read int)   # Repeat as many times as you type
                on the keyboard at this point #
```

A replicator should not be less than zero. Algol 68 states that a negative replicator should be treated as if it were zero, but *a68g* assumes that a negative replicator means that something went wrong and will give a runtime error.

A collection is simply a way to group a list of pictures, typically used when a replicator must be applied to it:

```
collection:
   '(' picture-list ')'.
```

Insertions give you control over new lines, new pages, spaces forward, or lets you insert string literals:

```
insertion:
   (replicator ‘k’,
   [replicator] ‘l’,
   [replicator] ‘/’,
   [replicator] ‘p’,
   [replicator] ‘x’,
   [replicator] ‘y’,
   [replicator] ‘q’,
   [replicator] (row-of-character-denotation)-sequence.
```

1. ”k” performs as many spaces forward as its replicator indicates.
2. ”l” is a new line; it calls `new line`.
3. ”/” is a new line, as in FORTRAN; it calls `new line`.
4. ”p” is a new page, it calls `new page`.
5. ”x” is a space, as in FORTRAN; it calls `space`.
6. ”y” is a backspace, it calls `backspace`.
7. ”q” is a space; it calls `space`.
8. ”(row-of-character-denotation)-sequence” is a sequence of string literals to be taken literally.

Examples:

```
20k
/
3/
n(k + 1)x
"Algol˜68"
```

Here is an example program that will draw a sine-curve on your screen:

```
INT n = 24;
FOR i TO n
DO REAL x = 2 * pi * (i - 0.5) / n;
   printf($l, n(40 -  ROUND (30 * sin (x)))k, "*"$)
OD
```

Patterns specify how to

1. transput a value of a standard mode.

2. direct transput depending on a boolean - or integer expression.
3. execute an embedded format.

We therefore have these patterns:

```
pattern:
   general-pattern,
   integer-pattern,
   real-pattern,
   complex-pattern,
   bits-pattern,
   string-pattern,
   boolean-pattern,
   choice-pattern,
   format-pattern.
```

8.10.2 General patterns

The general-pattern performs transput following default formatting:

```
general-pattern:
   ‘g’[strong-row-of-integer-enclosed-clause],
   ‘h’[strong-row-of-integer-enclosed-clause].
```

Any standard mode can be transput with ”g” or ”h” when the strong-row-of-integer-enclosed-clause is absent; default formatting is used which is the formatting that default-format routines `read`, `get`, `out` or `print` would apply. In case of an integer or real value to transput, the length of the strong-row-of-integer determines the formatting to be applied. In case of letter ”g” we get:

1. 1 element `(i)`. `whole (..., i)` is called.
2. 2 elements `(i, j)`. `fixed (..., i, j)` is called.
3. 3 elements `(i, j, k)`. `float(..., i, j, k)` is called.

In case of letter ”h” we always get scientific notation. Let `K` be 1 + the value `exp with` for the length of the integer or real value to transput, then we have this behaviour for ”h”:

1. 1 element `(i)`. `real(..., i, i + K + 4, K, 3)` is called. Since this gives `i` decimals before the exponent is adjusted to be a multiple of 3, we get `i + 1` digits in engineers notation and the width is automatically adjusted.

2. 2 elements `(i, j)`. `real(..., i, i + K + 4, K, j)` is called. If you want to print a number with exactly `i` decimals, you specify `h(i, 1)` and the width is automatically adjusted.

3. 3 elements `(i, j, k)`. `real(..., i, j, K, k)` is called. This prints as much digits as will fit in a width `i`, including `j` decimals, and the exponent is adjusted to be a multiple of `k`.

4. 4 elements `(i, j, k, l)`. `real(..., i, j, k, l)` is called. This gives you full control of the function `real` as no defaults are applied.

8.10.3 Integer patterns

An integer-pattern transputs integer values and consist of an optional sign-mould that lets you decide how to transput a sign, followed by an integer-mould that specifies how to transput each individual digit. We will meet elements here that are called frames: "s", "z" and "d". Frame "s" means that the next frame will be suppressed, "z" means to print a non-zero digit or a space otherwise, and "d" means to print a digit.

```
integer-pattern:
   [sign-mould] integer-mould.
```

Hence the sign-mould determines how to put a sign, would you want one:

```
sign-mould:
   [replicator] ['s'] ('z', 'd')]
   [([replicator] ['s'] ('z', 'd'), insertion)-sequence]
   ('+', '-').
```

Frames "+" and "-" are sign-frames; "+" forces a sign to be put, but a "-" only puts a minus if the number is negative. Example sign-moulds are:

```
+
-
zzz-
3z-
```

If you specify "z" frames before the sign "+" or "-" then the sign will shift left one digit for every non-zero digit that is begin put. In *a68g*, when a sign is shifted in a sign-mould, any character output by literal insertions in that sign-mould is replaced with a space as well, starting from the first z-frame until the sign is put.

```
integer-mould:
   [replicator] ['s'] ('z', 'd')
   ([replicator] [‘s’] (‘z’, ‘d’), insertion)-sequence.
```

The insertions let you put all kind of literals in between digits. For example:

```
4d          # transputs 9 as "0009", error on negative
              numbers since there is no sign-mould    #
zzz-d       # transputs -9 as " -9"                   #
3z-d        # transputs -9 as " -9"                   #
3z","2z-d   # transputs 100000 as " 100,000"          #
3d"-"3d"-"4d # transputs 5551234567 as 555-123-4567   #
```

8.10.4 Real patterns

A real-pattern follows the same logic as an integer-pattern. Sign-moulds are optional, and the exponent-part is optional. The decimal-point-frame and the exponent-frame ”e” are suppressible by preceding them with ”s”; this can for instance be used to replace the standard exponent character by an insertion of your liking.

```
real-pattern:
   [sign-mould] [‘s’] ‘.’ [insertion] integer-mould
   [[‘s’]‘e’ [insertion] [sign-mould] integer-mould].
```

Examples are:

```
d.3d             # transputs pi as "3.142"         #
d.3dez-d         # transputs 0.3333 as "3.333e -1" #
ds.","3dse"ˆ"z-d # transputs 0.3333 as "3,333ˆ -1" #
```

8.10.5 Complex patterns

A complex-pattern consists of a real-pattern for the real part and a real-pattern for the imaginary part of the value. The plus-i-times-frame ”i” is suppressible and can thus be replaced by an insertion.

```
complex-pattern:
   real-pattern [‘s’]‘i’ [insertion] real-pattern.
```

Examples:

```
-d.3di-d.3d       # transputs f.i. 0.000i-1.000 #
-d.3dsi-d.3d, "j" # transputs f.i. 0.000-1.000j #
```

8.10.6 Bits patterns

A bits-pattern has no sign-mould, since a `BITS` value is unsigned. In *a68g* the radix is specified by a replicator by which it can be dynamic.

```
bits-pattern:
   replicator ‘r’ integer-mould.
```

Examples:

```
2r7zd # binary 8-bit byte with trailing zero-bit suppression #
16r8d # hexadecimal 32 bit word without trailing zero-digit
        suppression                                         #
```

8.10.7 String patterns

The string-pattern transputs an object of modes `STRING`, `CHAR` and `[] CHAR`. The ”a” frames can be suppressed.

```
string-pattern:
   [replicator] [‘s’] ‘a’
   [([replicator] [‘s’] ‘a’, insertion)-sequence].
```

Examples:

```
printf ($7a$, "Algol68")     # prints "Algol68"  #
printf ($5a"-"2a, "Algol68") # prints "Algol-68" #
printf ($5a2sa, "Algol68")   # prints "Algol"    #
```

8.10.8 Boolean patterns

The boolean-expression transputs a `BOOL` value.

```
boolean-pattern:
   ‘b’,
   ‘b’ ‘(’ row-character-denotation ‘,’ row-character-denotation ‘)’.
```

If you just want the standard ”T” or ”F” transput, do not specify the row-of-character-denotations. If you do specify those, a boolean-pattern is the format-text equivalent of a conditional-clause. Example:

```
printf ($b ("true", "not true")$, read bool)
```

8.10.9 Choice patterns

A choice-pattern transputs a row-character-denotation as a function of an integer value. It is the format-text equivalent of an integer-case-clause.

```
choice-pattern:
   ‘c’ ‘(’ row-character-denotation-list ‘)’.
```

Examples:

```
printf ($c ("one", "two", "three")$, read int)
```

The Revised Report specification of getting using an integral choice pattern has the peculiarity that when two literals start with the same sequence of characters, the longer literal should appear first in the list. *a68g* makes no such demand, and will select the correct literal from the list whatever their order.

8.10.10 Format patterns

The format-pattern lets you dynamically choose the format you want to employ.

```
format-pattern:
   ‘f’ meek-format-enclosed-clause.
```

Here is an example where a format may be selected dynamically:

```
f (um | (INT): $g(0)$, (REAL): $g(0, 4)$)
```

The effect of a format-pattern is that is temporarily supersedes the active format for the file at hand. When this temporary format ends, the original format continues without invoking an format-end event.

8.11 Binary files

Files that are not meant to be inspected by humans can be in a compact form. Many large files will be stored in this form. They are called binary or unformatted files. Algol 68 allows you to write anything to binary files, just as for text files.

The only difference between transput to, or from, binary files is that instead of using the procedures `put` and `get`, you use the procedures `put bin` and `get bin`. The modes accepted by these procedures are identical with those accepted by `put` and `get` respectively.

Values of mode `REF STRING` can be read by `get bin`, but you should remember to set the string terminator using the procedure `make term`. However, you should note that the string terminator will *always* include new line and new page.

8.12 Internal files

Sometimes what you want is to write a large amount of data to a file, and later read it again in the same program. In such case you may consider using an internal file. Instead of locating the file on a device such as a hard disk, an internal file is associated with an object of mode `STRING` that will hold the data; hence file contents reside in RAM. To use an internal file, one uses the routine `associate` which is declared as

```
PROC associate = (REF FILE, REF STRING) VOID
```

On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition. When a file that is associated with a string is reset, getting restarts from the start of the associated string. Next (trivial) code fragment illustrates the use of associate in transput.

```
[n] COMPL u, v;
# code to give values to elements in u #
FILE in, out,
STRING z;
associate (in, z);
associate (out, z);
putf (out, u);
getf (in, v);
```

8.13 Other transput procedures

The procedure `idf` has the header

```
PROC idf = (REF FILE f) STRING:
```

and yields the identification of the file handled by the file `f`.

There are two other ways of closing a file. One is `scratch` and the other is `lock`:

```
PROC scratch = (REF FILE f) VOID:
PROC lock = (REF FILE f) VOID:
```

The procedure `scratch` deletes the file once it is closed. It is often used with temporary files. The procedure `lock` closes its file and then locks it so that it cannot be opened without some system action.

It is possible to rewind a file, that is, to let the next transput operation again start from the beginning of a file. The procedure provided for this purpose is `reset` which has the header

```
PROC reset = (REF FILE f) VOID
```

One possible use of this procedure is to output data to a file, then use `reset` followed by `get` to read the data from the file. In fact, the standard-prelude declares and opens such a file that is called `stand back`. It uses the `stand back channel`. Note that the procedure `read bin` is equivalent to `get bin(stand back, ...)` and the procedure `write bin` is equivalent to `put bin(standback, ...)`.

There is also a procedure `set` that attempts to set the file pointer to a position other than the beginning of the file as done by `reset`:

```
PROC set = (REF FILE f, INT n) INT
```

This routine deviates from the standard Algol 68 definition. It attempts to move the file pointer of `f` by `n` character positions with respect to the current position. If the file pointer would as a result of this move get outside the file, it is not changed and the routine set by `on file end` is called. If this routine returns `FALSE`, and end-of-file runtime error is produced. The routine returns an `INT` value representing system-dependent information on this repositioning. Whether a file allows setting, can be interrogated with set possible:

```
PROC set possible = (REF FILE f) BOOL
```

which does what it suggests - yield `TRUE` when `f` can be set, or `FALSE` otherwise.

Related to `set` are the routines `space` and `backspace`.

```
PROC space = (REF FILE f) VOID
```

The procedure advances the file pointer in file `f` by one character. It does *not* read or write a blank.

```
PROC backspace = (REF FILE f) VOID
```

The procedure attempts to retract the file pointer in file `f` by one character. It actually executes

```
VOID (set(f, -1))
```

8.14 Appendix: Specification of formatting routines

Next listing is a specification of `whole`, `fixed`, `float` and `real` implemented by *a68g*. It closely follows the Revised Report definition, that however does not define `real`:

```
# Actual implementation also includes LONG and LONG LONG modes #
MODE NUMBER = UNION (INT, REAL);
CHAR error char = "*";

# REM is not the same as MOD #
OP REM = (INT i, j) INT: (i - j * (i OVER j));
PRIO REM = 7;

PROC whole = (NUMBER v, INT width) STRING:
   CASE v IN
      (INT x):
       (INT length := ABS width - (x < 0 OR width > 0 | 1 | 0),
        INT n := ABS x;
        IF width = 0 THEN
            INT m := n; length := 0;
            WHILE m OVERAB 10; length +:= 1; m /= 0
            DO SKIP OD
        FI;
        STRING s := sub whole (n, length);
        IF length = 0 OR char in string (error char, LOC INT, s)
        THEN ABS width * error char
        ELSE
           (x < 0 | "-" |: width > 0 | "+" | "") PLUSTO s;
           (width /= 0 | (ABS width - UPB s) * " " PLUSTO s);
           s
        FI),
       (REAL x): fixed (x, width, 0)
   ESAC;

PROC fixed = (NUMBER v, INT width, after) STRING:
   CASE v IN
      (REAL x):
       IF INT length := ABS width - (x < 0 OR width > 0 | 1 | 0);
          after >= 0 AND (length > after OR width = 0)
       THEN REAL y = ABS x;
          IF width = 0
          THEN length := (after = 0 | 1 | 0);
             WHILE y + .5 * .1 ^ after > 10 ^ length
             DO length +:= 1 OD;
             length +:= (after = 0 | 0 | after + 1)
          FI;
          STRING s := sub fixed (y, length, after);
          IF NOT char in string (error char, LOC INT, s)
          THEN (length > UPB s AND y < 1.0 | "0" PLUSTO s);
               (x < 0 | "-" |: width > 0 | "+" | "") PLUSTO s;
               (width /= 0 | (ABS width - UPB s) * " " PLUSTO s);
               s
          ELIF after > 0
          THEN fixed (v, width, after - 1)
          ELSE ABS width * error char
          FI
       ELSE ABS width * error char
       FI,
      (INT x): fixed (REAL(x), width, after)
   ESAC;

PROC real = (NUMBER v, INT width, after, exp, modifier) STRING:
   CASE v IN
      (REAL x):
       IF INT before = ABS width - ABS exp -
                        (after /= 0 | after + 1 | 0) - 2;
          INT mod aft := after;
          SIGN before + SIGN after > 0
       THEN STRING s, REAL y := ABS x, INT p := 0;
          standardize (y, before, after, p);
          IF modifier > 0
          THEN WHILE p REM modifier ~= 0
               DO y *:= 10;
                  p -:= 1;
                  IF mod aft > 0
                  THEN mod aft -:= 1
                  FI
               OD
          ELSE WHILE y < 10.0 ^ (- modifier - 1)
               DO y *:= 10;
                  p -:= 1;
                  IF mod aft > 0
                  THEN mod aft -:= 1
                  FI
               OD;
               WHILE y > 10.0 ^ - modifier
               DO y /:= 10;
                  p +:= 1;
                  IF mod aft > 0
                  THEN mod aft +:= 1
                  FI
               OD
          FI;
          s := fixed (SIGN x * y,
                      SIGN width * (ABS width - ABS exp - 1), mod aft) +
                      "E" + whole (p, exp);
          IF exp = 0 OR char in string (error char, LOC INT, s)
          THEN float (x, width,
                      (mod aft /= 0 | mod aft - 1 | 0),
                      (exp > 0 | exp + 1 | exp - 1))
          ELSE s
          FI
       ELSE ABS width * error char
       FI,
      (INT x): float (REAL(x), width, after, exp)
   ESAC;
```

```

PROC float = (NUMBER v, INT width, after, exp) STRING:
   real (v, width, after, exp, 1);

PROC sub whole = (NUMBER v, INT width) STRING:
   CASE v IN
      (INT x):
       BEGIN STRING s, INT n := x;
          WHILE dig char (n MOD 10) PLUSTO s;
              n OVERAB 10; n /= 0
          DO SKIP OD;
          (UPB s > width | width * error char | s)
       END
   ESAC;

PROC sub fixed = (NUMBER v, INT width, after) STRING:
   CASE v IN
      (REAL x):
       BEGIN STRING s, INT before := 0;
          REAL y := x + .5 * .1 ^ after;
          PROC choosedig = (REF REAL y) CHAR:
             dig char ((INT c := ENTIER (y *:= 10.0);
                        (c > 9 | c := 9); y -:= c; c));
          WHILE y >= 10.0 ^ before DO before +:= 1 OD;
          y /:= 10.0 ^ before;
          TO before DO s PLUSAB choose dig(y) OD;
          (after > 0 | s PLUSAB ".");
          TO after DO s PLUSAB choose dig(y) OD;
          (UPB s > width | width * errorchar | s)
       END
   ESAC;

PROC standardize = (REF REAL y, INT before, after, REF INT p) VOID:
   BEGIN
      REAL g = 10.0 ^ before, REAL h := g * .1;
      WHILE y >= g DO y *:= .1; p +:= 1 OD;
      (y /= 0.0 | WHILE y < h DO y *:= 10.0; p -:= 1 OD);
      (y + .5 * .1 ^ after >= g | y := h; p +:= 1)
   END;

PROC dig char = (INT x) CHAR: "0123456789abcdef" [x + 1];

SKIP
```

Chapter 9

Units and coercions

9.1 Statements

This chapter describes units, contexts and coercions, as well as balancing. In this manual, a statement is a declaration or a unit. Declarations yield no value, even if they include an initial assignation. Units are the parts of the language which actually yield values arranged in this hierarchy:

```
   Units
      Tertiaries
         Secondaries
            Primaries
```

where each class includes the lower class. For example, all enclosed-clauses are primaries, but not all primaries are enclosed-clauses. This hierarchy of units prevents writing ambiguous programs; it specifies for example that

```
z[k] := z[k] + 1
```

can only mean

```
((z)[(k)]) := (((z)[(k)]) + (1))
```

or that

```
ref OF ori OF z :=: NIL
```

can only mean

```
(ref OF (ori OF (z))) :=: (NIL)
```

The units in each class are as follows:

1. unit
   (a) assignation
   (b) identity-relation
   (c) routine-text
   (d) jump
   (e) skip
   (f) assertion
   (g) conditional-unit
   (h) tertiary
2. tertiary
   (a) formula
   (b) nihil
   (c) stowed-function
   (d) secondary
3. secondary
   (a) generator
   (b) selection
   (c) primary
4. primary
   (a) identifier
   (b) call
   (c) cast
   (d) denotation
   (e) slice
   (f) enclosed-clause
5. enclosed-clause
   (a) closed-clause
   (b) collateral-clause
   (c) parallel-clause

(d) conditional-clause
(e) integer-case-clause
(f) united-case-clause
(g) loop-clause

9.2 Contexts

The circumstances which allow certain coercions are called contexts. Each context has an intrinsic strength. There are five contexts called strong, firm, meek, weak and soft. A strong context allows all coercions, whereas a soft context allows only one. The places in a program which have these contexts are:

1. Strong contexts
   (a) The actual-parameters of calls
   (b) The enclosed-clauses of casts
   (c) The source unit of assignations
   (d) The source unit of identity-declarations
   (e) The source unit of variable-declarations
   (f) The units of routine-texts
   (g) `VOID` units
   (h) Subordinated constituents of a balanced construct
2. Firm contexts
   (a) Operands of formulas
3. Meek contexts
   (a) Enquiry-clauses
   (b) Primaries of calls
   (c) The units following `FROM`, `BY` and `TO` or `DOWNTO`
   (d) Units in trimmers, subscripts and bounds
   (e) Enclosed-clause of a format-pattern
   (f) Enclosed-clause of a replicator
   (g) Left-hand side integral-tertiary of stowed-functions
   (h) The boolean-enclosed-clause of an assertion

4. Weak contexts
    (a) Primaries of slices
    (b) Secondaries of selections
    (c) Right-hand side tertiary of a stowed-function
5. Soft contexts
    (a) The destination tertiary of assignations
    (b) One tertiary in an identity-relation (to adapt it to the other tertiary)

9.3 Coercions

There are six coercions in the language, namely

1. dereferencing
2. deproceduring
3. uniting
4. widening
5. rowing
6. voiding

Allowed coercions per context are:

1. Strong context
    (a) dereferencing or deproceduring
    (b) uniting
    (c) widening
    (d) rowing
    (e) voiding
2. Firm context
    (a) dereferencing or deproceduring
    (b) uniting

3. Meek context
    (a) dereferencing or deproceduring
4. Weak context
    (a) dereferencing or deproceduring yielding a name if possible or a value otherwise.
5. Soft context
    (a) deproceduring

Note that first a sequence of deproceduring or dereferencing — alternatingly if needed — can take place, then uniting, then a sequence of widenings, after that rowing and finally voiding can take place. For example, you cannot coerce `[] INT` to `[] REAL`.

In any context, you can use a cast (see the section on primaries further on) which will always make a context strong and because all coercions are available in a strong context, you can use the cast to specify the mode you require (provided the source mode can be coerced to the required mode in a strong context).

9.3.1 Deproceduring

This coercion is available in all contexts. Deproceduring is the mechanism by which a parameter-less procedure is called. For example, a procedure having mode `PROC REAL`, when called yields a `REAL`. We can represent the coercion by

$$\texttt{PROC REAL} \Rightarrow \texttt{REAL}$$

The `PROC` is ”removed”, which is why it is called deproceduring. The coercion can be applied repeatedly, for instance

$$\texttt{PROC PROC REAL} \Rightarrow \texttt{PROC REAL} \Rightarrow \texttt{REAL}$$

Deproceduring only occurs with parameter-less procedures, and only if the context requires an object of the mode of the yield of the procedure in question.

9.3.2 Dereferencing

This is the process of moving from a name to the value to which it refers. That value can again be a name. For example, if we dereference `REF REAL`, the coercion can be represented by

$$\texttt{REF REAL} \Rightarrow \texttt{REAL}$$

As deproceduring, dereferencing can be applied repeatedly to produce for instance:

$$\texttt{REF REF REAL} \Rightarrow \texttt{REF REAL} \Rightarrow \texttt{REAL}$$

9.3.3 Dereferencing in a weak context

This is a variant of the dereferencing coercion in which any number of `REF`s can be removed except the last. This can be represented by

`REF REF REAL` ⇒ `REF REAL`

This coercion is only available in weak contexts and is needed in the primary of slices and the secondary of a selection to make sure that a slice can yield a name, or similarly a selection can yield a name — without this you would not be able to assign a value to an element of a row or to a field of a structure.

9.3.4 Dereferencing or deproceduring

In strong, firm, meek and weak contexts dereferencing and deproceduring will be applied alternatingly if this leads of coercion of the source-mode to the mode required by the context. For instance, in strong, firm and meek contexts we could have

`REF PROC REF REAL` ⇒ `PROC REF REAL` ⇒ `REF REAL` ⇒ `REAL`

though in a weak context we would have

`REF PROC REF [] REAL` ⇒ `PROC REF [] REAL` ⇒ `REF [] REAL`

9.3.5 Uniting

In this coercion, the mode of a value becomes a united-mode. For example, if `I` were an operator with both operands of mode `UNION(INT, REAL)`, then in the formula

```
0 I pi
```

both operands will be united to `UNION(INT, REAL)` before the operator is elaborated. These coercions can be represented by

$$\left.\begin{array}{l}\texttt{INT}\\ \texttt{REAL}\end{array}\right\} \Rightarrow \texttt{UNION(INT,REAL)}$$

Uniting is available in firm contexts and also in strong contexts where it precedes rowing.

9.3.6 Widening

In a strong context, an integer can be replaced by a real number and a real number replaced by a complex number, depending on the mode required. *a68g* implements next widenings:

1. `INT` $\Rightarrow$ `REAL`
2. `INT` $\Rightarrow$ `LONG INT`
3. `LONG INT` $\Rightarrow$ `LONG REAL`
4. `LONG INT` $\Rightarrow$ `LONG LONG INT`
5. `LONG LONG INT` $\Rightarrow$ `LONG LONG REAL`
6. `REAL` $\Rightarrow$ `COMPLEX`
7. `REAL` $\Rightarrow$ `LONG REAL`
8. `LONG REAL` $\Rightarrow$ `LONG COMPLEX`
9. `LONG REAL` $\Rightarrow$ `LONG LONG REAL`
10. `LONG LONG REAL` $\Rightarrow$ `LONG LONG COMPLEX`
11. `COMPLEX` $\Rightarrow$ `LONG COMPLEX`
12. `LONG COMPLEX` $\Rightarrow$ `LONG LONG COMPLEX`
13. `BITS` $\Rightarrow$ `LONG BITS`
14. `LONG BITS` $\Rightarrow$ `LONG LONG BITS`
15. `BITS` $\Rightarrow$ `[] BOOL`
16. `LONG BITS` $\Rightarrow$ `[] BOOL`
17. `LONG LONG BITS` $\Rightarrow$ `[] BOOL`
18. `BYTES` $\Rightarrow$ `[] CHAR`
19. `LONG BYTES` $\Rightarrow$ `[] CHAR`

Widening is not available in formulas since operands are in a firm context. Widening can be repeated, for example

`INT` $\Rightarrow$ `REAL` $\Rightarrow$ `LONG REAL` $\Rightarrow$ `LONG COMPLEX`

9.3.7 Rowing

If, in a strong context, a row is required and a value is provided whose mode is the mode of the elements of the row, then the value will be rowed. The resulting row (or added dimension) has one element and bounds $1 : 1$.

There are two cases to consider:

1. Suppose we want to row an element of mode `SOME` to a row `[] SOME`. The required rowed value is not a name. For example, if the required mode is `[] INT`, then the mode of an element is `INT`. In the identity-declaration

   ```
   [] INT i = 0
   ```

   the value yielded by the right-hand side (an integer) will be rowed and the coercion can be expressed as

   $$\texttt{INT} \Rightarrow \texttt{[] INT}$$

   If the value given is a row mode, such as `[] INT`, then there are two possible rowings that can occur:

   (a) `[] INT` ⇒ `[][] INT`, or

   (b) `[]INT` ⇒ `[, ]INT`
   (an extra dimension is added to the row)

2. If the row required is a name, then a name of an element can be rowed. For example, if the value supplied is a name with mode `REF SOME`, then a name with mode `REF [] SOME` will be created. Likewise, a name of mode `REF [] SOME` can be rowed to a name with mode `REF [, ] SOME` or a non-name with mode `[] REF [] SOME`, depending on the mode required.

9.3.8 Voiding

In a strong context, a value can be discarded, either because the mode `VOID` is explicitly stated, as in a procedure yielding `VOID` , or because the context demands it, as in the case of a semicolon (the go-on symbol).

Voiding can be applied to any value that is not a (variable whose value is a) parameter-less procedure. A parameter-less procedure is deprocedured and if it is not a `PROC VOID` the value after deproceduring is discarded. To avoid unexpected behaviour, deproceduring is not used to coerce an assignation, a generator or a cast to `VOID`. Hence if we consider

```
PROC p = VOID: ...;
PROC VOID pp;
pp := p;
...
```

then voiding the assignation `pp := p` does not involve deproceduring `p` after the assignation is completed.

9.4 Enclosed-clauses

There are seven types of enclosed-clause, most of which we have already met.

1. The simplest is the closed-clause which consists of a serial-clause enclosed in parentheses (or `BEGIN` and `END`). The range of any identifiers declared in the closed-clause is limited to that closed-clause.
2. Collateral-clauses are generally used as row-displays or stucture-displays: there must be at least two units. Remember that declarations are not units. The units are elaborated collaterally. This means that the order is undefined and may well be in parallel. Examples of collateral-clauses are:
3. A parallel clause looks exactly like a collateral-clause preceded by `PAR`. The constituent units (there must be at least two) are executed in parallel. A further section 9.5 discusses the *a68g* implementation of the parallel-clause.
4. The loop-clause is discussed in section 4.6.
5. The conditional-clause is discussed in section 4.2.
6. The integer-case-clause is discussed in section 4.5.
7. The united-clause-clause is discussed in section 7.3.

It should be noted that the enquiry-clause (in a conditional- or case-clause) is in a meek context whatever the context of the whole clause. Thus, the context of the clause is passed on only to the final unit in the `THEN`, `ELSE`, `IN` or `OUT` serial-clauses.

9.5 Parallel processing in *a68g*

Algol 68 supports parallel processing. Using the keyword `PAR`, a collateral-clause becomes a parallel-clause, in which the synchronisation of actions is controlled using semaphores. In *a68g* the parallel actions are mapped to POSIX threads when available on the hosting operating system (and *a68g* must be built to support the parallel-clause, see section 10.4.2).

Parallel units coordinate their actions by means of semaphores. A semaphore constitutes the classic method for restricting access to shared resources, such as shared stack and heap. It was invented by Edsger Dijkstra. Usually (and also in Algol 68) the term refers to a counting semaphore, since a binary semaphore is known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked-unlocked flag of a single resource. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem (a generic, abstract problem used for explaining issues with mutual exclusion), although they do not prevent resource deadlocks.

Semaphores remain in common use in programming languages that do not support other forms of synchronisation. Semaphores are the primitive synchronisation mechanism in many operating systems. The trend in programming language development is towards more structured forms of synchronisation, such as monitors and channels. In addition to their inadequacies in dealing with (multi-resource) deadlocks, semaphores do not protect the programmer from easy mistakes like taking a semaphore that is already held by the same process, or forgetting to release a semaphore.

A semaphore in Algol 68 is an object of mode `SEMA`:

```
MODE SEMA = STRUCT (REF INT F)
```

Note that the field cannot be directly selected. For a semaphore, next operators are defined:

1. `OP LEVEL = (INT a) SEMA`
   Yields a semaphore whose value is `a`.
2. `OP LEVEL = (SEMA a) INT`
   Yields the level of `a`, that is, field `F OF a`.
3. `OP DOWN = (SEMA a) VOID`
   The level of `a` is decremented. If it reaches 0, then the parallel unit that called this operator is hibernated until another parallel unit increments the level of `a` again.
4. `OP UP = (SEMA a) VOID`
   The level of `a` is incremented and all parallel units that were hibernated due to this semaphore being down are awakened.

Next classical example demonstrates the use of semaphores in Algol 68: producer-consumer-type parallel processes. In this simple example, production is incrementing an integer and consumption is decrementing it. Synchronisation is necessary to prevent both ”production” and ”consumption” from accessing the integer (the ”resource”) at the same time; hence the ”consuming” action has to wait for the ”producing” action to finish, and vice versa. This is done in line 2 by giving both actions a semaphore, the ”producing” semaphore at level 1 so production can start, and the ”consuming” semaphore at level 0 so it must initially wait for something being produced. Each parallel action starts by performing `DOWN` on its semaphore, than do its job, and then perform an `UP` on the semaphore of the other action.

```
1  BEGIN
2     INT n := 0, SEMA consume = LEVEL 0, produce = LEVEL 1;
3     PAR BEGIN # produce one #
4        DO DOWN produce;
5           print (n +:= 1);
6           UP consume
```

```
      OD,
      # consume one #
      DO DOWN consume;
         print (n -:= 1);
         UP produce
      OD
   END
END
```

The *a68g* parallel-clause deviates from the standard Algol 68 parallel-clause when parallel-clauses are nested. *a68g* parallel units behave like threads with private stacks. Hence if parallel units modify a shared variable then this variable must be declared outside the outermost parallel-clause, and a jump out of a parallel unit can only be targeted at a label outside the outermost parallel-clause.

9.6 Primaries

Primaries are denotations, applied-identifiers, casts, calls, format-texts, enclosed-closes and slices. Applied-identifiers are identifiers being used in a context, rather than in their declarations where they are defining-identifiers. We have met numerous examples of these. Routine-texts are not primaries, though format-texts are.

A cast consists of a mode indicant followed by an enclosed-clause. A cast puts its enclosed-clause in a strong context. It can be used to dereference exactly to the name required, as seen in paragraph 2.13. It can also be used to force the mode of a display when required:

```
z + COMPLEX (1, 0)
```

Calls were discussed in sections 6.4 and 6.5. In section 9.2, it was mentioned that the primary of a call is in a meek context. This applies even if the call itself is in a strong context. The primary of a call can be an enclosed-clause.

We discussed slices in section 5.6. They include simple subscripting. Whatever the context of the slice, the context of the primary of the slice is weak. This means that a final dereferencing is not performed.

There is another consequence of the weak context of the primary of a slice: collateral closes uses as row-displays can only be used in a strong context. So if you want to change the bounds of a row-display, the row-display must be enclosed in a cast. The context of units in subscripts and trimmers is meek.

Format-texts were discussed in section 8.10.

9.7 Secondaries

We have discussed both types of secondary (selections and generators), but there are other points which need mentioning.

Any primary can be regarded as a secondary, but not vice-versa.

Occasionally, when a procedure has a name parameter, the name may not be needed. Instead, therefore, of using an identifier of a name which is used for another purpose, which would be confusing, or declaring a name just for this purpose, which would be unnecessary, an anonymous name can be used. For example, a possible call of the procedure `char in string` could be

```
char in string(ch, LOC INT, str)
```

if you are only interested in whether the character is in the string and not in its position. A better strategy is to let your procedures accept `NIL` as a name, in which case it would not use the name.

When discussing selections in section 5.19, you may have wondered about the rules for placing parentheses when talking about rows in structures, rows of structures and rows in rows of structures. Firstly, it should be mentioned that in the secondary

```
im OF z
```

where z has mode `COMPLEX` or `REF COMPLEX`, the `z` itself is not only a secondary, it is also a primary — it is an applied-identifier. This means that in range of declarations

```
MODE PERPLEX = STRUCT (REAL x, COMPLEX z);
PERPLEX p
```

the selection

```
re OF z OF p
```

is valid because

```
z OF p
```

is itself a secondary.

Secondly, a primary is a secondary, but not necessarily the other way round. Consider these declarations:

```
MODE PERPLEX = ([3] COMPLEX z);
PERPLEX q
```

The selection `z OF q` has the mode `REF [] COMPLEX`. If you want to slice it, to get one of the constituent names of mode `REF COMPLEX` say, you cannot do so directly because in a slice, as mentioned in the previous section, what is sliced must be a primary. To convert the secondary into a primary you have to enclose it in parentheses thus converting it into an enclosed-clause; enclosed-clauses are primaries. So the first name of `z OF q` is yielded by `(z OF q)[1]`.

9.8 Tertiaries

Tertiaries are formulas, `NIL` and stowed-functions. Formulas were covered in chapter two. If a formula is to be used as a primary or a secondary, it must be an enclosed-clause. For example, in the formula `next OF (H declarer)`, the formula is a closed-clause which is a primary which is a secondary.

The only name having a denotation is `NIL`. `NIL` can have any mode which starts with `REF`.

Stowed-functions are described in section 5.16.

9.9 Units

Units are assignations, routine-texts, identity-relations (section 4.4), jumps (section 9.10), skips, assertions (section 9.11) or conditional-functions (section 4.3).

An assignation consists of three parts. The left-hand side must be a tertiary. Its value must be a name. Its context is soft, so no dereferencing is allowed unless a cast is used, but deproceduring is allowed. The second part is the assignation-token `:=` which is a rudimentary arrow. The right-hand side is a unit (including, of course, another assignation). Its context is strong so any coercion is permitted. The mode of its value must contain one less `REF` than the mode of the left-hand side.

The right-hand side of an assignation is a tertiary. If an assignation is to be used as a primary, a secondary or a tertiary, then it must be in an enclosed-clause. The value of an assignation is the value of the left-hand side: that is, it is a name. Assignations were discussed in section 2.10.

Routine-texts were discussed in chapter 6.

A skip is indicated by the keyword `SKIP`. This construct terminates and yields an undefined value of the mode required by the context, and can therefore only occur in a strong context. It is particularly useful in the following case. Consider the operator

```
OP INVERSE = (MATRIX m) MATRIX:
   IF DET m = 0
   THEN SKIP
```

```
  ELSE ...
  FI
```

If the determinant of a matrix is zero, no unique inverse exists and the operator yields an undefined matrix.

9.10 Jumps

Sometimes it is desirable to have more than one possible end-point of a serial-clause. This typically happens when a piece of code needs to abort. Like most other programming languages, Algol 68 allows to label units in serial-clauses. After a label-definition, you cannot write declarations.

A label is an identifier (unique in its range) followed by a colon. Declarations cannot be labelled. A jump to a label is invoked by writing the name of a label as a single primary. We have seen one jump already in this text:

```
stop
```

which effectively terminates execution since `stop` is a label defined at the last statements of the standard environment. Another example is the use of a jump in a transput event-routine:

```
on file end(standin, PROC (REF FILE f) BOOL: GO TO file empty)
```

To invoke a jump, you may write `GO TO` or `GOTO` in front of the identifier:

```
stop
GOTO stop
GO TO stop
```

Labels are not allowed in enquiry-clauses since you would be able to jump for instance from a then-part back into the condition. Declarations not allowed after a label-definition since you would for instance be able to jump back to before a declaration and execute it again, facilitating dangling names and scope errors. The rule is that any declaration in a serial-clause must be seen at most once before that serial-clause ends. This means that after a labelled-unit, you are free to use enclosed-clauses containing their private declarations since a jump out of a serial-clause effectively ends that serial-clause, and above rule is satisfied.

Jumps are like skips, they also yield an undefined object of the mode required by the context. But if the context expects a parameter-less procedure of mode `PROC VOID`, then a `PROC VOID` routine whose unit is that jump is yielded instead of making the jump:

```
#!/usr/local/bin/a68g
```

```

# Commented text taken from:
  C. H. Lindsey, A History of Algol 68,
  ACM Sigplan Notices, Volume 28, No. 3 March 1993 #

# ... But worse! Van Wijngaarden was now able to exhibit his pride and joy -
  his pseudo-switch [R8.2.7.2]. #

  [] PROC VOID switch = (e1, e2, e3);

# or even #

  LOC [1 : 3] PROC VOID switch var := (e1, e2, e3);
  switch var[2] := e3;

# To my shame, I must admit that this still works, although implementations
  tend not to support it. #

  switch var[2];

  print("should not be here");
  stop;

  e3: e2: e1: print ("jumped correctly")
```

Put above code in a file *jump.a68*, make it executable with *chmod* and execute the file to find

```
$ ./jump.a68
jumped correctly
```

which demonstrates that *a68g* implements proceduring to `PROC VOID`.

A completer, with keyword `EXIT`, provides a completion point for a serial-clause but cannot occur in an enquiry-clause. A completer can be placed wherever a semicolon (the go-on symbol `;`) can appear. A completer must be followed by a label-definition or the unit following the completer could not be reached. Here is an example of a completer in line 5:

```
1  (REAL x = read real;
2   IF i < 0
3   THEN GOTO negative
4   FI;
5   sqrt(i) EXIT
6   negative:
7   0
8  )
```

The example also illustrates why this is not a recommendable programming style. In fact, jumps have their use when it is required to jump out of nested clauses when something unexpected occurs from which recovery is not practical. Use of jumps should be confined to exceptions since they can make your programs illegible.

9.11 Assertions as pre- or postconditions

Algol 68 Genie supports an extension called assertions. Assertions can be viewed in two ways. First, they provide a notation for invariants that can be used to code a proof of correctness together with an algorithm. Hardly anyone does this, but assertions make for debugging statements. Assertion syntax reads:

```
assertion:
'ASSERT' meek-boolean-enclosed-clause.
```

Under control of the pragmat items 'assertions' and 'noassertions', the meek-boolean-enclosed-clause of an assertion is elaborated at runtime. If the meek-boolean-enclosed-clause yields `TRUE` execution continues but if it yields `FALSE`, a runtime error is produced. For example,

```
OP FACULTY = (INT n) INT:
   IF ASSERT (n >= 0);
      n > 0
   THEN n * FACULTY (n - 1)
   ELSE 1
   FI
```

will produce a runtime error when `FACULTY` is called with a negative argument.

9.12 Balancing

Enclosed-clauses such as conditional-clauses, integer-case-clauses and united-case-clauses yield one of a number of units, and it is quite possible for the units to yield values of different modes. The principle of balancing allows the context of all these units, except one, to be promoted to strong, whatever the context of the enclosed-clause. The one that is not promoted to strong remains in the imposed context of the clause, is of course the one mode to which the promoted units can be strongly coerced.

Balancing is also invoked for identity-relations which are dealt with in section 4.4.

Consider for example the formula

```
1 + (a > 0 | 1 | 0.0 )
```

where the context of the conditional-clause is firm hence widening is not allowed. Without balancing, the conditional-clause could yield a `REAL` or an `INT`. In this example, the principle of balancing would promote the context of the `INT` to strong and widen it to `REAL`. Balancing essentially means "making the modes the same".

In a balanced clause, one of the yielded units remains in the context of the clause and all the others are in a strong context, irrespective of the actual context of the clause.

9.13 A multi-pass scheme to parse Algol 68

As we have now seen all syntactic constructs of Algol 68, the end of this chapter is an appropriate place to explain, without going into all detail, how *a68g* executes a program. Algol 68 Genie employs a multi-pass scheme to parse Algol 68 [Lindsey 1993]:

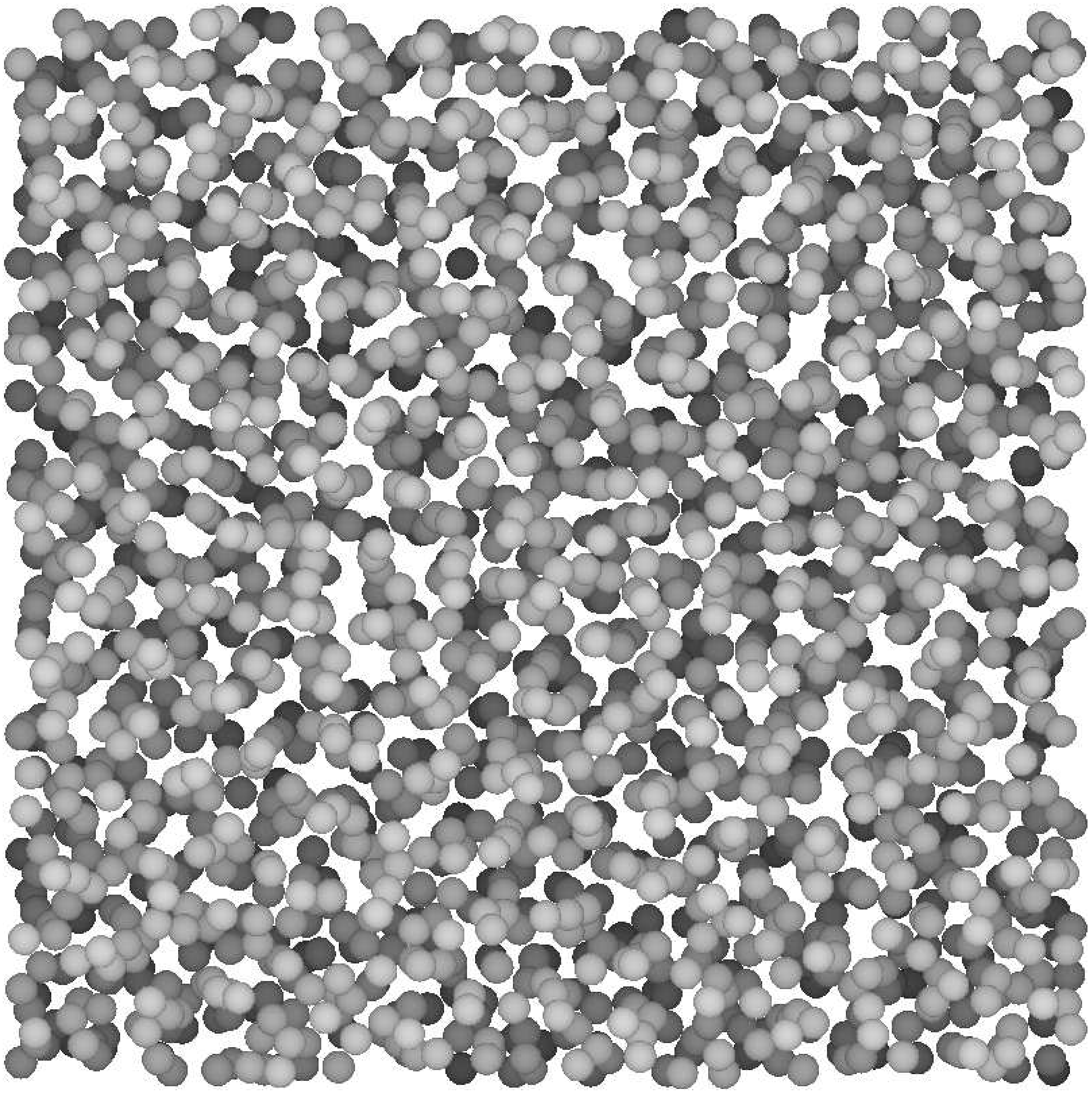
1. The *tokeniser*. The source file is tokenised, and if needed a refinement preprocessor elaborates a stepwise refined program. The result is a linear list of tokens that is input for the parser, that will transform the linear list into a syntax tree. *a68g* tokenises all symbols before the parser is invoked. This means that scanning does not use information from the parser. The scanner does some rudimentary parsing: format texts can have enclosed clauses in them, so information is recorded in a stack as to know what is being scanned. Also, the refinement preprocessor implements a (trivial) grammar.

2. The *parser*. First, parentheses are checked to see whether they match. Then a top-down parser determines the basic-block structure of the program so symbol tables can be set up that the bottom-up parser will consult as you can define things before they are applied. After that the bottom-up parser parses without knowing about modes while parsing and reducing. It can therefore not exchange `[ ... ]` with `( ... )` as is allowed by the Revised Report. This is solved by treating calls and slices as equivalent for the moment and letting the mode checker sort it out later. This is a Mailloux-type parser, in the sense that it scans a range for declarations — identifiers, operator tags and operator priorities — before it starts parsing, and thus allows for tags to be applied before they are declared.

3. The *mode checker*. The modes in the program are collected. Derived modes are calculated. Well-formedness is checked and structural equivalence is resolved. Then the modes of constructs are checked and coercions are inserted.

4. The *static-scope checker*. This pass checks whether you export names out of their scopes. Some cases cannot be detected by a static-scope checker, therefore the interpreter applies dynamic-scope checking.

5. The *interpreter*. The interpreter executes the syntax tree that results from the previous passes.



A Lennard-Jones $6 - 12$ *system near its triple point* $T^* = 0.72, \rho^* = 0.85$*, calculated and drawn using program 12.11. At the triple point, the three aggregation states coexist. Monte Carlo simulation is a valuable tool in studying thermodynamics of model systems. A classic review article is [Barker and Henderson 1976].*

Chapter 10

Using Algol 68 Genie

10.1 The Algol 68 Genie interpreter

This chapter describes the Algol 68 Genie interpreter *a68g*; how to install it on your computer system and how to use the program.

Algol 68 Genie (*a68g*) is open source software distributed under GNU GPL. This software is distributed in the hope that it will be useful, but **without any warranty**. Consult the GNU General Public License (*http://www.gnu.org/licenses/gpl.html*) for details. A copy of the license is in this manual.

An interpreter is a computer program that executes code written in a programming language. While interpretation and compilation are the two principal means for implementing programming languages, these are not fully distinct categories. An interpreter may be a program that either (1) executes source code directly, (2) translates source code into some intermediate representation which is executed or (3) executes stored code from a compiler which is part of the interpreter system. *a68g*, Perl, Python, MATLAB, and Ruby are examples of type (2), while UCSD Pascal and Java are type (3).

Algol 68 Genie is a nearly full implementation of Algol 68 as defined by [Revised Report 1976] and also implements partial parametrisation [Lindsey AB 39.3.1], which is an extension of Algol 68.

After successful parsing of an entire source program, the syntax tree, that serves as an intermediate program representation, is interpreted. The interpreter performs many runtime checks, therefore *a68g* resembles *FLACC* [Mailloux 1978]. Algol 68 Genie employs a multipass scheme to parse Algol 68 [Lindsey 1993].

10.2 Algol 68 Genie compared to the Revised Report

10.2.1 Features of *a68g*

The interpreter checks on many events, for example:

1. Assigning to, or dereferencing of, `NIL`.
2. Using uninitialised values.
3. Invalid operands to standard-prelude operators and procedures.
4. Bounds check when manipulating rows.
5. Overflow of arithmetic modes.
6. ”Dangling references”, that are names that refer to deallocated storage.

Precision of numeric modes (see sections 2.2, 2.4, 5.22 and 5.27):

1. Implementation of `LONG INT`, `LONG REAL`, `LONG COMPLEX` and `LONG BITS` with roughly doubled precision with respect to `INT`, `REAL`, `COMPLEX` and `BITS`.
2. Implementation of multi-precision arithmetic through `LONG LONG INT`, `LONG LONG REAL`, `LONG LONG COMPLEX` and `LONG LONG BITS` which are modes with user defined precision which is set by an option.

Further features:

1. On systems that support them, UNIX extensions that allow e.g. for executing child processes that communicate through pipes, matching regular expressions or fetching web page contents, see section 11.16.
2. Procedures for drawing using the GNU Plotting Utilities, see section 11.16.
3. Various extra numerical procedures, many of which from the GNU Scientific Library, see section 11.11.
4. Basic linear algebra and Fourier transform procedures from the GNU Scientific Library, see sections 11.12, 11.13 and 11.14.
5. Support for WAVE/PCM sound format, see section 11.20.
6. Support for PostgreSQL, an open-source relational database management system, enabling client applications in Algol 68, see section 11.19.

7. Format texts, straightening and formatted transput. Transput routines work generically on files, (dynamic) strings and UNIX pipes. See section 8.10.
8. Parallel-clause on platforms that support POSIX threads. See section 9.5.
9. Upper stropping is the default, quote stropping is optional.

10.2.2 Extra features compared to the Revised Report language

1. Implementation of C.H. Lindsey's partial parametrisation proposal, giving Algol 68 a functional sub language [Koster 1996]. See section 6.19.
2. A simple refinement preprocessor to facilitate top-down program construction. See section 10.9.3.
3. Implementation of pseudo-operators `ANDF` and `ORF` (or their respective alternatives `THEF`, `ANDTH` and `OREL`, `ELSF`. See section 4.3.
4. Implementation of pseudo-operators `TRNSP`, `DIAG`, `COL` and `ROW` as described by [Torrix 1977]. See section 5.16.
5. Implementation of `DOWNTO` with comparable function as `TO` in loop-clauses; `DOWNTO` decreases, whereas `TO` increases, the loop counter by the value of the (implicit) by-part. See section 4.6.
6. Implementation of a post-checked loop. A do-part may enclose a serial-clause followed by an optional until-part, or just enclose an until-part. This is an alternative to the paradigm Algol 68 post-check loop `WHILE ... DO SKIP OD`. An until-part consists of the keyword `UNTIL` followed by a meek-boolean-enquiry-clause. The loop-clause terminates when the enquiry-clause yields `TRUE`. See section 4.6.
7. Implementation of monadic- and dyadic operator `ELEMS` that operate on any row. The monadic operator returns the total number of elements while the dyadic operator returns the number of elements in the specified dimension, if this is a valid dimension. See section 5.5.
8. When option `brackets` is specified, `(...)`, `[...]` and `{...}` are equivalent to the parser and any pair can be used where Algol 68 requires open-symbols and close-symbols. This allows for clearer coding when parenthesis are nested. If `brackets` is not specified, `(...)` is an alternative for `[...]` in bounds and indexers, which is traditional Algol 68 syntax.
9. Implementation of operators `SET` and `CLEAR` for mode `BITS`.
10. The parser allows for colons, used in bounds and indexers, to be replaced by `..` which is the Pascal style.

10.2.3 Deviations from the Revised Report language

1. The important difference with the Revised Report transput model is that Algol 68 Genie transput does not operate on `FLEX [] FLEX [] FLEX [] CHAR`, but on `FLEX [] CHAR`. This maps better onto operating systems as UNIX or Linux.
2. The Algol 68 Genie parallel-clause deviates from the standard Algol 68 parallel-clause when parallel-clauses are nested. Algol 68 Genie's parallel units behave like threads with private stacks. Hence if parallel units modify a shared variable then this variable must be declared outside the outermost parallel-clause, and a jump out of a parallel unit can only be targeted at a label outside the outermost parallel-clause.
3. The interpreter does not implement so-called ghost-elements [RR 2.1.3.4], hence it cannot check bounds when assigning to rows of mode flexible-rows-of-rows-of-... when the destination has a flat descriptor (i.e. has zero elements). A mode flexible-row-of-...-row-of-... is treated as if it were flexible throughout. The scope checker recognises the potential introduction of transient references resulting from this peculiarity in *a68g*.
4. Algol 68 Genie does not recognise nonlocal environs [RR 5.2.3.2]. All environs are local. Hence constructs as

   ```
   REF INT i = (... | LOC INT | SKIP);
   ```

   or

   ```
   [n] REF INT a; FOR k TO n DO a[k] := LOC INT OD
   ```

   result in a scope violation error with Algol 68 Genie.
5. It is not possible to declare in a `[WHILE ...] DO ... [UNTIL ...] OD` part an identifier with equal spelling as the loop counter identifier.
6. If the context of a jump expects a parameterless procedure of mode `PROC VOID`, then a `PROC VOID` routine whose unit is that jump is yielded instead of making the jump. In standard Algol 68, this proceduring will take place if the context expects a parameterless procedure, while Algol 68 Genie limits this to `PROC VOID`.
7. Algol 68 Genie maps a declarer whose length is not implemented onto the most appropriate length available [RR 2.1.3.1]. Algol 68 Genie considers mapped modes equivalent to the modes they are mapped onto, while standard Algol 68 would still set them apart. Routines or operators for not-implemented lengths are mapped accordingly.

10.3 *a68g* transput versus standard Algol 68

a68g transput deviates from the Revised Report specification, as described below. For an overview of implemented procedures refer to the standard-prelude reference.

10.3.1 Features of *a68g*

1. Transput procedures operate generically on files, strings and Linux pipes.
2. Implementation of a procedure `real` that extends the functionality of `float`.
3. *a68g* implements *ALGOL68C* routines as `read int` and `print int`, but not routines as `get int` and `put int`.
4. There are two extra procedures to examine a file: `idf` and `term`.
5. If a file does not exist upon calling open, the default action will be to create it on the file system.
6. If a file exists upon calling establish, the event handler set by on open error will be invoked.
7. *a68g* can write to file stand error with associated channel stand error channel. This file is linked to the standard error stream that is usually directed at the console.
8. Insertions can be any combination of alignments and literals.
9. The argument for a general-pattern is an enclosed-clause yielding `[] INT` (which is a superset of the Revised Report specification.
10. Extended general-pattern `h` for transputting real values.
11. The radix for a bits-pattern can be $2 \ldots 16$. The radix can be dynamic.
12. The Revised Report specification of getting using an integral choice pattern has the peculiarity that when two literals start with the same sequence of characters, the longer literal should appear first in the list. *a68g* makes no such demand, and will select the correct literal from the list whatever their order.

10.3.2 Deviations of standard Algol 68

1. The important difference with the Revised Report transput model is that *a68g* transput does not operate on a `FLEX [] FLEX [] FLEX [] CHAR`, but on a `FLEX [] CHAR`. This maps better onto operating systems as Linux.
2. `establish` does not take arguments specifying the size of a file.
3. Getting and putting a file is essentially sequential. Only `reset` can intervene with sequential processing.
4. *a68g* does not currently permit switching between read mood and write mood unless the file is `reset` first (and the file was opened with `standback channel`). Whether a file is in read mood or write mood is determined by the first actual transput operation (`put` or `get`) on a file after opening it.

5. Since *a68g* transput operates on a `FLEX [] CHAR`, routine `associate` is associates a file to a `REF STRING` object. On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition, and `on file end` is invoked. When a file that is associated with a string is reset, getting restarts from the start of the associated string.

6. Since an end on file event handler cannot move the file pointer to a good position (reset also resets a file's read mood and write mood), encountering end of file terminates getting of a `STRING` value. Conform the Revised Report, getting of a `STRING` value will resume if the end of line event handler or the end of page handler returns `TRUE`.

7. There is no event routine `on char error mended`. Attempted conversion of an invalid value for a required mode, or attempted transput of a value that cannot be converted by the current format-pattern, evokes the event routine set by `on value error`.

8. When all arguments in a call of `readf`, `printf`, `writef`, `getf` or `putf` are processed, the format associated with the corresponding file is purged - that is, remaining insertions are processed and the format is discarded. If a pattern is encountered while purging, then there was no associated argument and the event routine set by on format error is called. When this event routine returns `FALSE` (the default routine always returns `FALSE`) a runtime error will be produced.

9. `SIMPLIN` and `SIMPLOUT` are generic for both formatted and unformatted transput. Therefore `SIMPLIN` and `SIMPLOUT` include both mode `FORMAT` and `PROC (REF FILE) VOID`. If a transput procedure encounters a value whose transput is not defined by that procedure, a runtime error occurs.

10. Insertion `x` has the same effect as insertion `q`; both call space.

11. When a sign is shifted in a sign-mould, any character output by literal insertions in that sign-mould is replaced with a space as well, starting from the first z-frame until the sign is put.

12. When getting a literal insertion, `space` is performed for every character in that literal. It is not checked whether read characters actually match the literal.

13. Routine `set` moves with respect to the current file pointer.

10.4 Installing Algol 68 Genie on GNU/Linux

First of all, you will need to download the latest version of Algol 68 Genie from

http://www.xs4all.nl/~jmvdveer/algol.html

In this chapter we assume that the latest version is packed in a file called

```
algol68g-mk14.1.tar.gz,
```

which is a gzipped tar-archive.

10.4.1 Brief installation guide

If you are a reasonably experienced Linux user, the following list of commands may suffice to help you on your way. If you do not feel confident to follow this short list, follow the detailed instructions in next section. Enter:

```
tar -xzf algol68g-mk14.1.tar.gz
cd algol68g-mk14.1
./configure -O2 --threads
make
make install     # if you are superuser, that is, "root" #
```

10.4.2 Detailed installation guide

This section describes into detail how to install *a68g* on Linux/Mac OS X.

Optional libraries

If available on your system, *a68g* can use next libraries to extend its functionality:

1. GNU plotutils for drawing from Algol 68, see
   http://www.gnu.org/software/plotutils/
2. GNU scientific library to extend the mathematical capabilities of the standard environ, see
   http://www.gnu.org/software/gsl/
3. PostgreSQL for writing PostgreSQL database client applications, see
   http://www.postgresql.org/

The configure script checks

```
/usr/lib
/usr/local/lib
/usr/include
/usr/local/include
```

for shared libraries and necessary includes. If necessary files are not detected then *a68g* builds without support for absent libraries.

Generic installation

For a generic installation, follow next instructions.

Step 1: Unpacking the distribution

Unpack the distribution by typing:

```
tar -xzf algol68g-mk14.1.tar.gz
```

Step 2: Configuring

Make the distribution directory the present working directory by typing:

```
cd algol68g-mk14.1
```

In the distribution directory type:

```
./configure
```

This will generate a makefile, which will be used by `make` to build *a68g*.

The configure script accepts various options. For an overview of options, type:

```
./configure --help
```

Note that various options for optimisation, debugging and profiling assume you are using *gcc*, *gdb* and *gprof*.

If you are interested in *a68g*'s parallel-clause, you need to build using POSIX threads. To build with POSIX threads, type:

```
./configure --threads
```

Step 3: Building The standard makefile will turn warnings off in case *gcc* is used. If you do not use *gcc*, or make *gcc* issue warnings, it is possible that your C compiler will issue spurious warnings suggesting that some variables might be clobbered by `longjmp` or `vfork`. For some reason data path analysis in such compiler cannot see that the code is safe since these variables are declared `volatile`. There is no risk of clobbering, and therefore such warning messages can be safely ignored.

If you are not the superuser or you do not want to install *a68g* for all users, type:

```
make
```

The executable *a68g* will be put in the present working directory, `algol68g-mk14.1`.

If you want to install *a68g* for all users on your system, type:

```
make install
```

to make available the executable and a manual page. Targets for those files are `/usr/local/bin` and `/usr/local/man`. If latter directory does not exist, `/usr/share/man` and then `/usr/man` are tried.

Installation can be undone by typing:

```
make uninstall
```

Note that in you can supply `CPPFLAGS`, `CFLAGS` and `LDFLAGS` to `make`; for instance type:

```
make CFLAGS="-O2"
```

to optimise the code so it will run faster.

To remove binaries used for building, thus recovering some disk space, type:

```
make clean
```

To remove all files used for building, type:

```
make distclean
```

Step 4: Starting Algol 68 Genie

When you install *a68g* in the present working directory, you may want to include the current directory in your `PATH` variable. Otherwise you will need to indicate the current directory by typing `./a68g` as in the following example:

```
$ ./a68g --version
Algol 68 Genie Mark 14.1 (released November 2008), copyright 2001-2008 J. Marcel van der Veer.
Algol 68 Genie is free software covered by the GNU General Public License.
There is ABSOLUTELY NO WARRANTY for Algol 68 Genie.
See the GNU General Public License for more details.

Compiled on Linux 2.6.26 i686 with gcc 4.3.2
Configured on 1 November 2008 01:00 with options "-O2 --threads"
GNU libplot 4.2
GNU Scientific Library 1.11
PostgreSQL libpq 8.3.3
Alignment 4 bytes
Default frame stack size: 3072 kB
Default expression stack size: 1024 kB
Default heap size: 24576 kB
Default handle pool size: 4096 kB
Default stack overhead: 512 kB
```

```
Effective system stack size: 8192 kB
```

If instead of above version statement you get a message like

```
./a68g: error in loading shared libraries
libgsl.so.0: cannot open shared object file:
No such file or directory
```

or

```
psql: error in loading shared libraries
libpq.so.4.1: cannot open shared object file:
No such file or directory
```

then your shared library search path was not properly set for the GNU Scientific Library or PostgreSQL's `libpq` library, respectively. The method to set the shared library search path varies between platforms, but the most widely used method is to set the environment variable `LD_LIBRARY_PATH`. In Bourne shells (`sh`, `ksh`, `bash`, `zsh`) you would use:

```
LD_LIBRARY_PATH=/usr/local/lib:/usr/local/pgsql/lib:\$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

while in `csh` or `tcsh` you would use:

```
setenv LD\_LIBRARY\_PATH /usr/local/lib:/usr/local/pgsql/lib:\$LD\_LIBRARY\_PATH
```

You should put these commands into a shell start-up file such as

`/etc/profile`

or

`~/.bash_profile`.

When in doubt, refer to the manual pages of your system.

10.5 Synopsis

To start *a68g* from the command-line you would use:

```
a68g [options | filename]
```

If *a68g* cannot open `filename`, it will try opening with extensions `.a68`, `.a68g`, `.algol68` or `.algol68g` respectively. Alternatively, under Linux an Algol 68 source file can be made a script by entering

```
#! [/usr/local/bin/]a68g [options]
```

as the *first* line in the source file. After making the script executable by typing

```
chmod +x filename
```

the script can be started from the command line by typing:

```
filename
```

On Linux, Algol 68 Genie recognises next environment variables (if they are undefined, *a68g* assumes default values for them):

`A68G_STANDIN`
The value will be the name of the file to which `stand in` will be redirected.

`A68G_STANDOUT`
The value will be the name of the file to which `stand out` will be redirected.

`A68G_STANDERROR`
The value will be the name of the file to which `stand error` will be redirected.

`A68G_OPTIONS`
The value will be tokenised and processed as options. These options will supersede options set in file `.a68grc` (vide infra).

10.6 Diagnostics

This section discusses how *a68g* specifies its diagnostics. The interpreter checks for many events and since some diagnostics are synthesised it is not possible to provide a finite list of *a68g*'s diagnostics. Diagnostics would typically be presented to you like this:

```
7            REF [] INT q2 = p[3,];
                       1    2
a68g: warning: 1: tag "q2" is not used (detected in VOID closed-clause
starting at "BEGIN" in line 1).
a68g: error: 2: attempt at storing a transient name (detected in VOID
closed-clause starting at "BEGIN" in line 1).
```

If possible, *a68g* writes the offending line and indicates by means of single-digit markers the positions where diagnostics have been issued. Note that the source of the error will often only be *near* that marker. Then a list follows of diagnostics issued for that line. As is usual in the GNU/Linux world, the first element of a diagnostic is the name of the program giving the diagnostic, in casu *a68g*. If the error occurs in an included source file, the included source file's name will be the next element. If the source is contained in a single file than the source file name is not stated. Then follows the marker that indicates the position in the line where the diagnostic was issued. Finally a descriptive message is written.

In case a diagnostic is not related to a specific source line, the diagnostic looks simpler since a source line and markers can be omitted, for instance:

```
$ a68g --precision=1k --heap=256G examples/mc.a68
a68g: error in option "--heap=256g" (numerical result out of range).
```

which on a 32-bit platform suggests you probably wanted to specify `--heap=256M`.

Diagnostics come in different levels of severity. A *warning* will not impede execution of a program, but it will draw your attention to some issue; for instance:

```
106          ABS diff < 100 ANDF exp (- diff) > random
             1
a68g: warning: 1: construct is an extension.
```

indicates that this statement contains an extension (here, `ANDF`) so this program may not be portable.

An *error* impedes successful compilation and execution of a program. Some condition was encountered that you must fix before execution could take place. For instance:

```
2     IN (UNION (INT, BOOL)): SKIP,
         1
a68g: error: 1: UNION (BOOL, INT) is neither component nor subset of
UNION (CHAR, BOOL) (detected in united-case-clause starting at "CASE"
in line 1).
```

or

```
38        matvec (1, 10, bool, ca, aa);
          1
a68g: error: 1: REF BOOL cannot be coerced to INT in a
strong-argument (detected in closed-clause starting at "("
in line 3).
```

obviously are errors that need fixing before successful compilation and execution.

A *syntax error* is given when an error has been made in syntax according to chapter B. Such diagnostic is typically issued by the parser. In case of a syntax error, *a68g* tries to give additional information in an attempt to help you localising and diagnosing the exact error. For instance, in

```
103           CASE v
              1
a68g: syntax error: 1: incorrect parenthesis nesting; encountered
end-symbol in line 152 but expected esac-symbol; check for "END"
without matching "BEGIN" and "CASE" without matching "ESAC".
```

a case-clause was terminated with `END` so *a68g* straightforwardly suggests to check for a missing `BEGIN` or a missing or misspelled `ESAC`. In next example

```
317   OP * = (VEC,v  REAL r) VEC: r * v;
              1
a68g: syntax error: 1: expected parameter-pack, which is not
consistent with the construct starting with "(" in line 317
followed by a declarer and then ",", "v", a parameter-list, ")".
```

the parser explains that it wanted to parse a parameter-pack but it could not complete that since a declarer was followed by a comma-symbol. Clearly `VEC,r` should have been written `VEC r,`. These synthetic explanations can get quite verbose, as in

```
385   proc set pixel = (ICON i, PT p, COLOR c ) VOID:
                     1
a68g: syntax error: 1: expected serial-clause, which is not
consistent with the construct starting with a serial-clause
starting in line 5 followed by "=" in line 385 and then a
routine-text, ";" in line 386, "procsetpixelrgb" in line 387,
"=", a routine-text, ";" in line 388 etcetera.
```

which indicates that things looked all-right up to line 385, but then a serial-clause was equated to a routine-text which is odd. You will be quick to note that a missing `PROC` symbol was incorrectly spelled in lower-case, and the same will have happened in line 387.

When your program is being executed, a *runtime error* can occur indicating that some condition arose that made continuation of execution either impossible or undesirable. A typical example would be

```
8         CASE CASE n
               1
a68g: runtime error: 1: REF [] INT value from united value is exported
out of its scope (detected in VOID united-case-clause starting at
 "CASE" in this line).
```

but if any information can be obtained from runtime support about *what* went wrong, it will be included in the diagnostic as in

```
41          get (standin, (x, space, y, space, number, new line));
            1
a68g: runtime error: 1: cannot open ".txt" for getting (no such
file or directory) (detected in VOID loop-clause starting at "TO"
in line 38).
```

which means that the file name that you specified with `open` does not exist now that you decided to read from it — it indeed seems you only wrote an extension. Note that you can intercept a runtime error by specifying option `-monitor` or `--debug` by which the monitor will be entered when a runtime error occurs, so you can inspect what went wrong; see section 10.8.

10.7 Options

Options are passed to *a68g* either from the file `.a68grc` in the working directory, the

```
A68G_OPTIONS
```

environment variable, the command-line or through pragmats. Precedence is as follows: pragmats supersede command-line options, command-line options supersede options from

```
A68G_OPTIONS
```

that supersede options in

```
.a68grc
```

Options have syntax `-[-]option [[=] value[suffix]]`

Option names are not case sensitive, but option arguments are. In the list below, uppercase letters denote letters mandatory for recognition. Note that option syntax is a superset of Linux standards since *a68g* was initially written on other operating systems than Linux and had to maintain a degree of compatibility with option syntax of all of them.

Listing options, tracing options and `pragmat`, `-nopragmat`, take their effect when they are encountered in a left-to-right pass of the program text, and can thus be used to generate a cross reference for a particular part of the user program.

In integral arguments, suffix k, M or G is accepted to specify multiplication by 2^{10}, 2^{20} or 2^{30} respectively. The suffix is case-insensitive.

10.7.1 One-liners

• `Print unit`

Prints the value yielded by Algol 68 unit `unit`. In this way *a68g* can execute one-liners from the command-line. The one-liner is written in file `.a68g.x` in the working directory.

Example: `a68g -p "sqrt (2 * pi)"`

• `Execute unit`

Executes Algol 68 unit `unit`. In this way *a68g* can execute one-liners from the command-line. The one-liner is written in file `.a68g.x` in the working directory.

Example: `a68g -e "printf (($lh$, 4 * atan (-1)))"`

10.7.2 Memory size

Algol 68 relieves the programmer from managing memory, hence *a68g* manages allocation and de-allocation of heap space. However, if memory size were not bounded, *a68g* might claim all available memory before the garbage collector is called. Since this could impede overall system performance, memory size is bounded.

• `HEAP number`

Sets heap size to `number` bytes. This is the size of the block that will actually hold data, not handles. At runtime *a68g* will use the heap to store temporary arrays, so even a program that has no heap generators requires heap space, and can invoke the garbage collector.

Example: `PR heap=32M PR`

• `HANDLES number`

Sets handle space size to `number` bytes. This is the size of the block that holds handles that point to data in the heap. A reference to the heap does not point to the heap, but to a handle as to make it possible for the garbage collector to compact the heap.

Example: `PR handles=2M PR`

• `FRAME number`

Sets frame stack size to `number` bytes. A deeply recursive program with many local variables may require a large frame stack space.

Example: `PR frame=1M PR`

• `STACK number`

Sets expression stack size to `number` bytes. A deeply recursive program may require a large expression stack space.

Example: `PR stack=512k PR`

• `OVERHEAD number`

Sets the overhead, which is a safety margin, for the expression stack and the frame stack to `number` bytes. Since stacks grow by relatively small amounts at a time, the interpreter checks stack sizes only where recursion may set in. It is checked whether stacks have grown into the overhead. For example, if the frame stack size is 512 kB and the overhead is set to 128 kB, the interpreter will signal an imminent stack overflow when the frame stack pointer exceeds $512 - 128 = 384$ kB.

When the overhead is set to a too small value, a segment violation may occur.

Example: `PR overhead=64k PR`

10.7.3 Listing options

• `EXTensive`

Generates an extensive listing, including source listing, syntax tree, cross reference et cetera. This listing can be quite bulky.

• `LISTing`

Generates concise listing.

• `MOIDS`

Generates overview of moids in listing file.

• `PRELUDElisting`

Generates a listing of preludes.

• `SOURCE, NOSOURCE`

Switches listing of source lines in listing file.

• `STatistics`

Generates statistics in listing file.

• `TREE, NOTREE`

Generates syntax tree listing in listing file. This option can make the listing file bulky, so use considerately.

• `UNUSED`

Generates an overview of unused tags in the listing file.

• `Xref, NOXref`

Switches generating a cross reference in the listing file.

10.7.4 Interpreter options

• `MONitor, DEBUG`

Start the program in the monitor. You will have to start program execution manually. Also, upon encountering a runtime error, instead of terminating execution, the monitor will be entered.

• `Assertions, NOAssertions`

Switches elaboration of assertions.

• `BACKtrace, NOBACKtrace`

Switches stack backtracing in case of a runtime error.

• `BReakpoint, NOBReakpoint`

Switches setting of breakpoints on following program lines.

• `TRace, NOTRace`

Switches tracing of a program.

• `PRECision number`

Sets precision for `LONG LONG` modes to `number` significant digits. The precision cannot be less than the precision of `LONG` modes.

Algorithms for extended precision in *a68g* are not really suited for precisions larger than about a thousand digits. State of the art in the field offers more efficient algorithms than implemented here.

• `TImelimit number`

Interrupts the interpreter after `number` seconds; a *time limit exceeded* runtime error will be given. This can be useful to detect endless loops.

Trivial example:

```
$ a68g -ti=1 -e "DO SKIP OD"
1     (DO SKIP OD)
       1
a68g: runtime error: 1: time limit exceeded (detected in
VOID loop-clause starting at "DO" in this line).
```

10.7.5 Miscellaneous options

• `APropos, Help, INfo [ string ]`

Prints info on options if `string` is omitted, or prints info on `string` otherwise. This option can also give brief information on syntactic constructs:

• `UPPERstropping, QUOTEstropping`

Sets the stropping regime. Upper stropping is the default. Quote stropping is implemented to let *a68g* scan vintage source code.

• `BRackets`

Enables non-traditional use of brackets. This option makes `{..}` and `[..]` equivalent to `(..)` for the parser.

• `CHECK, NORUN`

Check syntax only, interpreter does not start.

• `RUN`

Override `NORUN`.

• `EXIT, --`

Ignore further options from command line or current pragmat. This option is usually provided under Linux to allow source file names to have a name beginning with a minus-sign.

• `File string`

Accept argument `string` as generic filename.

• `PEDANTIC`

Equivalent to `WARNINGS PORTCHECK`.

• `PORTcheck, NOPORTcheck`

Enable or suppress portability warning messages.

• `WARNINGS, NOWarnings`

Enable warning messages or suppresses suppressible warning messages.

• `PRagmats, NOPRagmats`

Switches elaboration of pragmat items. When disabled, pragmat items are ignored, except for option `PRAGMATS`.

• `REDuctions`

Prints reductions made by the parser. This option was originally meant for parser debugging but can be quite instructive (and very verbose).

• `VERBose`

Informs on actions.

• `VERSion`

States the version of the running copy of *a68g*.

• `Warnings`

Enables warning messages.

10.8 The monitor

The interpreter has a monitor for basic debugging of a running program. The monitor is invoked when *a68g* receives signal `SIGINT` (which usually comes from typing `CTRL-C`), when standard procedures `debug` or `break` are called, or when a runtime error occurs while option `monitor` or `debug` is in effect. Breakpoints can be indicated a priori through pragmats as well.

Following shows an example where the monitor is invoked by typing `CTRL-C`:

```
$ a68g buggy
^C
121   WHILE n ~= 0 DO
            -
Terminate a68g (yes|no): n
This is the main thread
(a68g)
```

The prompt `(a68g)` tells you that the monitor is awaiting a command. Next section describes the commands that the monitor can process.

10.8.1 Monitor commands

As with options, monitor command names are not case sensitive, but arguments are. In the list below, uppercase letters denote letters mandatory for recognition. Currently, the monitor recognises next commands:

• `APropos, Help, INfo [ string ]`

Prints info on monitor commands if `string` is omitted, or prints info on `string` otherwise.

• `Breakpoint n [ IF expression ]`

Set breakpoints on units in line `n`. If you supply an expression then interruption will only take place if this expression yields `TRUE`. If the expression is not correct, it will be deleted. The interpreter only breaks on units that are statements: in serial-, collateral- and enquiry-clauses, in unit-lists in in-parts of integer- or united-case-clauses, and sources in declarations. Note that a breakpoint expression is a nice debugging tool, but it slows down execution a lot.

• `Breakpoint Watch [ expression ]`

Sets the watchpoint expression. Interruption will take place whenever the expression yields `TRUE`. The interpreter only breaks on units that are statements: in serial-, collateral- and enquiry-clauses, in unit-lists in in-parts of integer- or united-case-clauses, and sources in declarations. If the expression is not correct, it will be deleted. Note that a watchpoint expression is a nice debugging tool, but it slows down execution a lot.

• `Breakpoint n Clear`

Clears breakpoints on units in line `n`.

• `Breakpoint [ List ]`

Lists all breakpoints and the watchpoint expression.

• `Breakpoint Clear [ ALL ]`

Clears all breakpoints and the watchpoint expression.

• `Breakpoint Clear Breakpoints`

Clears all breakpoints.

• `Breakpoint Clear Watchpoint`

Clears the watchpoint expression.

• `BT n`

Print `n` frames in the stack following the dynamic link (BackTrace) (default `n = 3`).

• `CAlls n`

Print `n` frames in the call stack (default `n = 3`).

• `Continue, Resume`

Continue execution.

• `DO command, EXEC command`

Pass `command` to the shell and print the return code.

• `Evaluate expression, X expression`

Evaluate `expression` and show its result. For a description of monitor expressions see the next section.

• `EXamine n`

Print value of all identifiers named `n` in the call stack.

• `EXIt, HX, Quit`

Terminates *a68g*.

• `Frame n`

Select stack frame `n` as the current stack frame. If `n` is not specified, print the current stack frame. If 0 is specified, the current stack frame will be the top of the frame stack.

• `HEAp n`

Print contents of the heap with address not greater than `n`.

• `HT`

Halts typing to standard output.

• `LINk n`

Print `n` frames in the stack, following the static link (default `n = 3`). This will give generally give a shorter backtrace than `stack` since the static link links to the frame embedding (a) the actual lexical level or (b) the incarnations of the active procedure, while the dynamic link just points at the previous lexical level (and thus walks through all incarnations of a recursive procedure and all ranges of clauses).

• `RERun, REStart`

Restarts the Algol 68 program without resetting breakpoints.

• `RESET`

Restarts the Algol 68 program and resets breakpoints.

• `RT`

Resumes typing to standard output.

• `List n m`

If `m` is omitted, show `n` lines around the interrupted line (default `n = 10`). If `n` and `m` are supplied, show lines `n` up to `m`.

• `Next`

Resume execution until the next unit (that can be a breakpoint) is reached; do not enter routine-texts.

• `STEp`

Resume execution until the next unit (that can be a breakpoint) is reached.

• `FINish, OUT`

Resume execution until the next unit (that can be a breakpoint) is reached, *after* the current procedure incarnation will have finished. You typically use this to leave a procedure and to stop in the caller.

• `Until n`

Resume execution until the first unit (that can be a breakpoint) is reached, on line number `n`.

• `PROmpt n`

Set prompt to `n`. Default prompt is `(a68g)`.

• `Sizes`

Show sizes of various memory segments.

• `STAck n`

Print `n` frames in the stack following the dynamic link (default `n = 3`).

• `Where`

Show the line where interruption took place.

• `XRef n`

Give detailed information on source line `n`.

10.8.2 Monitor expressions

Monitor expressions provide basic means to do arithmetic in the monitor, and to change stored values. Monitor expression syntax is similar to Algol 68 syntax. Important points are:

1. Expressions, assignation sources and assignation destinations are strictly evaluated from left to right.

2. Actual assignation is done from right to left.

   ```
   (a68g) x a[1] := 0
   (REF INT) refers to heap(892000)
   (INT) +0
   (a68g) x a[2] := a[1] := 1
   (REF INT) refers to heap(892016)
   (INT) +1
   ```

   Note that the result of the evaluation of an expression is preceded by the mode of that result.

3. Only procedures and operators from standard environ can be called, and operator priorities are taken from standard environ.

4. Operands are denoters, identifiers, closed-clauses, calls and slices. Casts are also operands but can only be made to mode `[LONG][LONG] REAL` or to a (multiple) reference to an indicant

   ```
   (a68g) x REAL (1)
   (REAL) +1.00000000000000E  +0
   (a68g) x REF TREE (k) IS NIL
   (BOOL) F
   (a68g) x REF TREE (k) := NIL
   (REF REF STRUCT (INT f, REF SELF next)) refers to heap(600000)
   (REF STRUCT (INT f, REF SELF next)) NIL
   ```

5. Slicing does not support trimmers. Only `[` and `]` are allowed as brackets for indexers.

6. Selections are allowed, but not multiple selections.

   ```
   (a68g) x im OF z
   (REF LONG REAL) refers to frame(80)
   (LONG REAL) +3.1415926535897932384626433832795029
   ```

7. Dereferencing is the only coercion in arithmetic. Names can be compared through `IS` and `ISNT`, and `NIL` is supplied, but there is no soft coercion of operands.

10.9 The Preprocessor

a68g has a basic preprocessor. Currently, the preprocessor supports these features:

1. concatenation of lines,
2. inclusion of files,
3. refinement preprocessor,
4. switching the preprocessor on or off.

10.9.1 Concatenation of lines

Concatenation of lines is similar to what the C preprocessor does. Any line that ends in a backslash (′́) will be concatenated with the line following it. For example:

```
STRING s := "spanning two \
lines"
```

will become

```
STRING s := "spanning two lines"
```

Using a backslash as an escape character causes no interference with Algol 68 source text since when using upper stropping, a backslash is an unworthy character; when using quote stropping a backslash is a times-ten-symbol, but a real-denotation cannot span end-of-line hence no interference occurs.

Note that when you make use of line concatenation, diagnostics in a concatenated line may be placed in an earlier source line than its actual line in the original source text. In order to preserve line numbering as much as possible, a line is emptied but not deleted if it is joined with a preceding one. This shows in the listing file as emptied lines.

10.9.2 Inclusion of files

a68g supports inclusion of other files in an Algol 68 source text. The inclusion directive reads:

```
PR read "filename" PR
```

or

```
PR include "filename" PR
```

The file with name filename is inserted textually before the line that holds the file inclusion directive. In this way tokens remain in their original lines, which will give more accurate placement of diagnostics. It is therefore recommended that a file inclusion directive be the only text on the line it is in. A file will only be inserted once, on attempted multiple inclusion the file is ignored. Attempted multiple inclusion may for example result from specifying, in an included file, an inclusion directive for an already included file.

10.9.3 The refinement preprocessor

Algol 68 Genie is equipped with a basic refinement preprocessor, which allows programming through stepwise refinement as taught in [Koster 1978, 1981]. The idea is to facilitate program construction by elaborating the description of the solution to a problem as ever more detailed steps until the description of the solution is complete. (See also Wirth's well known lecture).

Refinement syntax can be found in the syntax summary. Refinements cannot be recursive, nor can their definitions be nested. Also, refinement definitions must be unique, and a refinement can only be applied once (refinements are not procedures). *a68g* will check whether a program looks like a stepwise refined program. The refinement preprocessor is transparent to programs that are not stepwise refined.

10.9.4 Switching the preprocessor on or off

t is possible to switch the preprocessor on or off. The preprocessor is per default switched on. If switched off, it will no longer process preprocessor items embedded in pragmats, except for switching the preprocessor on again. Concatenation of lines takes place even if the preprocessor is switched off.

```
PR preprocessor PR
```

Switches the preprocessor on if it is switched off.

```
PR nopreprocessor PR
```

Switches the preprocessor off if it is switched on.

10.10 Interpreter speed

The speed of the interpreter will of course depend on platform and application. *a68g* first constructs a syntax tree for a program where after the interpreter "executes" this syntax tree, therefore interpreter speed is relatively good. As to have a measure of speed *a68g* was rated with the Whetstone benchmark (see section 12.3). Next table compiles approximate ratings for *a68g* and for some legacy machines (which ran Algol 68 implementations).

System	MWhets	Remarks
ICL 1906A	0.6	ALGOL68R
VAX 11/780	1	FORTRAN
CYBER 205	12 (Scalar)	FORTRAN
Pentium 4 (3 GHz, single core)	44	*a68g*/Linux 2.6.26 i686/*gcc* 4.3.1
Pentium 4 (3 GHz)	820	C/OpenBSD/*gcc*

From above data one could conclude that *a68g* on a 3 GHz Pentium runs the Whetstone test roughly as fast as C on a ″160 MHz″ Pentium. However *a68g* offers many runtime checks while the compiled C code offered none, therefore this speed comparison overrates the speed of compiled C code by a factor of probably 2 to 3.

10.11 Limitations and bugs

Next *a68g* issues are known:

1. The interpreter offers optional checking of the system stack. When this check is not activated, or on systems where this check would not work, the following may result in a segment violation (and possibly a core dump): (1) deep recursion, (2) garbage collection of deeply recursive data structures or (3) using jumps to move between incarnations of recursive procedures.

2. When the stack overhead is set to a too small value, a segment violation may occur.

3. Algorithms for extended precision (LONG LONG arithmetic modes) are not really suited for precisions larger than about a thousand digits. State of the art in the field offers more efficient algorithms than implemented here.

4. Overflow- and underflow checks on `REAL` and `COMPLEX` operations require IEEE-754 compatibility. Many processor types, notably ix86's and PowerPC processors, are IEEE-754 compatible.

5. A garbage collector cannot solve all memory allocation issues. It is therefore possible to get an unexpected ″out of memory″ diagnostic. Two options in such case are (1) to increase heap size or (2) to call standard-prelude routine ″sweep heap″ or ″preemptive sweep heap″ at strategic positions in the program.

6. In some versions of `libplot`, pseudo-gif plotters produce garbled graphics when more than 256 different colours are specified. Algol 68 Genie does not work around this problem in `libplot`.

7. Some platforms cannot give `libplot` proper support for all plotters. Linux with the X window system lets `libplot` implement all plotters but for example one Win32 binary has problems with X and postscript plotters. For example, the Win32 executable provided for *a68g* will give runtime errors as `X plotter missing` or `postscript plotter missing`. On other platforms it is possible that a plotter produces garbage or gives a message as `output stream jammed`. Algol 68 Genie does not work around this deficiency.

8. In some versions of `libplot`, X plotters do not flush after every plotting operation. It may happen that a plotting operation does not show until `close` is called for that plotter (after closing, an X plotter window stays on the screen (as a forked process) until you type `"q"` while it has focus, or click in it). Algol 68 Genie does not work around this buffering mechanism in `libplot`.

Chapter 11

Prelude

11.1 The standard-environ

An Algol 68 program run with *a68g* is embedded in the next environ:

```
1  BEGIN
2     COMMENT
3     Here the standard-prelude and library-prelude are included.
4     COMMENT
5     ...
6     BEGIN
7        MODE DOUBLE = LONG REAL,
8             QUAD = LONG LONG REAL,
9             DEVICE = FILE,
10            TEXT = STRING;
11       start: commence:
12       BEGIN COMMENT
13             Here your program is embedded.
14             COMMENT
15             ...
16       END;
17       stop: abort: halt: SKIP
18    END
19 END
```

At line 5 the prelude is included. At line 15 your code is included. A consequence of this standard-environ is that an *a68g* program does not need to be an enclosed-clause; a serial-clause suffices.

11.2 The standard-prelude

Next sections up to and including the section on transput describe the facilities in the standard-prelude supplied with *a68g* .

11.3 Standard modes

Many of the modes available in the standard-prelude are built from the standard modes of the language which are all defined in the Revised Report.

1. `VOID`
   This mode has one value: `EMPTY`. It is used as the yield of routines, in casts and in unions.
2. `INT`
   In *a68g* , these precisions are available:
   (a) `INT`
   (b) `LONG INT`
   (c) `LONG LONG INT`
3. `REAL`
   In *a68g* , these precisions are available:
   (a) `REAL`
   (b) `LONG REAL`
   (c) `LONG LONG REAL`
4. `BOOL`
   This mode has two values, `TRUE` and `FALSE`.
5. `CHAR`
   This mode is used for most character operations.
6. `STRING`
   This mode is defined as

   ```
   MODE STRING = FLEX [1 : 0] CHAR
   ```

7. `COMPLEX, COMPL`
   This is not a primitive mode because it is a structure with two fields:

   ```
   MODE COMPLEX = STRUCT (REAL re, im)
   ```

However, the widening coercion will convert a `REAL` value into a `COMPLEX` value, and transput routines will not straigthen a `COMPLEX` - or `REF COMPLEX` value. Like `REAL`s, the following precisions are available in *a68g* :

(a) `COMPLEX`
(b) `LONG COMPLEX`
(c) `LONG LONG COMPLEX`

8. `BITS`
   This mode is equivalent to a computer word regarded as a group of bits (binary digits) numbered 1 to `bits width`. These precisions are available in *a68g* :

   (a) `BITS`
   (b) `LONG BITS`
   (c) `LONG LONG BITS`

9. `BYTES`
   This mode stores a row-of-character in a single value. These precisions are available in *a68g* :

   (a) `BYTES`
   (b) `LONG BYTES`

10. `SEMA`
    Semaphores are used to synchronise parallel actions. This mode is defined as

    ```
    MODE SEMA = STRUCT (REF INT F)
    ```

    Note that the field cannot be directly selected.

11. `CHANNEL`
    Channels describe the properties of a `FILE`.

12. `FILE`
    A `FILE` structure holds status of transputting to or from a specific stream of bytes.

13. `FORMAT`
    Holds an internal representation of format-texts and their elaboration.

11.4 Environment enquiries

Algol 68 was the first programming language to contain declarations which enable a programmer to determine the characteristics of the implementation. The enquiries are divided into a number of different groups.

11.4.1 Enquiries about precisions

Any number of `LONG` or `SHORT` can be given in the mode specification of numbers, but only a few such modes are distinguishable in any implementation. The following environment enquiries tell which modes are distinguishable.

1. `INT int lengths`
   1+ the number of extra lengths of integers.
2. `INT int shorths`
   1+ the number of short lengths of integers.
3. `INT real lengths`
   1+ the number of extra lengths of real numbers.
4. `INT real shorths`
   1+ the number of short lengths of real numbers.
5. `INT bits lengths`
   1+ the number of extra lengths of `BITS`.
6. `INT bits shorths`
   1+ the number of short lengths of `BITS`.
7. `INT bytes lengths`
   1+ the number of extra lengths of `BYTES`.
8. `INT bytes shorths`
   1+ the number of short lengths of `BYTES`.

11.4.2 Characteristics of modes

1. `INT max int`
   The maximum value of mode `INT`.
2. `LONG INT long max int`
   The maximum value of mode `LONG INT`.
3. `LONG LONG INT long long max int`
   The maximum value of mode `LONG INT`.
4. `REAL max real`
   The largest real value.
5. `REAL small real`
   The smallest real which, when added to 1.0, gives a sum larger than 1.0.

6. `LONG REAL long max real`
   The largest long real value.
7. `LONG REAL long small real`
   The smallest long real which, when added to 1.0, gives a sum larger than 1.0.
8. `LONG LONG REAL long long max real`
   The largest long long real value.
9. `LONG LONG REAL long long small real`
   The smallest long long real which, when added to 1.0, gives a sum larger than 1.0.
10. `INT int width`
    The maximum number of decimal digits expressible by an integer.
11. `INT long int width`
    The maximum number of decimal digits expressible by a long integer.
12. `INT long long int width`
    The maximum number of decimal digits expressible by a long long integer.
13. `INT bits width`
    The number of bits required to hold a value of mode `BITS`.
14. `INT long bits width`
    The number of bits required to hold a value of mode `LONG BITS`.
15. `INT long long bits width`
    The number of bits required to hold a value of mode `LONG LONG BITS`.
16. `INT bytes width`
    The number of bytes required to hold a value of mode `BYTES`.
17. `INT long bytes width`
    The number of bytes required to hold a value of mode `LONG BYTES`.
18. `INT real width`
    The maximum number of significant decimal digits in a real.
19. `INT exp width`
    The maximum number of decimal digits in the exponent of a real.
20. `INT long real width`
    The maximum number of significant decimal digits in a long real.
21. `INT long exp width`
    The maximum number of decimal digits in the exponent of a long real.
22. `INT long long real width`
    The maximum number of significant decimal digits in a long real.

23. `INT long long exp width`
    The maximum number of decimal digits in the exponent of a long long real.

11.4.3 Mathematical constants

1. `REAL pi`
   3.14159265358979

2. `LONG REAL long pi`
   3.14159265358979323846264338 33

3. `LONG LONG REAL long long pi`
   3.14159265358979323846264338327950288419716939937510582097494459
   at default precision.

11.4.4 Character set enquiries

The absolute value of Algol 68 characters range from 0 to the value of `max abs char`. Furthermore, the operator `REPR` will convert any `INT` up to `max abs char` to a character. What character is represented by `REPR 225` will depend on the character set used by the displaying device.

1. `INT max abs char`
   The largest positive integer which can be represented as a character.

2. `CHAR null character`
   This is `REPR 0`.

3. `CHAR blank`
   This is the space character.

4. `CHAR error char`
   This character is used by the formatting routines for invalid values.

5. `CHAR flip`
   This character is used to represent `TRUE` in transput.

6. `CHAR flop`
   This character is used to represent `FALSE` in transput.

11.5 Standard operators

The number of distinct operators is vastly increased by the availability of SHORT and LONG modes. Thus it is imperative that some kind of shorthand be used to describe the operators. Following the subsection on the method of description are sections devoted to operators with classes of operands. The end of this section contains tables of all the operators.

11.5.1 Method of description

Where an operator has operands and yield which may include LONG or SHORT, the mode is written using L . For example,

```
OP + = (L INT, L INT)L INT:
```

is a shorthand for the following operators:

```
OP + = (INT, INT) INT:
OP + = (LONG INT, LONG INT) LONG INT:
OP + = (LONG LONG INT, LONG LONG INT) LONG LONG INT:
```

Ensure that wherever L is replaced by SHORTs or LONGs, it should be replaced by the same number of SHORTs or LONGs throughout the definition of that operator. This is known as ”consistent substitution”. Note that any number of SHORTs or LONGs can be given in the mode of any value whose mode accepts such constructs (INT, REAL, COMPLEX and BITS), but the only modes which can be distinguished are those specified by the environment enquiries in section 11.4.1. *a68g* maps a declarer whose length is not implemented onto the most appropriate length available [RR 2.1.3.1]. Routines or operators for not-implemented lengths are mapped accordingly. *a68g* considers mapped modes equivalent to the modes they are mapped onto, while standard Algol 68 would still set them apart.

11.5.2 Operator synonyms

Algol 68 provides a plethora of operators, and in some cases also synonyms for operators. In the listings further on in this chapter, only one synonym will be defined per operator. Here is a list of synonyms for operator symbols implemented in *a68g* :

1. AND for &
2. ˆ for **
3. ˜ for NOT

4. `~=` for `/=`
5. `^=` for `/=`
6. `I` for `+*`
7. `EQ` for `=`
8. `NE` for `/=`
9. `LE` for `<=`
10. `LT` for `<`
11. `GE` for `>=`
12. `GT` for `>`
13. `^` for `**`
14. `OVER` for `%`
15. `MOD` for `%*`
16. `PLUSAB` for `+:=`
17. `MINUSAB` for `-:=`
18. `TIMESAB` for `*:=`
19. `DIVAB` for `/:=`
20. `OVERAB` for `%:=`
21. `MODAB` for `%*:=`
22. `PLUSTO` for `+=`

11.5.3 Standard priorities

The priority of an operator is independent of the mode of the operands or result. The standard prelude sets next priorities for operators:

1. `+:=`, `-:=`, `*:=`, `/:=`, `%:=`, `%*:=`, `+=`
2. `OR`
3. `AND`, `XOR`
4. `=`, `/=`
5. `<`, `<=`, `>=`, `>`
6. `-`, `+`
7. `*`, `/`, `%`, `%*`, `ELEM`
8. `**`, `UP`, `DOWN`, `SHL`, `SHR`, `LWB`, `UPB`
9. `+*`, `I`

These priorities can be changed by `PRIO` in your program, but this easily leads to incomprehensible programs. Use `PRIO` for your own dyadic operators.

11.5.4 Operators with row operands

Both monadic and dyadic forms are available. We will use the mode `ROW` to denote the mode of any row.

1. Monadic.
   `OP LWB = (ROW r) INT`
   `OP UPB = (ROW r) INT`
   Yield the lower bound or upper bound for the first dimension of `r`.
   `OP ELEMS = (ROW r) INT`
   Yields the number of elements, in all dimensions, of `r`.

2. Dyadic.
   `OP LWB = (INT n, ROW r) INT`
   `OP UPB = (INT n, ROW r) INT`
   Yield the lower bound or upper bound of the `n`-th dimension of `r`.
   `OP ELEMS = (ROW) INT`
   Yields the number of elements in the `n`-th dimension of `r`.

11.5.5 Operators with `BOOL` operands

1. `OP ABS = (BOOL a) INT`
   `ABS TRUE` yields 1 and `ABS FALSE` yields 0.

2. `OP AND = (BOOL a, b) BOOL`
   Logical `AND`.
3. `OP OR = (BOOL a, b) BOOL`
   Logical inclusive `OR`.
4. `OP XOR = (BOOL a, b) BOOL`
   Logical exclusive OR, `XOR`.
5. `OP NOT = (BOOL a) BOOL` Logical `NOT`: yields `TRUE` if `a` is `FALSE` and yields `FALSE` if `a` is `TRUE` .
6. `OP = = (BOOL a, b) BOOL`
   `TRUE` if `a` equals `b` and `FALSE` otherwise.
7. `OP /= = (BOOL a, b) BOOL`
   `TRUE` if `a` not equal to `b` and `FALSE` otherwise.

11.5.6 Operators with **`INT`** operands

The `L` shorthand is used for operators taking values of any precision.

Monadic operators

Consistent substitution applies to all those operators in this section which use the `L` shorthand: apart from `LENG` and `SHORTEN`, the precision of the yield is the same as the precision of the operand.

1. `OP + = (L INT a) L INT`
   The identity operator.
2. `OP - = (L INT a) L INT`
   The negation operator.
3. `OP ABS = (L INT a) L INT`
   The absolute value: `(a < 0 | - a | a)`.
4. `OP SIGN = (L INT a) INT`
   Yields -1 for a negative operand, $+1$ for a positive operand and 0 for a zero operand.
5. `OP ODD = (L INT a) BOOL`
   Yields `TRUE` if the operand is odd and `FALSE` if it is even. This is a relic of times long past.

6. `OP LENG = (L INT a) LONG L INT`
   Converts its operand to the next longer precision.

7. `OP SHORTEN = (LONG L INT a) L INT` Converts its operand to the next shorter precision. If `a` exceeds `l max int` for the next shorter precision, a runtime error occurs.

Dyadic operators

In this section, consistent substitution is used wherever the `L` shorthand is used. For operators with mixed operands, see section 11.5.9.

1. `OP + = (L INT a, L INT b) L INT`
   Integer addition: $a + b$.

2. `OP - = (L INT a, L INT b) L INT`
   Integer subtraction: $a - b$.

3. `OP * = (L INT a, L INT b) L INT`
   Integer multiplication: $a \times b$.

4. `OP / = (L INT a, L INT b) L REAL`
   Integer fractional division. Even if the quotient is a whole number (for example, $6/3$), the yield always has mode `L REAL`.

5. `OP % = (L INT a, L INT b) L INT`
   Integer division.

6. `OP %* = (L INT a, L INT b) L INT`
   Integer modulo, which always yields a non-negative result (see section 3.6).

7. `OP ** = (L INT a, INT b) L INT`
   Computes a^b for $b \geq 0$.

8. `OP +* = (L INT a, L INT b) L COMPLEX`
   Joins two integers into a complex number $a + bi$ of the same precision.

9. `OP = = (L INT a, L INT b) BOOL`
   Integer equality: $a = b$.

10. `OP /= = (L INT a, L INT b) BOOL`
    Integer inequality: $a \neq b$.

11. `OP < = (L INT a, L INT b) BOOL`
    Integer ”less than” $a < b$.

12. `OP <= = (L INT a, L INT b) BOOL`
    Integer ″not greater than″ $a \leq b$.

13. `OP >= = (L INT a, L INT b) BOOL`
    Integer ″not less than″: $a \geq b$.

14. `OP > = (L INT a, L INT b) BOOL`
    Integer ″greater than″: $a > b$.

11.5.7 Operators with REAL operands

The shorthand `L` is used for these operators can have operands of any precision.

Monadic operators

1. `OP + = (L REAL a) L REAL`
   Real identity.

2. `OP - = (L REAL a) L REAL`
   Real negation: $-a$.

3. `OP ABS = (L REAL a)L REAL`
   The absolute value.

4. `OP SIGN = (L REAL a) INT`
   Yields -1 for negative operands, $+1$ for positive operands and 0 for a zero operand.

5. `OP ROUND = (L REAL a) L INT`
   Rounds its operand to the nearest integer. The operator checks for integer overflow.

6. `OP ENTIER = (L REAL a)L INT`
   Yields the largest integer not larger than the operand. The operator checks for integer overflow.

7. `OP LENG = (L REAL a) LONG L REAL`
   `OP LENG = (SHORT L REAL a) L REAL`
   Converts its operand to the next longer precision.

8. `OP SHORTEN = (LONG L REAL a) L REAL`
   Converts its operand to the next shorter precision. If a value exceeds `l max real` for the next shorter precision, a runtime error occurs. The mantissa will be rounded.

Dyadic operators

In this section, consistent substitution is used wherever the `L` shorthand appears. For operators with mixed operands, see section 11.5.9.

1. `OP + = (L REAL a, L REAL b) L REAL`
   Real addition $a + b$.

2. `OP - = (L REAL a, L REAL b) L REAL`
   Real subtraction $a - b$.

3. `OP * = (L REAL a, L REAl b) L REAL`
   Real multiplication $a \times b$.

4. `OP / = (L REAL a, L REAL b) L REAL`
   Real divison a/b.

5. `OP +* = (L REAL a, L REAL b) L COMPLEX`
   Joins two reals into a complex number $a + bi$ of the same precision.

6. `OP = = (L REAL a, L REAL b) BOOL`
   Real equality: $a = b$.

7. `OP /= = (L REAL a, L REAL b) BOOL`
   Real inequality: $a \neq b$.

8. `OP < = (L REAL a, L REAL b) BOOL`
   Real "less than": $a < b$.

9. `OP <= = (L REAL a, L REAL b) BOOL`
   Real "not greater than": $a \leq b$.

10. `OP >= = (L REAL a, L REAL b) BOOL`
    Real "not less than": $a \geq b$.

11. `OP > = (L REAL a,L REAL b)BOOL`
    Real "greater than": $a > b$.

11.5.8 Operators with `COMPLEX` operands

Algol 68 Genie offers a rich set of operators and routines for complex numbers. Consistent substitution applies to operators using the `L` shorthand.

Monadic operators

1. `OP RE = (L COMPLEX a) L REAL`
   Yields the real component: `re OF a`.

2. `OP IM = (L COMPLEX a) L REAL`
   Yields the imaginary component: `im OF a`.

3. `OP ABS = (L COMPLEX a) L REAL`
   Yields the absolute value (a magnitude) of its argument.

4. `OP ARG = (L COMPLEX a) L REAL`
   Yields the argument of the complex number.

5. `OP CONJ = (L COMPLEX a) L COMPLEX`
   Yields the conjugate complex number.

6. `OP + = (L COMPLEX a) L COMPLEX`
   Complex identity.

7. `OP - = (L COMPLEX a) L COMPLEX`
   Complex negation.

8. `OP LENG = (L COMPLEX a) LONG L COMPLEX`
   Converts its operand to the next longer precision.

9. `OP SHORTEN = (LONG L COMPLEX a) L COMPLEX`
   Converts its operand to the next shorter precision. If either of the components of the complex number exceeds `l max real` for the next shorter precision, a runtime error occurs.

Dyadic operators

In this section, consistent substitution is used wherever the `L` shorthand appears. For operators with mixed operands, see section 11.5.9.

1. `OP + = (L COMPLEX a, L COMPLEX b) L COMPLEX`
   Complex addition for both components $a + b$.

2. `OP - = (L COMPLEX a, L COMPLEX b) L COMPLEX`
   Complex subtraction for both components $a - b$.

3. `OP * = (L COMPLEX a, L COMPLEX b) L COMPLEX`
   Complex multiplication $a * b$.

4. `OP / = (L COMPLEX a, L COMPLEX b) L COMPLEX`
   Complex division a/b.

5. `OP = = (L COMPLEX a, L COMPLEX b) BOOL`
   Complex equality $a = b$.

6. `OP /= = (L COMPLEX a, L COMPLEX b) BOOL`
   Complex inequality $a \neq b$.

11.5.9 Operators with mixed operands

Consistent substitution is applicable to all operators using the `L` shorthand. Additional shorthands are used as follows:

1. The shorthand `P` stands for `+`, `-`, `*` or `/`.
2. The shorthand `R` stands for `<`, `<=`, `=`, `/=`, `>=`, `>`,
   or `LT`, `LE`, `EQ`, `NE`, `GE`, `GT`.
3. The shorthand `E` stands for `=` `/=`,
   or `EQ` or `NE`.

1. `OP P = (L INT a, L REAL b) L REAL`
2. `OP P = (L REAL a, L INT b) L REAL`
3. `OP P = (L INT a, L COMPLEX b) L COMPLEX`
4. `OP P = (L COMPLEX a, L INT b) L COMPLEX`
5. `OP P = (L REAL a, L COMPLEX b) L COMPLEX`
6. `OP P = (L COMPLEX a, L REAL b) L COMPLEX`
7. `OP R = (L INT a, L REAL b) BOOL`
8. `OP R = (L REAL a, L INT b) BOOL`
9. `OP E = (L INT a, L COMPLEX b) BOOL`
10. `OP E = (L COMPLEX a, L INT b) BOOL`
11. `OP E = (L REAL a, L COMPLEX b) BOOL`
12. `OP E = (L COMPLEX a, L REAL b) BOOL`
13. `OP ** = (L REAL a, INT b) L REAL`
14. `OP ** = (L COMPLEX a, INT b) L COMPLEX`
15. `OP +* = (L INT a, L REAL b) L COMPLEX`
16. `OP +* = (L REAL a, L INT b) L COMPLEX`

11.5.10 Operators with `BITS` operands

Consistent substitution applies to all operators using the `L` shorthand.

Monadic operators

1. `OP BIN = (L INT a) L BITS`
   Mode conversion.
2. `OP ABS = (L BITS a) L INT`
   Mode conversion.
3. `OP NOT = (L BITS a) L BITS`
   Yields the bits obtained by inverting each bit in the operand.
4. `OP LENG = (L BITS a) LONG L BITS`
   Converts a bits value to the next longer precision by adding zero bits to the more significant end.
5. `OP SHORTEN = (LONG L BITS a) L BITS`
   Converts a bits value to a value of the next shorter precision.

Dyadic operators

1. `OP AND = (L BITS a, L BITS b) L BITS`
   The logical "AND" of corresponding binary digits in `a` and `b`.
2. `OP OR = (L BITS a, L BITS b) L BITS`
   The logical "OR" of corresponding binary digits in `a` and `b`.
3. `OP SHL = (L BITS a, INT b) L BITS`
   The left operand shifted left by the number of bits specified by the right operand. New bits shifted in are zero. If the right operand is negative, shifting is to the right.
4. `OP SHR = (L BITS a, INT b) L BITS`
   The left operand shifted right by the number of bits specified by the right operand. New bits shifted in are zero. If the right operand is negative, shifting is to the left.
5. `OP ELEM = (INT a,L BITS b) BOOL`
   Yields `TRUE` if bit `a` is set, and `FALSE` if it is not set.
6. `OP = = (L BITS a, L BITS b) BOOL`
   Logical equality $a = b$.

7. `OP /= = (L BITS a, L BITS b) BOOL`
   Logical inequality $a \neq b$.

8. `OP <= = (L BITS a, L BITS b) BOOL`
   Yields `TRUE` `a` is a subset of `b` or `FALSE` otherwise: `(a OR b) = b`

9. `OP >= = (L BITS a, L BITS b) BOOL`
   Yields `TRUE` `b` is a subset of `a` or `FALSE` otherwise: `(a OR b) = a`

11.5.11 Operators with `CHAR` operands

The shorthands in section 11.5.9 apply here.

1. `OP ABS = (CHAR a) INT`
   The integer equivalent of a character.

2. `OP REPR = (INT a) CHAR`
   The character representation of an integer. The operand should be in the range $0 \ldots$ `max abs char`.

3. `OP + = (CHAR a, CHAR b) STRING`
   The character `b` is appended to the character `a` (concatenation).

4. `OP E = (CHAR a, CHAR b) BOOL`
   Equality or inequality of characters.

5. `OP R = (CHAR a, CHAR b) BOOL`
   Relative ordering of characters.

11.5.12 Operators with `STRING` operands

The shorthands in section 11.5.9 apply here.

1. `OP ELEM = (INT a, STRING b) CHAR`
   Yields `b[a]`. This is an *ALGOL68C* operator.

2. `OP + = (STRING a, STRING b) STRING`
   String `b` is appended to string `a` (concatenation).

3. `OP + = (CHAR a, STRING b) STRING`
   String `b` is appended to character `a`.

4. `OP + = (STRING a, CHAR b) STRING`
   Character `b` is appended to string `a`.

5. `OP * = (INT a, STRING b) STRING`
   Yields `a` times string `b`, concatenated.

6. `OP * = (STRING a, INT b) STRING`
   Yields `b` times string `a`, concatenated.

7. `OP * = (INT a, CHAR b) STRING`
   Yields `a` times character `b`, concatenated.

8. `OP * = (CHAR a, INT b) STRING`
   Yields `b` times character `a`, concatenated.

9. `OP E = (STRING a, STRING b) BOOL`
   `OP E = (CHAR a, STRING b) BOOL`
   `OP E = (STRING a, CHAR b) BOOL`
   Equality or inequality of characters and strings.

10. `OP R = (STRING a, STRING b) BOOL`
    `OP R = (CHAR a, STRING b) BOOL`
    `OP R = (STRING a, CHAR b) BOOL`
    Alphabetic ordering of strings.

11.5.13 Operators with BYTES operands

The shorthands in section 11.5.9 apply here.

1. `OP LENG = (BYTES a) LONG BYTES`
   Converts a bytes value to longer width by padding null characters.

2. `OP SHORTEN = (LONG BYTES a) L BYTES`
   Converts a value to normal width.

3. `OP ELEM = (INT a, L BYTES b) CHAR`
   Yields the $a - th$ character in `b`.

4. `OP + = (L BYTES a, L BYTES b) BYTES`
   Concatenation $a + b$

5. `OP E = (L BYTES a, L BYTES b) BOOL`
   Equality or inequality of byte strings.

6. `OP R = (L BYTES a, L BYTES b) BOOL`
   Alphabetic ordering of byte strings.

11.5.14 Operators combined with assignation

Consistent substitution applies to all operators containing the `L` shorthand.

1. `+:=`
   The operator is a shorthand for `a := a + b`.

Left operand	Right operand	Yield
`REF L INT`	`L INT`	`REF L INT`
`REF L REAL`	`L INT`	`REF L REAL`
`REF L COMPLEX`	`L INT`	`REF L COMPLEX`
`REF L REAL`	`L REAL`	`REF L REAL`
`REF L COMPLEX`	`L REAL`	`REF L COMPLEX`
`REF L COMPLEX`	`L COMPLEX`	`REF L COMPLEX`
`REF STRING`	`CHAR`	`REF STRING`
`REF STRING`	`STRING`	`REF STRING`
`REF L BYTES`	`L BYTES`	`REF L BYTES`

2. `+=`
   The operator is a shorthand for `b := a + b`.

Left operand	Right operand	Yield
`STRING`	`REF STRING`	`REF STRING`
`CHAR`	`REF STRING`	`REF STRING`
`L BYTES`	`REF L BYTES`	`REF L BYTES`

3. `-:=`
   The operator is a shorthand for `a := a - b`.

Left operand	Right operand	Yield
`REF L INT`	`L INT`	`REF L INT`
`REF L REAL`	`L INT`	`REF L REAL`
`REF L COMPLEX`	`L INT`	`REF L COMPLEX`
`REF L REAL`	`L REAL`	`REF L REAL`
`REF L COMPLEX`	`L REAL`	`REF L COMPLEX`
`REF L COMPLEX`	`L COMPLEX`	`REF L COMPLEX`

4. `*:=`
   The operator is a shorthand for `a := a * b`.

Left operand	Right operand	Yield
`REF L INT`	`L INT`	`REF L INT`
`REF L REAL`	`L INT`	`REF L REAL`
`REF L COMPLEX`	`L INT`	`REF L COMPLEX`
`REF L REAL`	`L REAL`	`REF L REAL`
`REF L COMPLEX`	`L REAL`	`REF L COMPLEX`
`REF L COMPLEX`	`L COMPLEX`	`REF L COMPLEX`
`REF STRING`	`INT`	`REF L COMPLEX`

5. `/:=`
   The operator is a shorthand for `a := a / b`.

Left operand	Right operand	Yield
`REF L REAL`	`L INT`	`REF L REAL`
`REF L REAL`	`L REAL`	`REF L REAL`
`REF L COMPLEX`	`L INT`	`REF L COMPLEX`
`REF L COMPLEX`	`L REAL`	`REF L COMPLEX`
`REF L COMPLEX`	`L COMPLEX`	`REF L COMPLEX`

6. `OP %:= = (REF L INT a, L INT b) REF L INT`
   The operator is a shorthand for `a := a % b`.

7. `OP %*:= = (REF L INT a, L INT b) REF L INT`
   The operator is a shorthand for `a:=a%*b`.

11.5.15 Synchronisation operators

a68g implements the parallel-clause on platforms that support POSIX threads. See section 9.5.

1. `OP LEVEL = (INT a) SEMA`
   Yields a semaphore whose value is `a`.

2. `OP LEVEL = (SEMA a) INT`
   Yields the level of `a`, that is filed `F OF a`.

3. `OP DOWN = (SEMA a) VOID`
   The level of `a` is decremented. If it reaches 0, then the parallel unit that called this operator is hibernated until another parallel unit increments the level of `a` again.

4. `OP UP = (SEMA a) VOID`
   The level of `a` is incremented and all parallel units that were hibernated due to this semaphore being down are awakened.

11.6 Standard procedures

11.6.1 Procedures for real numbers

The shorthand `L` is used to simplify the list of procedures.

1. `PROC L sqrt = (L REAL x) L REAL`
   The square root of `x` provided that $x \geq 0$.
2. `PROC L curt = (L REAL x) L REAL`
   The cube root of `x` provided that $x \geq 0$.
3. `PROC L exp = (L REAL x) L REAL`
   Yields e^x.
4. `PROC L ln = (L REAL x) L REAL`
   The natural logarithm of `x` provided that $x > 0$.
5. `PROC L log = (L REAL x) L REAL`
   The logarithm of `x` to base 10.
6. `PROC L sin = (L REAL x) L REAL`
   The sine of `x`, where `x` is in radians.
7. `PROC L arcsin = (L REAL x) L REAL`
   The inverse sine of `x`.
8. `PROC L cos = (L REAL x) L REAL`
   The cosine of `x`, where `x` is in radians.
9. `PROC L arccos = (L REAL x) L REAL`
   The inverse cosine of `x`.
10. `PROC L tan = (L REAL x) L REAL`
    The tangent of `x`, where `x` is in radians.
11. `PROC L arctan = (L REAL x)L REAL`
    The inverse tangent of `x`.
12. `PROC L arctan2(L REAL x, y) L REAL`
    The angle whose tangent is y/x. The angle will be in range $[-\pi, \pi]$.
13. `PROC L sinh = (L REAL x) L REAL`
    The hyperbolic sine of `x`.
14. `PROC L arcsinh = (L REAL x) L REAL`
    The inverse hyperbolic sine of `x`.
15. `PROC L cosh = (L REAL x) L REAL`
    The hyperbolic cosine of `x`.
16. `PROC L arccosh = (L REAL x) L REAL`
    The inverse hyperbolic cosine of `x`.
17. `PROC L tanh = (L REAL x)L REAL`
    The hyperbolic tangent of `x`.

18. `PROC L arctanh = (L REAL x)L REAL`
    The inverse hyperbolic tangent of `x`.

11.6.2 Procedures for complex numbers

The shorthand `L` is used to simplify the list of procedures.

1. `PROC L complex sqrt = (L COMPLEX z) L COMPLEX`
   The square root of `z`.
2. `PROC L complex exp = (L COMPLEX z) L COMPLEX`
   Yields e^z.
3. `PROC L complex ln = (L COMPLEX z) L COMPLEX`
   The natural logarithm of `z`.
4. `PROC L complex sin = (L COMPLEX z) L COMPLEX`
   The sine of `z`.
5. `PROC L complex arcsin = (L COMPLEX z) L COMPLEX`
   The inverse sine of `z`.
6. `PROC L complex cos = (L COMPLEX z) L COMPLEX`
   The cosine of `z`.
7. `PROC L complex arccos = (L COMPLEX z) L COMPLEX`
   The inverse cosine of `z`.
8. `PROC L complex tan = (L COMPLEX z)L COMPLEX`
   The tangent of `z`.
9. `PROC L complex arctan = (L COMPLEX z)L COMPLEX`
   The inverse tangent of `z`.
10. `PROC L complex sinh = (L COMPLEX z) L COMPLEX`
    The hyperbolic sine of `z`.
11. `PROC L complex arcsinh = (L COMPLEX z) L COMPLEX`
    The inverse hyperbolic sine of `z`.
12. `PROC L complex cosh = (L COMPLEX z) L COMPLEX`
    The hyperbolic cosine of `z`.
13. `PROC L complex arccosh = (L COMPLEX z) L COMPLEX`
    The inverse hyperbolic cosine of `z`.

14. `PROC L complex tanh = (L COMPLEX z) L COMPLEX`
    The hyperbolic tangent of `z`.

15. `PROC L complex arctanh = (L COMPLEX z) L COMPLEX`
    The inverse hyperbolic tangent of `z`.

11.6.3 Random-number generator

a68g implements a Tausworthe generator that has a period of order 2^{113}, from the GNU Scientific Library.

1. `PROC first random = (INT seed) VOID`
   Initialises the random number generator using `seed`.

2. `PROC L next random = L REAL` Generates the next random number. The result will be in $[0 \ldots 1 >$.

11.6.4 Miscellaneous procedures and operators

1. `PROC L bits pack = ([] BOOL a) L BITS`
   Packs `a` into a value of mode `L BITS`.

2. `PROC L bytes pack = (STRING text) L BYTES`
   Converts `text` to a `L BYTES` representation.

3. `PROC char in string = (CHAR c, REF INT p, STRING t) BOOL`
   If found, assigns position of first occurrence of `c` in `t` to `p` if `p` is not `NIL`. Returns whether `c` is found.

4. `PROC last char in string = (CHAR c, REF INT p, STRING t) BOOL`
   If found, assigns position of last occurrence of `c` in text to `p` if `p` is not `NIL`. Returns whether `c` is found.

5. `PROC string in string = (STRING c, REF INT p, STRING t) BOOL`
   If found, assigns position of first occurrence of `c` in `t` to `p` if `p` is not `NIL`. Returns whether `c` is found.

6. `OP SORT = ([] STRING row) [] STRING`
   Yields a copy of `row`, with elements sorted in ascending order. `SORT` employs a quicksort algorithm

7. `PROC is alnum = (CHAR c) BOOL`
   Checks whether `c` is an alphanumeric character; it is equivalent to

`is alpha(c) OR is digit(c).`

8. `PROC is alpha = (CHAR c) BOOL`
   Checks whether `c` is an alphabetic character; it is equivalent to

   `is upper(c) OR is lower(c)`

   In some locales, there may be additional characters for which this routine holds.

9. `PROC is cntrl = (CHAR c) BOOL`
   Checks whether `c` is a control character.

10. `PROC is digit = (CHAR c) BOOL`
    Checks whether `c` is a digit (0 through 9).

11. `PROC is graph = (CHAR c) BOOL`
    Checks whether `c` is a printable character except space

12. `PROC is lower = (CHAR c) BOOL`
    Checks whether `c` is a lower case character.

13. `PROC is print = (CHAR c) BOOL`
    Checks whether `c` is a printable character including space.

14. `PROC is punct = (CHAR c) BOOL`
    Checks whether `c` is a printable character which is not a space or an alphanumeric character.

15. `PROC is space = (CHAR c) BOOL`
    Checks whether `c` is white-space characters. In the C and POSIX locales, these are space, form-feed, newline, carriage return, horizontal tab, and vertical tab .

16. `PROC is upper = (CHAR c) BOOL`
    Checks whether c is an upper case letter.

17. `PROC is xdigit = (CHAR c) BOOL`
    Checks whether `c` is a hexadecimal digit, that is, one of $0123456789abcdefABCDEF$.

18. `PROC to lower = (CHAR c) CHAR`
    Converts the letter `c` to lower case, if possible.

19. `PROC to upper = (CHAR c) CHAR`
    Converts the letter c to upper case, if possible.

20. `PROC system = (STRING command) INT`
    Passes the string command to the operating system for execution. The operating system is expected to return an integer value that will be returned by system.

21. `PROC break = VOID`
    Raises SIGINT, after which *a68g* will stop in its monitor routine. The monitor may be entered on a unit executed after the actual call to break; see details on breakpoints. To enter the monitor immediately, call debug or monitor.

22. `PROC debug = VOID`
    `PROC monitor = VOID`
    a68g will stop in its monitor routine immediately.

23. `PROC evaluate = (STRING expression) STRING`
    Evaluate expression in the monitor and returning the resulting value as a string.

24. `PROC program idf = STRING`
    Returns the name of the source file.

25. `PROC seconds = REAL`
    Returns time in seconds elapsed since an arbitrary origin (for instance, since process start or since system start).

26. `PROC clock = REAL`
    `PROC cpu time = REAL`
    Returns time in seconds elapsed since interpreter started.

27. `PROC sweep heap = VOID`
    Invokes the garbage collector immediately. Sometimes the interpreter cannot sweep the heap when memory is required, for instance when copying stowed objects. In rare occasions this may lead to a program running out of memory. Calling this routine just before the position where the program ran out of memory might help to keep it running. Otherwise (or, alternatively) a larger heap size should be given to the program.

28. `PROC preemptive sweep heap = VOID`
    Invokes the garbage collector if the heap is ”sufficiently full”. Sometimes the interpreter cannot sweep the heap when memory is required, for instance when copying stowed objects. In rare occasions this may lead to a program running out of memory. Calling this routine just before the position where the program ran out of memory might help to keep it running. Otherwise (or, alternatively) a larger heap size should be given to the program.

29. `PROC garbage = LONG INT`
    Returns the number of bytes recovered by the garbage collector.

30. `PROC collections = INT`
    Returns the number of times the garbage collector was invoked.

31. `PROC collect seconds = REAL`
    Returns an estimate of the time used by the garbage collector.

32. `PROC system stack size = INT`
    Returns the size of the system stack.

33. `PROC system stack pointer = INT`
    Returns the (approximate) value of the system stack pointer.

34. `PROC stack pointer = INT`
    Returns the value of the evaluation stack pointer. (Note: this is an Algol 68 stack, not the system stack)

11.7 Transput

An introduction to Algol 68 transput is in chapter 8. The function of this section is to document all the transput declarations so that you can use it as a reference manual.

11.7.1 Transput modes

Only two modes are available:

1. `FILE` A structure containing details of a file.
2. `CHANNEL` A structure whose fields are routines returning truth values which determine the available methods of access to a file.

11.7.2 Standard channels

a68g initialises four channels at start-up:

1. `CHANNEL stand in channel`
2. `CHANNEL stand out channel`
3. `CHANNEL stand back channel`
4. `CHANNEL stand error channel`

These four channels have similar properties because they use the same access procedures. The standard input channel is `stand in channel`. Files on this channel have the following properties:

stand in channel	
reset possible	FALSE
set possible	FALSE
get possible	TRUE
put possible	FALSE
bin possible	FALSE

The `stand out channel` is the standard output channel. Files on this channel have the following properties:

stand out channel	
reset possible	FALSE
set possible	FALSE
get possible	FALSE
put possible	TRUE
bin possible	FALSE

The `stand back channel` is the standard input/output channel. Files on this channel have the following properties:

stand back channel	
reset possible	TRUE
set possible	TRUE
get possible	TRUE
put possible	TRUE
bin possible	TRUE

The `stand error channel` is the standard error output channel. Files on this channel have the following properties:

stand out channel	
reset possible	FALSE
set possible	FALSE
get possible	FALSE
put possible	TRUE
bin possible	FALSE

11.7.3 Standard files

Four standard files are provided:

1. `REF FILE stand in`
   On Linux this file is opened on `stdin` with `stand in channel`.

2. `REF FILE stand out`
   On Linux this file is opened on `stdout` with `stand out channel`.

3. `REF FILE stand error`
   On Linux this file is opened on `stderror` with `stand error channel`.

4. `REF FILE stand back`
   If this file is used, a temporary file will be established on the file system.

11.7.4 Opening files

These procedures are available for opening files:

1. `PROC establish = (REF FILE f, STRING n, CHANNEL c) INT`
   Establishes a new file `f` with file name `n` and channel `c`. If the file already exists, an `on open error` event occurs. The procedure yields zero on success, otherwise an integer denoting an error.

2. `PROC open = (REF FILE f, STRING n, CHANNEL c) INT`
   Establishes a new file `f` with file name `n` and channel `c`. If the file does not exists, it will be created. The procedure yields zero on success, otherwise an integer denoting an error.

3. `PROC create = (REF FILE f, CHANNEL c) INT`
   Creates a temporary file (on Linux with a unique identification in the directory `/tmp`) using the given channel `c`.

4. `PROC associate = (REF FILE f, REF STRING s) VOID` Associates file `f` with string `s`. Note that the Revised Reporty specifies a `REF [][][] CHAR` argument where *a68g* specifies a `REF STRING` argument. On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition, and on file end is invoked. When a file that is associated with a string is reset, getting restarts from the start of the associated string.

11.7.5 Closing files

Three procedures are provided.

1. `PROC close = (REF FILE f) VOID`
   This is the common procedure for closing a file. The procedure checks whether the file is open.

2. `PROC lock = (REF FILE f) VOID`
   The Algol 68 Revised Report requires `lock` to close the file in such a manner that some

system action is required before it can be reopened. *a68g* closes the file and then removes all access permissions.

3. `PROC scratch = (REF FILE f)VOID`
   The file is closed and then deleted from the file system.

11.7.6 Transput routines

The procedures in this section are responsible for the transput of actual values. Firstly, default-format transput is covered and then binary transput. In each section, the shorthand `L` is used for the various precisions of numbers and bits values.

Default-format transput

1. `PROC put = (REF FILE f,`
   `[] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID`
   Writes `items` to file `f` using default formatting.
2. `PROC print =`
   `([] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID`
   Writes `items` to file standout using default formatting.
3. `PROC get = (REF FILE f,`
   `[] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID`
   Reads `items` from file `f` using default formatting.
4. `PROC read =`
   `([] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID`
   Reads `items` from file standin using default formatting.

Formatted transput

1. `PROC putf = (REF FILE f, [] UNION (OUTTYPE,`
   `PROC (REF FILE) VOID, FORMAT) items) VOID`
   Writes `items` to file `f` using formatted transput.
2. `PROC printf = ([] UNION (OUTTYPE,`
   `PROC (REF FILE) VOID, FORMAT) items) VOID`
   Writes `items` to file standout using formatted transput.
3. `PROC getf = (REF FILE f, [] UNION (INTYPE,`
   `PROC (REF FILE) VOID, FORMAT) items) VOID`
   Reads `items` from file `f` using formatted transput.

4. `PROC readf = ([] UNION (INTYPE,`
   `PROC (REF FILE) VOID, FORMAT) items) VOID`
   Reads `items` from file standin using formatted transput.

Binary transput

Binary transput performs no conversion, thus providing a means of storing data in a compact form in files or reading data.

1. `PROC write bin = ([] OUTTYPE x) VOID`
   This is equivalent to `put bin(stand back,x)`.
2. `PROC put bin =`
   `(REF FILE f, [] OUTTYPE x) VOID`
   This procedure outputs data in a compact form. Then external size is the same as the internal size.
3. `PROC read bin=([]SIMPLIN x)VOID`
   This procedure is equivalent to

   ```
   get bin(stand back, x)
   ```
4. `PROC get bin = (REF FILE f,[] INTYPE x)VOID`
   This procedure reads data in a compact form.

It should also be noted that the procedure `make term`, although usually used with default-format transput, can also be used with binary transput for reading a `STRING`.

11.7.7 Interrogating files

A number of procedures are available for interrogating the properties of files:

1. `PROC bin possible = (REF FILE f) BOOL`
   Yields `TRUE` if binary transput is possible.
2. `PROC put possible = (REF FILE f) BOOL`
   Yields `TRUE` if data can be sent to the file.
3. `PROC get possible = (REF FILE f) BOOL`
   Yields `TRUE` if data can be got from the file.
4. `PROC set possible = (REF FILE f) BOOL`
   Yields `TRUE` if the file can be browsed: that is, if the position in the file for further transput can be set.

5. `PROC compressible = (REF FILE f) BOOL`
   Yields `TRUE` if the file is compressible. This is a dummy routine supplied for backwards compatibility since *a68g* sets the compressible field of all channels to `TRUE`.

6. `PROC reidf possible = (REF FILE f) BOOL`
   Yields `TRUE` if the identification of the file can be changed.

7. `PROC idf = (REF FILE f) STRING`
   Yields the identification string of `f`, if it is set.

8. `PROC term = (REF FILE file) STRING`
   Yields the terminator string of `f`, if it is set.

11.7.8 File properties

1. `PROC make term=(REF FILE f, STRING term) VOID`
   Makes `term` the current string terminator.

11.7.9 Event routines

For each routine, the default behaviour will be described. In each case, if the user routine yields `FALSE`, the default action will be elaborated. If it yields `TRUE`, the action depends on the event.

1.
   ```
   PROC on file end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   PROC on logical file end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   PROC on physical file end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   ```
   These procedure let you provide a routine to be called when end of file is reached on file `f`. The default action on file end is to produce a runtime error.

2.
   ```
   PROC on format end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   ```
   Format end events are caused when in formatted transput, the format gets exhausted. The procedure `on format end` lets you provide a procedure of mode `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, transput simply continues, otherwise the format that just ended is restarted and transput resumes.

3.
   ```
   PROC on line end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   PROC on page end =
   (REF FILE f, PROC (REF FILE) BOOL p) VOID
   ```

Line end and page end events are caused when while reading, the end of line or end of page is encountered. These are events so you can provide a routine that for instance automatically counts the number of lines or pages read. The procedures `on line end` and `on page end` let the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE` , transput continues, otherwise a `new line` in case of end of line, or `new page` in case of end of page, is executed and than transput resumes.

Be careful when reading strings, since end of line and end of page are string terminators! If you provide a routine that mends the line - or page end, be sure to call `new line` or `new page` before returning `TRUE`. In case of a default action, `new line` or `new page` must be called explicitly, for instance in the `read` statement, otherwise you will read nothing but empty strings as you do not eliminate the terminator.

4. `PROC on open error =`
   `(REF FILE f, PROC (REF FILE) BOOL p) VOID`
   Open error events are caused when a file cannot be opened as required. For instance, you want to read a file that does not exist, or write to a read-only file. The procedure `on open error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE` , the program continues, otherwise a runtime error occurs.

5. `PROC on value error =`
   `(REF FILE f, PROC (REF FILE) BOOL p) VOID`
   Value error events are caused when transputting a value that is not a valid representation of the mode of the object being transput. The procedure `on value error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If you do transput on the file within the procedure ensure that a value error will not occur again! If the programmer-supplied routine yields `TRUE` , transput continues, otherwise a runtime error occurs.

6. `PROC on format error =`
   `(REF FILE f, PROC (REF FILE) BOOL p) VOID`
   Format error events are caused when an error occurs in a format, typically when patterns are provided without objects to transput. The procedure `on format error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE` , the program continues, otherwise a runtime error occurs.

7. `PROC on transput error =`
   `(REF FILE f, PROC (REF FILE) BOOL p) VOID`
   This event is caused when an error occurs in transput that is not covered by the other events, typically conversion errors (value out of range et cetera). The procedure `on`

`transput error` lets the programmer provide a procedure with mode `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE` , the program continues, otherwise a runtime error occurs.

11.7.10 Formatting routines

There are four procedures for conversion of numbers to strings. The procedures `whole`, `fixed` and `float` will return a string of `error char`s if the number to be converted cannot be represented in the given width.

1. `PROC whole = (NUMBER v, INT width) STRING`
   The procedure converts integer values. Leading zeros are replaced by spaces and a sign is included if $width > 0$. If `width` is zero, the shortest possible string is yielded. If a real number is supplied for the parameter `v`, then the call `|`fixed(v, width, 0)— is elaborated.

2. `PROC fixed =(NUMBER v, INT width, after) STRING`
   The procedure converts real numbers to fixed point form, that is, without an exponent. The total number of characters in the converted value is given by the parameter `width` whose sign controls the presence of a sign in the converted value as for `whole`. The parameter `after` specifies the number of required digits after the decimal point. From the values of `width` and `after`, the number of digits in front of the decimal point can be calculated. If the space left in front of the decimal point is insufficient to contain the integral part of the value being converted, digits after the decimal point are sacrificed.

3. `PROC float =(NUMBER v, INT width, after, exp) STRING`
   The procedure converts reals to floating-point form (”scientific notation”). The total number of characters in the converted value is given by the parameter `width` whose sign controls the presence of a sign in the converted value as for `whole`. Likewise, the sign of `exp` controls the presence of a preceding sign for the exponent. If `exp` is zero, then the exponent is expressed in a string of minimum length. In this case, the value of `width` must not be zero. Note that `float` always leaves a position for the sign. If there is no sign, a blank is produced instead. The values of `width`, `after` and `exp` determine how many digits are available before the decimal point and, therefore, the value of the exponent. The latter value has to fit into the width specified by `exp` and so, if it cannot fit, decimal places are sacrificed one by one until either it fits or there are no more decimal places (and no decimal point). If it still doesn't fit, digits before the decimal place are also sacrificed. If no space for digits remains, the whole string is filled with `error char`.

4. `PROC real =`
   `(NUMBER x, INT width, after, exp width, modifier) STRING`
   Converts `x` to a `STRING` representation. If `modifier` is a positive number, the resulting string will present `x` with its exponent a multiple of `modifier`. If $modifier = 1$, the returned string is identical to that returned by `float`. A common choice for `modifier` is

3 which returns the so-called engineers notation of `x`. If $modifier \leq 0$, the resulting string will present `x` with `ABS modifier` digits before the decimal point and its exponent adjusted accordingly; compare this to FORTRAN nP syntax.

11.7.11 Layout routines

These routines provide formatting capability on both input and output.

1. `PROC set = (REF FILE f, INT n) INT`
   This routine deviates from the standard Algol 68 definition. It attempts to move the file pointer by `n` character positions with respect to the current position. If the file pointer would as a result of this move get outside the file, it is not changed and the routine set by `on file end` is called. If this routine returns `FALSE`, and end-of-file runtime error is produced. The routine returns an `INT` value representing system-dependent information on this repositioning.

2. `PROC reset = (REF FILE f) VOID`
   Resets the file to the state it was in directly after `open`, `create` or `establish`. Default event-routines are installed and the file pointer is reset.

3. `PROC space = (REF FILE f) VOID`
   The procedure advances the file pointer in file `f` by one character. It does *not* read or write a blank.

4. `PROC backspace = (REF FILE f) VOID`
   The procedure attempts to retract the file pointer in file `f` by one character. It executes

   ```
   VOID (set(f, -1))
   ```

5. `PROC new line = (REF FILE f) VOID`
   On input, skips any remaining characters on the current line and positions the file pointer at the beginning of the next line. This means that all characters on input are skipped until a linefeed character is read. On output, the routine writes a linefeed character.

6. `PROC new page = (REF FILE f) VOID`
   On input, skips any remaining characters on the current page and positions the file pointer at the beginning of the next page. This means that all characters on input are skipped until a formfeed character is read. On output, a formfeed character is written.

11.8 The library-prelude

a68g supplies many routines, operators and constants, not described by the Revised Report, for linear algebra, Fourier transforms, drawing and plotting, PCM sounds and PostgreSQL database support. Many of these routines require optional libraries; if these libraries are not present on your system, *a68g* will not support library-prelude elements that require them.

11.9 *ALGOL68C* transput procedures

For compatibility with *ALGOL68C*, *a68g* offers a number of transput procedures. Following routines read an object of the resulting mode of the respective routine from `stand in`:

1. `PROC INT read int`
2. `PROC LONG INT read long int`
3. `PROC LONG LONG INT read long long int`
4. `PROC REAL read real`
5. `PROC LONG REAL read long real`
6. `PROC LONG LONG REAL read long long real`
7. `PROC COMPLEX read complex`
8. `PROC LONG COMPLEX read long complex`
9. `PROC LONG LONG COMPLEX read long long complex`
10. `PROC BOOL read bool`
11. `PROC BITS read bits`
12. `PROC LONG BITS read long bits`
13. `PROC LONG LONG BITS read long long bits`
14. `PROC CHAR read char`
15. `PROC STRING read string`

Following routines write an object of the parameter mode of the respective routine to `stand out`:

1. `PROC (INT) VOID print int`
2. `PROC (LONG INT) VOID print long int`

3. `PROC (LONG LONG INT) VOID print long long int`
4. `PROC (REAL) VOID print real`
5. `PROC (LONG REAL) VOID print long real`
6. `PROC (LONG LONG REAL) VOID print long long real`
7. `PROC (COMPLEX) VOID print complex`
8. `PROC (LONG COMPLEX) VOID print long complex`
9. `PROC (LONG LONG COMPLEX) VOID print long long complex`
10. `PROC (BOOL) VOID print bool`
11. `PROC (BITS) VOID print bits`
12. `PROC (LONG BITS) VOID print long bits`
13. `PROC (LONG LONG BITS) VOID print long long bits`
14. `PROC (CHAR) VOID print char`
15. `PROC (STRING) VOID print string`

11.10 Drawing and plotting

On systems that have installed the GNU Plotting Utilities, *a68g* can be linked with the `libplot` library to extend the standard-prelude. This section contains a list of routines supported by the current revision of *a68g* .

The GNU Plotting Utilities offer a number of routines for drawing 2-D graphics. This allows for drawing in X windows, postscript format, pseudo graphical interface format (pseudo-gif) and portable any-map format (pnm) from Algol 68. Note that *a68g* is not a full Algol 68 binding for `libplot` .

A plotter (window, postscript file, et cetera) can be considered as a stream to which plotting commands are written. Therefore *a68g* considers plotters to be objects of mode `REF FILE`. This seems consistent with the design of Algol 68 transput since a file is associated with a channel that describes the device, and therefore can be considered as a description of a device driver.

A plotter must be opened and closed, just like any file. To specify the file to be a drawing device, `stand draw channel` is declared, that yields `TRUE` when inspected by `draw possible`. To specify details on the plotter type and page size, the procedure `draw device` has to be called. A sample program is given below:

```
# ’window’ is an X window with 600x400 pixels #
FILE window;
draw device (window, "X", "600x400");
open (window, "Hello!", stand draw channel);
# ’draw erase’ makes the window appear on the screen #
draw erase (window);
# Goto pixel 0.25x600, 0.5x400 #
draw move (window, 0.25, 0.5);
# Colour will be 100% red, 0% green, 0% blue #
draw colour (window, 1, 0, 0);
# Write an aligned text #
draw text (window, "c", "c", "Hello world!");
# Flush the drawing command buffer -
  only needed for real-time plotters like X #
draw show (window);
# Give audience a chance to read the message #
VOID (read char);
# Make the window disappear from the screen #
close (window);
```

In some versions of `libplot` , X plotters do not flush after every plotting operation. It may happen that a plotting operation does not show until `close` is called for that plotter. After closing, an X plotter window stays on the screen (as a forked process) until you type `"q"` while it has focus, or click in it. If you click the close-button of such window, you will get an error message like

```
XIO: fatal IO error 11 (Resource temporarily unavailable)
on X server ":0.0" after 198525 requests (198525 known
processed) with 0 events remaining.
```

Algol 68 Genie does not work around this buffering mechanism in `libplot` .

11.10.1 Setting up a graphics device

1. `PROC draw device =`
   `(REF FILE file, STRING type, STRING params) BOOL`
   `PROC make device =`
   `(REF FILE file, STRING type, STRING params) BOOL`
   Set parameters for plotter `file`. Parameter `type` sets the plotter type. *a68g* supports `"X"` for X Window System, `"ps"` for postscript, `"gif"` for pseudo graphical interface format and `"pnm"` for portable any-map format. `X` plotters in *a68g* are only available with UNIX/Linux/Mac OS X. `X` plotters write graphics to an X Window System display rather than to an output stream. A runtime error results if the plotter could not be opened.

Argument `params` is interpreted as an image size. For X, gif and pnm the syntax reads `x pixelsxy pixels`, for instance `"500x500"`. For postscript, image size is the size of the page on which the graphics display will be positioned. Any ISO page size in the range `"a0"` - `"a4"` or ANSI page size in the range `"a"` - `"e"` may be specified. Recognised aliases are `"letter"` for `"a"` and `"tabloid"` for `"b"`. Other valid page sizes are `"legal"`, `"ledger"` and `"b5"`. The graphics display will be a square region centred on the specified page and occupying its full width, with allowance being made for margins. This routine returns `TRUE` if it succeeds.

Some platforms cannot give `libplot` proper support for all plotters. Linux with the X window system lets `libplot` implement all plotters but for example one Win32 binary has problems with X and postscript plotters. For example, the Win32 executable provided for *a68g* will give runtime errors as `X plotter missing` or `postscript plotter missing`. On other platforms it is possible that a plotter produces garbage or gives a message as `output stream jammed`. Algol 68 Genie does not work around this deficiency.

2. `PROC draw erase = (REF FILE file) VOID`
   Begins the next frame of a multi-frame page, by clearing all previously plotted objects from the graphics display, and filling it with the background colour (if any).

3. `PROC draw show = (REF FILE file) VOID`
   Flushes all pending plotting commands to the display device. This is useful only if the currently selected plotter does real-time plotting, since it may be used to ensure that all previously plotted objects have been sent to the display and are visible to the user. It has no effect on plotters that do not do real-time plotting.

4. `PROC draw move = (REF FILE file, REAL x, y) VOID`
   The graphics cursor is moved to (x, y). The values of `x` and `y` are fractions $0 \ldots 1$ of the page size in respectively the `x` and `y` directions.

5. `PROC draw aspect = (REF FILE file) REAL`
   Yields the aspect ratio $ysize/xsize$.

6. `PROC draw fill style = (REF FILE file, INT level) VOID`
   Sets the fill fraction for all subsequently drawn objects. A value of $level = 0$ indicates that objects should be unfilled, or transparent. This is the default. A value in the range $16r0001 \ldots 16rffff$, indicates that objects should be filled. A value of 1 signifies complete filling; the fill colour will be the colour specified by calling `draw colour`. If $level = 16rffff$, the fill colour will be white. Values $16r0002 \ldots 16rfffe$ represent a proportional saturation level, for instance $16r8000$ denotes a saturation of 0.5.

7. `PROC draw linestyle = (REF FILE file, [] CHAR style) VOID`
   Sets the line style for all lines subsequently drawn on the graphics display. The supported line styles are `"solid"`,`"dotted"`,`"dotdashed"`,`"shortdashed"`,`"longdashed"`,

"dotdotdashed", "dotdotdotdashed", and "disconnected". The first seven correspond to the following dash patterns:

```
"solid"    -------------------------------
"dotted"   - - - - - - - -
"dotdashed"       ---- - ---- - ---- -
"shortdashed"     ---- ---- ---- ----
"longdashed"      ------- ------- -------
"dotdotdashed"    ---- - - ---- - -
"dotdotdotdashed" ---- - - - ---- - - -
```

In the preceding patterns, each hyphen stands for one line thickness. This is the case for sufficiently thick lines, at least. So for sufficiently thick lines, the distance over which a dash pattern repeats is scaled proportionately to the line thickness. The "disconnected" line style is special. A "disconnected" path is rendered as a set of filled circles, each of which has diameter equal to the nominal line thickness. One of these circles is centred on each of the juncture points of the path (i.e., the endpoints of the line segments or arcs from which it is constructed). Circles with "disconnected" line style are invisible. Disconnected circles are not filled. All line styles are supported by all plotters.

8. `PROC draw linewidth = (REF FILE file, REAL thickness) VOID`
   Sets the thickness of all lines subsequently drawn on the graphics display. The value of `thickness` is a fraction $0 \ldots 1$ of the page size in the `y` direction. A negative value resets the thickness to the default. For plotters that produce bitmaps, i.e., X plotters, and pnm plotters, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn.

11.10.2 Specifying colours

In some versions of `libplot` , pseudo-gif plotters produce garbled graphics when more than 256 different colours are specified. Algol 68 Genie does not work around this problem in `libplot` .

1. `PROC draw background colour =`
   `(REF FILE file, REAL red, green, blue) VOID`
   `PROC draw background color =`
   `(REF FILE file, REAL red, green, blue) VOID`
   Sets the background colour for plotter `file` its graphics display, using fractional intensities in a 48-bit RGB colour model. The arguments `red`, `green` and `blue` specify the red, green and blue intensities of the background colour. Each is a real value in the range $0 \ldots 1$. The choice $0, 0, 0$ signifies black, and the choice $1, 1, 1$ signifies white. This procedure affects only plotters that produce bitmaps, i. e. , X plotters and pnm plotters. Its

effect is that when the draw erase procedure is called for this plotter, its display will be filled with the specified colour.

2. `PROC draw background colour name =`
   `(REF FILE file, STRING name) VOID`
   `PROC draw background color name =`
   `(REF FILE file, STRING name) VOID`
   Sets the background colour for plotter `file` its graphics display. Argument `name` is a case insensitive string. Accepted names are essentially those supported by the X Window System.

3. `PROC draw colour = (REF FILE file, REAL red, green, blue) VOID`
   `PROC draw color = (REF FILE file, REAL red, green, blue) VOID`
   Sets the foreground colour for plotter `file` its graphics display, using fractional intensities in a 48-bit RGB colour model. The arguments `red`, `green` and `blue` specify the red, green and blue intensities of the background colour. Each is a real value in the range $0 \dots 1$. The choice $0, 0, 0$ signifies black, and the choice $1, 1, 1$ signifies white.

4. `PROC draw colour name = (REF FILE file, STRING name) VOID`
   `PROC draw color name = (REF FILE file, STRING name) VOID`
   Sets the foreground colour for plotter `file` its graphics display. Argument `name` is a case insensitive string. Accepted names are essentially those supported by the X Window System.

11.10.3 Drawing objects

1. `PROC draw point = (REF FILE file, REAL x, y) VOID`
   The arguments specify the co-ordinates (x, y) of a point that will be drawn. The graphics cursor is moved to (x, y). The values of `x` and `y` are fractions $0 \dots 1$ of the page size in respectively the `x` and `y` directions.

2. `PROC draw line = (REF FILE file, REAL x, y) VOID`
   The arguments specify the co-ordinates (x, y) of a line that will be drawn starting from the current graphics cursor. The graphics cursor is moved to (x, y). The values of `x` and `y` are fractions $0 \dots 1$ of the page size in respectively the `x` and `y` directions.

3. `PROC draw rect = (REF FILE file, REAL x, y) VOID`
   The arguments specify the corner coordinates (x, y) of a rectangle that will be drawn starting from the diagonally opposing corner represented by the current graphics cursor. The graphics cursor is moved to (x, y). The values of `x` and `y` are fractions $0 \dots 1$ of the page size in respectively the `x` and `y` directions.

4. `PROC draw circle = (REF FILE file, REAL x, y, r) VOID`
   The three arguments specifying the centre (x, y) and radius `r` of a circle that is drawn. The graphics cursor is moved to (x, y). The values of `x` and `y` are fractions $0 \dots 1$ of the

page size in respectively the x and y directions. The value of r is a fraction $0 \ldots 1$ of the page size in the y direction.

Note that libplot plots its primitives without applying anti-aliasing. Next program demonstrates how to draw an anti-aliased circle using a method from X. Wu:

```
FILE f;
INT resolution = 1000;
open (f, "circle.pnm", stand draw channel);
make device (f, "pnm", whole(resolution, 0) + "x" + whole (resolution, 0));

PROC circle antialiased = (INT x0, y0, r) VOID:
(
   REAL rgb r = 0.5, rgb g = .5, rgb b = .5;
   REAL alpha = 0.6;
   PROC point = (INT x, x0, y, y0) VOID:
   (
      draw point (f, (x + x0) / resolution, (y + y0) / resolution);
      draw point (f, (y + x0) / resolution, (x + y0) / resolution);
      draw point (f, (x0 - x) / resolution, (y + y0) / resolution);
      draw point (f, (x0 - y) / resolution, (x + y0) / resolution);
      draw point (f, (x + x0) / resolution, (-y + y0) / resolution);
      draw point (f, (y + x0) / resolution, (-x + y0) / resolution);
      draw point (f, (x0 - x) / resolution, (-y + y0) / resolution);
      draw point (f, (x0 - y) / resolution, (-x + y0) / resolution)
   );
   FOR x FROM 0 TO r
   WHILE REAL y = sqrt (r * r - x * x);
         INT y1 = ENTIER y;
         x <= y
   DO IF y1 = y
      THEN draw colour (f, alpha * rgb r, alpha * rgb g, alpha * rgb b);
           point (x, x0, y1, y0)
      ELSE REAL d0 = ABS (y1 - y) * alpha;
           draw colour (f, d0 * rgb r, d0 * rgb g, d0 * rgb b);
           point (x, x0, y1 + 1, y0)
      FI;
      FOR y2 FROM y1 DOWNTO 0
      DO REAL d = (1 - alpha) * sqrt (r * r - x * x - y2 * y2) / r + alpha;
         draw colour (f, d * rgb r, d * rgb g, d * rgb b);
         point (x, x0, y2, y0)
      OD
   OD
);

circle antialiased (resolution OVER 2, resolution OVER 2, resolution OVER 3);
close (f)
```

11.10.4 Drawing text

1. `PROC draw text =`
   `(REF FILE file, CHAR h justify, v justify, [] CHAR s) VOID`
   The justified string `s` is drawn according to the specified justifications. If `h justify` is equal to `"l"`, `"c"`, or `"r"`, then the string will be drawn with left, centre or right justification, relative to the current graphics cursor position. If `v justify` is equal to `"b"`, `"x"`, `"c"`, or `"t"`, then the bottom, baseline, centre or top of the string will be placed even with the current graphics cursor position. The graphics cursor is moved to the right end of the string if left justification is specified, and to the left end if right justification is specified. The string may contain escape sequences of various sorts (see section ”Text string format and escape sequences” in the plotutils manual), though it should not contain line feeds or carriage returns; in fact it should include only printable characters. The string may be plotted at a nonzero angle, if `draw textangle` has been called.

2. `PROC draw text angle = (REF FILE file, INT angle) VOID`
   Argument `angle` specifies the angle in degrees counter clockwise from the `x` (horizontal) axis in the user coordinate system, for text strings subsequently drawn on the graphics display. The default angle is zero. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`.

3. `PROC draw font name = (REF FILE file, [] CHAR name) VOID`
   The case-insensitive string `name` specifies the name of the font to be used for all text strings subsequently drawn on the graphics display. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`. The default font name depends on the type of plotter. It is `"Helvetica"` for X plotters, for pnm it is `"HersheySerif"`. If the argument `name` is NIL or an empty string, or the font is not available, the default font name will be used. Which fonts are available also depends on the type of plotter; for a list of all available fonts, see section ”Available text fonts” in the plotutils manual.

4. `PROC draw font size = (REF FILE file, INT size) VOID`
   Argument `size` is interpreted as the size, in the user coordinate system, of the font to be used for all text strings subsequently drawn on the graphics display. A negative value for `size` sets the size to the default, which depends on the type of plotter. Typically, the default font size is $1/50$ times the size (minimum dimension) of the display. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`.

11.11 Mathematical functions

11.11.1 `COMPLEX` functions

Next routines require that *a68g* is linked to the GNU Scientific Library.

1. `PROC complex sinh = (COMPLEX z) COMPLEX`
   Complex hyperbolic sine of argument `z`.
2. `PROC complex arc sinh = (COMPLEX z) COMPLEX`
   Complex inverse hyperbolic sine of argument `z`.
3. `PROC complex cosh = (COMPLEX z) COMPLEX`
   Complex hyperbolic cosine of argument `z`.
4. `PROC complex arc cosh = (COMPLEX z) COMPLEX`
   Complex inverse hyperbolic cosine of argument `z`.
5. `PROC complex tanh = (COMPLEX z) COMPLEX`
   Complex hyperbolic tangent of argument `z`.
6. `PROC complex arc tanh = (COMPLEX z) COMPLEX`
   Complex inverse hyperbolic tangent of argument `z`.

11.11.2 Error and gamma functions

Next routines require that *a68g* is linked to the GNU Scientific Library. except `inverse erf` and `inverse erfc`.

1. `PROC erf = (REAL x) REAL`
   Error function of argument `x`.
2. `PROC inverse erf = (REAL x) REAL`
   Inverse error function of argument `x`.
3. `PROC erfc = (REAL x) REAL`
   Complementary error function of argument `x`, defined as $1 - erf(x)$.
4. `PROC inverse erfc = (REAL x) REAL`
   Inverse complementary error function of argument `x`.
5. `PROC gamma = (REAL x) REAL`
   Gamma function of argument $\Gamma(x)$.
6. `PROC incomplete gamma = (REAL a, x) REAL`
   Incomplete gamma function $P(a, x)$.
7. `PROC ln gamma = (REAL x) REAL`
   Natural logarithm of the gamma function of argument $ln\Gamma(x)$.

8. `PROC factorial = (REAL x) REAL`
   Factorial of argument `x`, defined as $\Gamma(1 + x)$.

9. `PROC beta = (REAL a, b) REAL`
   Beta function of arguments `a` and `b`, defined as $\Gamma(a)\Gamma(b)/\Gamma(a + b)$.

10. `PROC incomplete beta = (REAL a, b, x) REAL`
    Normalised incomplete beta function of arguments `a`, `b` and `x`.

11.11.3 Airy functions

Next routines require that *a68g* is linked to the GNU Scientific Library.

1. `PROC airy ai = (REAL x) REAL`
   Evaluates the airy function $Ai(x)$.

2. `PROC airy bi = (REAL x) REAL`
   Evaluates the airy function $Bi(x)$.

3. `PROC airy ai derivative = (REAL x) REAL`
   Evaluates the airy function $Ai'(x)$.

4. `PROC airy bi derivative = (REAL x) REAL`
   Evaluates the airy function $Bi'(x)$.

11.11.4 Bessel functions

Next routines require that *a68g* is linked to the GNU Scientific Library.

1. `PROC bessel jn = (REAL n, REAL x) REAL`
   Regular cylindrical Bessel function $J_n(x)$. Argument `n` must be integral.

2. `PROC bessel yn = (REAL n, REAL x) REAL`
   Irregular cylindrical Bessel function $Y_n(x)$. Argument `n` must be integral.

3. `PROC bessel in = (REAL n, REAL x) REAL`
   Regular modified cylindrical Bessel function $I_n(x)$. Argument `n` must be integral.

4. `PROC bessel exp in = (REAL n, REAL x) REAL`
   Scaled regular modified cylindrical Bessel function $e^{-|x|}I_n(x)$. Argument `n` must be integral.

5. `PROC bessel kn = (REAL n, REAL x) REAL`
   Irregular modified cylindrical Bessel function $K_n(x)$. Argument `n` must be integral.

6. `PROC bessel exp kn = (REAL n, REAL x) REAL`
   Scaled regular modified cylindrical Bessel function $e^{-|x|}K_n(x)$. Argument `n` must be integral.

7. `PROC bessel jl = (REAL l, REAL x) REAL`
   Evaluates the regular spherical Bessel function $j_l(x)$. Argument `l` must be integral.

8. `PROC bessel yl = (REAL l, REAL x) REAL`
   Irregular spherical Bessel function $y_l(x)$. Argument `l` must be integral.

9. `PROC bessel exp il = (REAL n, REAL x) REAL`
   Scaled regular modified spherical Bessel function $e^{-|x|}i_l(x)$. Argument `l` must be integral.

10. `PROC bessel exp kl = (REAL n, REAL x) REAL`
    Scaled irregular modified spherical Bessel function $e^{-|x|}k_l(x)$. Argument `l` must be integral.

11. `PROC bessel jnu = (REAL nu, REAL x) REAL`
    Fractional regular cylindrical Bessel function $J_\nu(x)$.

12. `PROC bessel ynu = (REAL nu, REAL x) REAL`
    Fractional irregular cylindrical Bessel function $Y_\nu(x)$.

13. `PROC bessel inu = (REAL nu, REAL x) REAL`
    Evaluates the fractional regular modified Bessel function $J_\nu(x)$.

14. `PROC bessel exp inu = (REAL nu, REAL x) REAL`
    Scaled fractional regular modified Bessel function $e^{-|x|}I_\nu(x)$.

15. `PROC bessel knu = (REAL nu, REAL x) REAL`
    Fractional irregular modified Bessel function $Y_\nu(x)$.

16. `PROC bessel exp knu = (REAL nu, REAL x) REAL`
    Scaled fractional irregular modified Bessel function $e^{-|x|}K_\nu(x)$.

11.11.5 Elliptic integrals

Next routines require that *a68g* is linked to the GNU Scientific Library.

1. `PROC elliptic integral k = (REAL x) REAL`
   Complete elliptic integral of the first kind $K(x)$.

2. `PROC elliptic integral e = (REAL x) REAL`
   Complete elliptic integral of the second kind $E(x)$.

3. `PROC elliptic integral rf = (REAL x, y, z) REAL`
   Carlson's elliptic integral of the first kind $R_F(x, y, z)$.

4. `PROC elliptic integral rd = (REAL x, y, z) REAL`
   Carlson's elliptic integral of the second kind $R_D(x, y, z)$.

5. `PROC elliptic integral rj = (REAL x, y, z, rho) REAL`
   Carlson's elliptic integral of the third kind $R_J(x, y, z, \rho)$.

6. `PROC elliptic integral rc = (REAL x, y) REAL`
   Carlson's elliptic integral of the third kind $R_C(x, y)$.

11.12 Linear algebra

Next routines require the GNU Scientific Library. These routines provide a simple vector and matrix interface to Algol 68 rows of mode:

```
[] REAL       # vector #
[, ] REAL     # matrix #
[] COMPLEX    # complex vector #
[, ] COMPLEX  # complex matrix #
```

Routines in this chapter convert Algol 68 rows to objects compatible with vector- and matrix formats used by BLAS routines. However, they are always passed to GSL routines.

11.12.1 Monadic operators

1.
   ```
   OP + = ([] REAL u) [] REAL
   OP + = ([, ] REAL u) [, ] REAL
   OP + = ([] COMPLEX u) [] COMPLEX
   OP + = ([, ] COMPLEX u) [, ] COMPLEX
   ```
   Evaluate $+u$.

2.
   ```
   OP - = ([] REAL u) [] REAL
   OP - = ([, ] REAL u) [, ] REAL
   OP - = ([] COMPLEX u) [] COMPLEX
   OP - = ([, ] COMPLEX u) [, ] COMPLEX
   ```
   Evaluate $-u$.

3.
   ```
   OP NORM = ([] REAL u) REAL
   OP NORM = ([] COMPLEX u) REAL
   ```
   Euclidian norm of vector `u`.

4. `OP TRACE = ([, ] REAL u) REAL`
   `OP TRACE = ([, ] COMPLEX u) COMPLEX`
   Trace (sum of diagonal elements) of square matrix `u`.

5. `OP T = ([, ] REAL u) [, ] REAL`
   `OP T = ([, ] COMPLEX u) [, ] COMPLEX`
   Transpose of matrix `u`. Note that operator `T` yields a copy of the transpose of its argument, whereas pseudo-operator `TRNSP` yields a descriptor without copying its argument.

6. `OP DET = ([, ] REAL u) REAL`
   `OP DET = ([, ] COMPLEX u) COMPLEX`
   Determinant of square matrix `u` by *LU* decomposition.

7. `OP INV = ([, ] REAL u) [, ] REAL`
   `OP INV = ([, ] COMPLEX u) [, ] COMPLEX`
   Inverse of square matrix `u` by *LU* decomposition.

11.12.2 Dyadic operators

1. `OP = = ([] REAL u, v) BOOL`
   `OP = = ([, ] REAL u, v) BOOL`
   `OP = = ([] COMPLEX u, v) BOOL`
   `OP = = ([, ] COMPLEX u, v) BOOL`
   Evaluate whether `u` equals `v`.

2. `OP /= = ([] REAL u, v) BOOL`
   `OP /= = ([, ] REAL u, v) BOOL`
   `OP /= = ([] COMPLEX u, v) BOOL`
   `OP /= = ([, ] COMPLEX u, v) BOOL`
   Evaluate whether `u` does not equal `v`.

3. `OP DYAD = ([] REAL u, v) [, ] REAL`
   `OP DYAD = ([] COMPLEX u, v) [, ] COMPLEX`
   Dyadic (or tensor) product of `u` and `v`. The priority of `DYAD` is 3. This means that (assuming standard priorities) addition, subtraction, multiplication and division have priority over a dyadic product:
   `r + dr DYAD t * r`
   is equivalent to
   `(r + dr) DYAD (t * r)`.

4. `OP + = ([] REAL u, v) [] REAL`
   `OP + = ([, ] REAL u, v) [, ] REAL`
   `OP + = ([] COMPLEX u, v) [] COMPLEX`
   `OP + = ([, ] COMPLEX u, v) [, ] COMPLEX`
   Evaluate $u + v$.

5. `OP +:= ([] REAL u, v) [] REAL`
   `OP +:= ([, ] REAL u, v) [, ] REAL`
   `OP +:= ([] COMPLEX u, v) [] COMPLEX`
   `OP +:= ([, ] COMPLEX u, v) [, ] COMPLEX`
   Evaluate `u := u + v`.

6. `OP - = ([] REAL u, v) [] REAL`
   `OP - = ([, ] REAL u, v) [, ] REAL`
   `OP - = ([] COMPLEX u, v) [] COMPLEX`
   `OP - = ([, ] COMPLEX u, v) [, ] COMPLEX`
   Evaluate $u - v$.

7. `OP -:= ([] REAL u, v) [] REAL`
   `OP -:= ([, ] REAL u, v) [, ] REAL`
   `OP -:= ([] COMPLEX u, v) [] COMPLEX`
   `OP -:= ([, ] COMPLEX u, v) [, ] COMPLEX`
   Evaluate `u := u - v`.

8. `OP * = ([] REAL u, REAL v) [] REAL`
   `OP * = ([, ] REAL u, REAL v) [, ] REAL`
   `OP * = ([] COMPLEX u, COMPLEX v) [] COMPLEX`
   `OP * = ([, ] COMPLEX u, COMPLEX v) [, ] COMPLEX`
   Scaling of `u` by a scalar `v`: $u \times v$.

9. `OP * = (REAL u, [] REAL v) [] REAL`
   `OP * = (REAL u, [, ] REAL v) [, ] REAL`
   `OP * = (COMPLEX u, [] COMPLEX v) [] COMPLEX`
   `OP * = (COMPLEX u, [, ] COMPLEX v) [, ] COMPLEX`
   Scaling of `u` by a scalar `v`: $u \times v$.

10. `OP * = ([, ] REAL u, [] REAL v) [] REAL`
    `OP * = ([, ] COMPLEX u, [] COMPLEX v) [] COMPLEX`
    Matrix-vector product $u \cdot v$.

11. `OP * = ([] REAL u, [, ] REAL v) [] REAL`
    `OP * = ([] COMPLEX u, [, ] COMPLEX v) [] COMPLEX`
    Vector-matrix product $u \cdot v$, that equals $v^T \cdot u$.

12. `OP * = ([] REAL u, v) REAL`
    `OP * = ([] COMPLEX u, v) COMPLEX`
    Inner product of $u \cdot v$.

13. `OP * = ([, ] REAL u, [, ] REAL v) [, ] REAL`
    `OP * = ([, ] COMPLEX u, [, ] COMPLEX v) [, ] COMPLEX`
    Matrix-matrix product $u \cdot v$.

14. `OP *:= = (REF [] REAL u, REAL v) REF [] REAL`
    `OP *:= = (REF [, ] REAL u, REAL v) REF [, ] REAL`
    `OP *:= = (REF [] COMPLEX u, COMPLEX v) REF [] COMPLEX`
    `OP *:= = (REF [, ] COMPLEX u, COMPLEX v) REF [, ] COMPLEX`
    Scaling by a scalar `u := u * v`.

15. `OP / = ([] REAL u, REAL v) [] REAL`
    `OP / = ([, ] REAL u, REAL v) [, ] REAL`
    `OP / = ([] COMPLEX u, COMPLEX v) [] COMPLEX`
    `OP / = ([, ] COMPLEX u, COMPLEX v) [, ] COMPLEX`
    Scaling by a scalar u/v.

16. `OP /:= = (REF [] REAL u, REAL v) REF [] REAL`
    `OP /:= = (REF [, ] REAL u, REAL v) REF [, ] REAL`
    `OP /:= = (REF [] COMPLEX u, COMPLEX v) REF [] COMPLEX`
    `OP /:= = (REF [, ] COMPLEX u, COMPLEX v) REF [, ] COMPLEX`
    Evaluate `u := u / v`.

11.13 Solution of linear algebraic equations

This paragraph describes routines for solving linear algebraic equations. These are intended for "small" linear equations where simple algorithms are acceptable. Should you be interested in solving large linear equations please refer to for instance LAPACK.

11.13.1 *LU* decomposition through Gaussian elimination

A square matrix has an *LU* decomposition into upper and lower triangular matrices

$$P \cdot A = L \cdot U$$

where A is a square matrix, P is a permutation matrix, L is a unit lower triangular matrix and U is an upper triangular matrix. For square matrices this decomposition can be used to convert the linear equation $A \cdot x = b$ into a pair of triangular equations

$$L \cdot y = P \cdot b, U \cdot x = y$$

which can be solved by forward- and back-substitution.

The algorithm that GSL uses in *LU* decomposition is Gaussian elimination with partial pivoting. Advantages of *LU* decomposition are that it works for any square matrix and will efficiently produce all solutions for a linear equation. A disadvantage of *LU* decomposition is that that it cannot find approximate (least-square) solutions in case of singular or ill-conditioned matrices, that are better solved by a singular value decomposition.

1. `PROC lu decomp =`
   `([, ] REAL a, REF [] INT p, REF INT sign) [, ] REAL`
   `PROC complex lu decomp =`
   `([, ] COMPLEX a, REF [] INT p, REF INT sign) [, ] COMPLEX`
   These routines factorise the square matrix `a` into the *LU* decomposition $P \cdot A = L \cdot U$. The diagonal and upper triangular part of the returned matrix contain U. The lower triangular part of the returned matrix holds L. Diagonal elements of L are unity, and are not stored. Permutation matrix P is encoded in the permutation `p`. The sign of the permutation is given by `sign`. It has the value $(-1)^n$, where `n` is the number of row interchanges in the permutation.
2. `PROC lu solve = ([, ] REAL a, lu, [] INT p, [] REAL b) [] REAL`
   `PROC complex lu solve =`
   `([, ] COMPLEX a, lu, [] INT p, [] COMPLEX b) [] COMPLEX`
   These routines solve the equation $A \cdot x = b$ and apply an iterative improvement to `x`, using the *LU* decomposition of `a` into `lu`, `p` as calculated by `[complex] lu decomp`.
3. `PROC lu inv = ([, ] REAL lu, [] INT p) [, ] REAL`
   `PROC complex lu inv = ([, ] COMPLEX lu, [] INT p) [, ] COMPLEX`
   These routines yield the inverse of a matrix from its *LU* decomposition `lu`, `p` as calculated by `[complex] lu decomp`. The inverse is computed by solving $A \cdot A^{inv} = 1$. It is not recommended to use the inverse matrix to solve a linear equation $A \cdot x = b$ by applying $x = A^{inv} \cdot b$; use `[complex] lu solve` for better precision instead.
4. `PROC lu det = ([, ] REAL lu, INT sign) REAL`
   `PROC complex lu det = ([, ] COMPLEX lu, INT sign) COMPLEX`
   These routines yield the determinant of a matrix from its *LU* decomposition `lu`, `sign` as calculated by `[complex] lu decomp`. The determinant is computed as the product of the diagonal elements of U and the sign of the row permutation `sign`.

11.13.2 Singular value decomposition

A $M \times N$ matrix A has a singular value decomposition in the product of a $M \times N$ orthogonal matrix U, a $N \times N$ diagonal matrix S and the transpose of a $N \times N$ orthogonal square matrix V, such that

$$A = U \cdot S \cdot V^T$$

The singular values S_{ii} are zero or positive, off-diagonal elements are zero.

Singular value decomposition has many practical applications. The ratio of the largest to the smallest singular value is called the condition number; a high ratio means that the equation is ill conditioned. Zero singular values indicate a singular matrix A, the number of non-zero singular values is the rank of A. Small singular values should be edited by choosing a suitable tolerance since finite numerical precision may result in a singular value to be close to, but no equal to, zero.

1. `PROC sv decomp =`
   `([, ] REAL a, REF [, ] REAL v, REF [] REAL s) [, ] REAL`
   This routine factorises a $M \times N$ matrix A into the singular value decomposition for $M \geq N$. The routine yields U. The diagonal elements of singular value matrix S are stored in vector `s`. The singular values are non-negative and form a non-increasing sequence from s_1 to s_N. Matrix `v` contains on completion the elements of V in untransposed form. To form the product
   $$U \cdot S \cdot V^T$$
   it is necessary to take the transpose of V. This routine uses the Golub-Reinsch SVD algorithm.

2. `PROC svd solve = ([, ] REAL u, v, [] REAL s, [] REAL b) [] REAL`
   This routine solves the system $A \cdot x = b$ using the singular value decomposition u, s, v of A computed by `svd decomp`. Only non-zero singular values are used in calculating the solution. The parts of the solution corresponding to singular values of zero are ignored. Other singular values can be edited out by setting them to zero before calling this routine. In the overdetermined case where A has more rows than columns the routine returns solution `x` which minimises $||A \cdot x - b||^2$.

11.13.3 QR decomposition

A $M \times N$ matrix A has a decomposition into the product of an orthogonal $M \times M$ square matrix Q, where $Q^T \cdot Q = I$, and an $M \times N$ right-triangular matrix R, such that $A = Q \cdot R$. A linear equation $A \cdot x = b$ can be converted to the triangular equation $R \cdot x = Q^T \cdot b$, which can be solved by back-substitution.

QR decomposition can be used to determine an orthonormal basis for a set of vectors since the first N columns of Q form an orthonormal basis for the range of A.

1. `PROC qr decomp = ([, ] REAL a, REF [] REAL t) [, ] REAL`
   This routine factorises the $M \times N$ matrix `a` into the QR decomposition $A = Q \cdot R$. The diagonal and upper triangular part of the returned matrix contain matrix R. The vector `t` and the columns of the lower triangular part of matrix `a` contain the Householder coefficients and Householder vectors which encode the orthogonal matrix Q. Vector `t` must have length $min(M, N)$. The algorithm used to perform the decomposition is Householder QR (Golub and Van Loan, *Matrix Computations*, Algorithm 5.2.1).

2. `PROC qr solve ([, ] REAL a, [] REAL t, [] REAL b) [] REAL`
   This routine solves the square system $A \cdot x = b$ using the QR decomposition of A into a, t computed by `qr decomp`. The least-squares solution for rectangular equations can be found using `qr ls solve`.

3. `PROC qr ls solve ([, ] REAL a, [] REAL t, [] REAL b) [] REAL`
   This routine finds the least squares solution to the overdetermined system $A \cdot x = b$ where

$M \times N$ matrix A has more rows than columns, id est $M > N$. The least squares solution minimises the Euclidean norm of the residual $||A \cdot x - b||^2$. The routine uses the QR decomposition of A into a, t computed by `qr decomp`.

11.13.4 Cholesky decomposition

A symmetric, positive definite square matrix A has a Cholesky decomposition into a product of a lower triangular matrix L and its transpose L^T, $A = L \cdot L^T$. This is sometimes referred to as taking the square root of a matrix. The Cholesky decomposition can only be carried out when all the eigen values of the matrix are positive. This decomposition can be used to convert the linear system $A \cdot x = b$ into a pair of triangular equations $L \cdot y = b$, $L^T \cdot x = y$, which can be solved by forward and back-substitution.

1. `PROC cholesky decomp = ([, ] REAL a) [, ] REAL`
   This routine factorises the positive-definite symmetric square matrix `a` into the Cholesky decomposition $A = L \cdot L^T$. The diagonal and lower triangular part of the returned matrix contain matrix L. The upper triangular part of the input matrix contains L^T, the diagonal terms being identical for both L and L^T. If the matrix is not positive-definite then the decomposition will fail.

2. `PROC cholesky solve = ([, ] REAL a, [] REAL b) [] REAL`
   This routine solves the system $A \cdot x = b$ using the Cholesky decomposition of A into matrix `a` computed by `cholesky decomp`.

11.14 Fourier transforms

This section describes mixed-radix discrete fast Fourier transform (FFT) algorithms for complex data. The forward transform (to the time domain) is defined by

$$F_k = \sum_{j=0}^{N-1} e^{-2\pi ik/N} f_j$$

while the inverse transform (to the frequency domain) is defined by

$$f_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{2\pi ik/N} F_j$$

and the backward transform is an unscaled inverse transform defined by

$$f_k = \sum_{j=0}^{N-1} e^{2\pi ik/N} F_j$$

The mixed-radix procedures work for FFTs of any length. They are a reimplementation of Paul Swarztrauber's FFTPACK library.

The mixed-radix algorithm is based on sub-transform modules - highly optimised small length FFTs which are combined to create larger FFTs. There are efficient modules for factors of 2, 3, 4, 5, 6 and 7. The modules for the composite factors of 4 and 6 are faster than combining the modules for 2×2 and 2×3.

For factors which are not implemented as modules there is a general module which uses Singleton's method. Lengths which use the general module will still be factorised as much as possible. For example, a length of 143 will be factorised into 11×13. Large prime factors, e.g. 99991, should be avoided because of the $O(n^2)$ complexity of the general module. The procedure `prime factors` can be used to detect inefficiencies in computing a FFT.

For physical applications it is important to note that the index appearing in the discrete Fourier transform (DFT) does not correspond directly to a physical frequency. If the time-step of the DFT reads `dt` then the frequency-domain includes both positive and negative frequencies, ranging from $-1/(2dt)$ to $+1/(2dt)$. The positive frequencies are stored from the beginning of the row up to the middle, and the negative frequencies are stored backwards from the end of the row. When N is even, next frequency table holds:

Index	Time	Frequency
1	0	0
2	dt	$1/(Ndt)$
...	...	...
$N/2$	$(N/2-1)dt$	$(N/2-1)/(Ndt)$
$N/2+1$	$(N/2)dt$	$\pm 1/(2dt)$
$N/2+2$	$(N/2+1)dt$	$-(N/2-1)/(Ndt)$
...	...	...
$N-1$	$(N-2)dt$	$-2/(Ndt)$
N	$(N-1)dt$	$-1/(Ndt)$

When the length N of the row is even, the location $N/2+1$ contains the most positive and negative frequencies $\pm 1/(2dt)$ that are equivalent. If N is odd, this central value does not appear, but the general structure of above table still applies.

1. `PROC prime factors = (INT n) [] INT`
   Factorises argument `n` and yields the factors as a row of integral values. This routine can be used to determine whether the Fourier transform of a row of length `n` can be calculated in an efficient way.

2.
   ```
   PROC fft complex forward = ([] COMPLEX) [] COMPLEX
   PROC fft complex backward = ([] COMPLEX) [] COMPLEX
   PROC fft complex inverse = ([] COMPLEX) [] COMPLEX
   PROC fft forward = ([] REAL) [] COMPLEX
   ```

`PROC fft backward = ([] COMPLEX) [] REAL`
`PROC fft inverse = ([] COMPLEX) [] REAL`
These procedures compute forward, backward (i.e., an unscaled inverse) and inverse FFTs, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length of the argument rows. Efficient modules are provided for sub transforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(N^2)$, module.

11.15 Constants

This paragraph describes physical constants such as the speed of light or the gravitational constant. The constants are available in the MKSA system (meter-kilograms-seconds-ampere) and the CGS system (centimeter-grams-seconds).

11.15.1 Fundamental constants

1. `REAL mksa speed of light`
   `REAL cgs speed of light`
   Speed of light in vacuum, c.
2. `REAL mksa vacuum permeability`
   Permeability of vacuum μ_0.
3. `REAL mksa vacuum permittivity`
   Permittivity of free space ϵ_0.
4. `REAL num avogadro`
   Avogadro's number N_a.
5. `REAL mksa faraday`
   `REAL cgs faraday`
   Molar charge of one Faraday.
6. `REAL mksa boltzmann`
   `REAL cgs boltzmann`
   Boltzmann constant k_B.
7. `REAL mksa molar gas`
   `REAL cgs molar gas`
   Molar gas constant R.
8. `REAL mksa standard gas volume`
   `REAL cgs standard gas volume`
   Standard gas volume V_0.

9. `REAL mksa planck constant`
   `REAL cgs planck constant`
   Planck constant h.

10. `REAL mksa planck constant bar`
    `REAL cgs planck constant bar`
    Planck constant $\hbar = h/(2\pi)$.

11. `REAL mksa gauss`
    `REAL cgs gauss`
    Magnetic field strength of one Gauss.

11.15.2 Length and velocity

1. `REAL mksa micron`
   `REAL cgs micron`
   Length of one micron.

2. `REAL mksa hectare`
   `REAL cgs hectare`
   Area of one hectare.

3. `REAL mksa miles per hour`
   `REAL cgs miles per hour`
   Speed of one mile per hour.

4. `REAL mksa kilometers per hour`
   `REAL cgs kilometers per hour`
   Speed of one kilometer per hour.

11.15.3 Astronomy and astrophysics

1. `REAL mksa astronomical unit`
   `REAL cgs astronomical unit`
   Length of one astronomical unit au.

2. `REAL mksa gravitational constant`
   `REAL cgs gravitational constant`
   Gravitational constant G.

3. `REAL mksa light year`
   `REAL cgs light year`
   Distance of one light-year ly.

4. `REAL mksa parsec`
   `REAL cgs parsec`
   Distance of one parsec pc.

5. `REAL mksa grav accel`
   `REAL cgs grav accel`
   Gravitational acceleration on Earth g.

6. `REAL mksa solar mass`
   `REAL cgs solar mass`
   Mass of the Sun.

11.15.4 Atomic and nuclear physics

1. `REAL mksa electron charge`
   `REAL cgs electron charge`
   Charge of an electron e.

2. `REAL mksa electron volt`
   `REAL cgs electron volt`
   Energy of one electron volt eV.

3. `REAL mksa unified atomic mass`
   `REAL cgs unified atomic mass`
   Unified atomic mass amu.

4. `REAL mksa mass electron`
   `REAL cgs mass electron`
   Mass of an electron m_e.

5. `REAL mksa mass muon`
   `REAL cgs mass muon`
   Mass of a muon m_μ.

6. `REAL mksa mass proton`
   `REAL cgs mass proton`
   Mass of a proton m_p.

7. `REAL mksa mass neutron`
   `REAL cgs mass neutron`
   Mass of a neutron m_n.

8. `REAL num fine structure`
   Electromagnetic fine structure constant, α.

9. `REAL mksa rydberg`
   `REAL cgs rydberg`
   The Rydberg constant Ry in units of energy. This is related to the Rydberg inverse wavelength R by $Ry = hcr$.

10. `REAL mksa bohr radius`
    `REAL cgs bohr radius`
    Bohr radius a_0.

11. `REAL mksa angstrom`
    `REAL cgs angstrom`
    Length of one Ångstrom.

12. `REAL mksa barn`
    `REAL cgs barn`
    Area of one barn.

13. `REAL mksa bohr magneton`
    `REAL cgs bohr magneton`
    Bohr magneton μ_b.

14. `REAL mksa nuclear magneton`
    `REAL cgs nuclear magneton`
    Nuclear magneton μ_n.

15. `REAL mksa electron magnetic moment`
    `REAL cgs electron magnetic moment`
    Absolute value of the magnetic moment of an electron μ_e. the physical magnetic moment of an electron is negative.

16. `REAL mksa proton magnetic moment`
    `REAL cgs proton magnetic moment`
    Magnetic moment of a proton μ_p.

11.15.5 Time

1. `REAL mksa minute`
   `REAL cgs minute`
   Number of seconds in one minute.

2. `REAL mksa hour`
   `REAL cgs hour`
   Number of seconds in one hour.

3. `REAL mksa day`
   `REAL cgs day`
   Number of seconds in one day.

4. `REAL mksa week`
   `REAL cgs week`
   Number of seconds in one week.

11.15.6 Imperial units

1. `REAL mksa inch`
   `REAL cgs inch`
   Length of one inch.

2. `REAL mksa foot`
   `REAL cgs foot`
   Length of one foot.

3. `REAL mksa yard`
   `REAL cgs yard`
   Length of one yard.

4. `REAL mksa mile`
   `REAL cgs mile`
   Length of one mile.

5. `REAL mksa mil`
   `REAL cgs mil`
   Length of one mil (1/1000th of an inch).

11.15.7 Nautical units

1. `REAL mksa nautical mile`
   `REAL cgs nautical mile`
   Length of one nautical mile.

2. `REAL mksa fathom`
   `REAL cgs fathom`
   Length of one fathom.

3. `REAL mksa knot`
   `REAL cgs knot`
   Speed of one knot.

11.15.8 Volume

1. `REAL mksa acre`
   `REAL cgs acre`
   Area of one acre.

2. `REAL mksa liter`
   `REAL cgs liter`
   Volume of one liter.

3. `REAL mksa us gallon`
   `REAL cgs us gallon`
   Volume of one us gallon.

4. `REAL mksa canadian gallon`
   `REAL cgs canadian gallon`
   Volume of one canadian gallon.

5. `REAL mksa uk gallon`
   `REAL cgs uk gallon`
   Volume of one uk gallon.

6. `REAL mksa quart`
   `REAL cgs quart`
   Volume of one quart.

7. `REAL mksa pint`
   `REAL cgs pint`
   Volume of one pint.

11.15.9 Mass and weight

1. `REAL mksa pound mass`
   `REAL cgs pound mass`
   Mass of one pound.

2. `REAL mksa ounce mass`
   `REAL cgs ounce mass`
   Mass of one ounce.

3. `REAL mksa ton`
   `REAL cgs ton`
   Mass of one ton.

4. `REAL mksa metric ton`
   `REAL cgs metric ton`
   Mass of one metric ton (1000 kg).

5. `REAL mksa uk ton`
   `REAL cgs uk ton`
   Mass of one uk ton.

6. `REAL mksa troy ounce`
   `REAL cgs troy ounce`
   Mass of one troy ounce.

7. `REAL mksa carat`
   `REAL cgs carat`
   Mass of one carat.

8. `REAL mksa gram force`
   `REAL cgs gram force`
   Force of one gram weight.

9. `REAL mksa pound force`
   `REAL cgs pound force`
   Force of one pound weight.

10. `REAL mksa kilopound force`
    `REAL cgs kilopound force`
    Force of one kilopound weight.

11. `REAL mksa poundal`
    `REAL cgs poundal`
    Force of one poundal.

11.15.10 Thermal energy and power

1. `REAL mksa calorie`
   `REAL cgs calorie`
   Energy of one Calorie.

2. `REAL mksa btu`
   `REAL cgs btu`
   Energy of one British Thermal Unit.

3. `REAL mksa therm`
   `REAL cgs therm`
   Energy of one therm.

4. `REAL mksa horsepower`
   `REAL cgs horsepower`
   Power of one horsepower.

11.15.11 Pressure

1. `REAL mksa bar`
   `REAL cgs bar`
   Pressure of one Bar.
2. `REAL mksa std atmosphere`
   `REAL cgs std atmosphere`
   Pressure of one standard atmosphere.
3. `REAL mksa torr`
   `REAL cgs torr`
   Pressure of one Torricelli.
4. `REAL mksa meter of mercury`
   `REAL cgs meter of mercury`
   Pressure of one meter of mercury.
5. `REAL mksa inch of mercury`
   `REAL cgs inch of mercury`
   Pressure of one inch of mercury.
6. `REAL mksa inch of water`
   `REAL cgs inch of water`
   Pressure of one inch of water.
7. `REAL mksa psi`
   `REAL cgs psi`
   pressure of one pound per square inch.

11.15.12 Viscosity

1. `REAL mksa poise`
   `REAL cgs poise`
   Dynamic viscosity of one Poise.
2. `REAL mksa stokes`
   `REAL cgs stokes`
   Kinematic viscosity of one Stokes.

11.15.13 Light and illumination

1. `REAL mksa stilb`
   `REAL cgs stilb`
   Luminance of one stilb.

2. `REAL mksa lumen`
   `REAL cgs lumen`
   Luminous flux of one lumen.

3. `REAL mksa lux`
   `REAL cgs lux`
   Illuminance of one lux.

4. `REAL mksa phot`
   `REAL cgs phot`
   Illuminance of one phot.

5. `REAL mksa footcandle`
   `REAL cgs footcandle`
   Illuminance of one footcandle.

6. `REAL mksa lambert`
   `REAL cgs lambert`
   Luminance of one lambert.

7. `REAL mksa footlambert`
   `REAL cgs footlambert`
   Luminance of one footlambert.

11.15.14 Radioactivity

1. `REAL mksa curie`
   `REAL cgs curie`
   Activity of one Curie.

2. `REAL mksa roentgen`
   `REAL cgs roentgen`
   Exposure of one Röntgen.

3. `REAL mksa rad`
   `REAL cgs rad`
   Absorbed dose of one Rad.

11.15.15 Force and energy

1. `REAL mksa newton`
   `REAL cgs newton`
   SI unit of force, one Newton.

2. `REAL mksa dyne`
   `REAL cgs dyne`
   Force of one Dyne.

3. `REAL mksa joule`
   `REAL cgs joule`
   SI unit of energy, one Joule.

4. `REAL mksa erg`
   `REAL cgs erg`
   Energy of one Erg.

11.15.16 Prefixes

1. `REAL num yotta`
   10^{24}

2. `REAL num zetta`
   10^{21}

3. `REAL num exa`
   10^{18}

4. `REAL num peta`
   10^{15}

5. `REAL num tera`
   10^{12}

6. `REAL num giga`
   10^{9}

7. `REAL num mega`
   10^{6}

8. `REAL num kilo`
   10^{3}

9. `REAL num milli`
   10^{-3}

10. `REAL num micro`
    10^{-6}

11. `REAL num nano`
    10^{-9}

12. `REAL num pico`
    10^{-12}

13. `REAL num femto`
    10^{-15}

14. `REAL num atto`
    10^{-18}

15. `REAL num zepto`
    10^{-21}

16. `REAL num yocto`
    10^{-24}

11.15.17 Printer units

1. `REAL mksa point`
   `REAL cgs point`
   Length of one printer's point $1/72$".

2. `REAL mksa texpoint`
   `REAL cgs texpoint`
   Length of one TEX point $1/72.27$".

11.16 Linux extensions

a68g maps to Linux as to provide basic means with which a program can

1. access command line arguments,
2. access directory - and file information,
3. access environment variables,
4. access system error information,

5. start child processes and communicate with them by means of pipes. *a68g* transput procedures as `getf` or `putf` operate on pipes,

6. send a request to a HTTP server, fetch web page contents, and match regular expressions in a string.

11.16.1 General definitions

1. `MODE PIPE = STRUCT (REF FILE read, write, INT pid)`
   Used to set up communication between processes. See for instance `execve child pipe`.

2. `PROC utc time = [] INT`
   Returns UTC calendar time, or an empty row if the function fails. Respective array elements are year, month, day, hours, minutes, seconds, day of week and daylight-saving-time flag.

3. `PROC local time = [] INT`
   Returns local calendar time, or an empty row if the function fails. Respective array elements are year, month, day, hours, minutes, seconds, day of week and daylight-saving-time flag.

4. `PROC argc = INT`
   Returns the number of command-line arguments.

5. `PROC argv = (INT k) STRING`
   Returns command-line argument `k` or an empty string when `k` is not valid.

6. `PROC get env = (STRING var) STRING`
   Returns the value of environment variable `var`.

For accessing system error information through `errno` next routines are available:

1. `PROC reset errno = VOID`
   Resets the global error variable `errno`.

2. `PROC errno = INT` Returns the global error variable `errno`.

3. `PROC strerror = (INT errno) STRING`
   Returns a string describing the error code passed as `errno`.

11.16.2 File system

1. `PROC file is block device = (STRING file) BOOL`
   Returns whether `file` is a block device. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

2. `PROC file is char device = (STRING file) BOOL`
   Returns whether `file` is a character device. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

3. `PROC file is directory = (STRING file) BOOL`
   Returns whether `file` is a directory. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

4. `PROC file is regular = (STRING file) BOOL`
   Returns whether `file` is a regular file. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

5. `PROC file is fifo = (STRING file) BOOL`
   Returns whether `file` is a first-in-first-out file, for instance a pipe. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

6. `PROC file is link = (STRING file) BOOL`
   Returns whether `file` is a first-in-first-out file, for instance a pipe. If the file does not exists or if an error occurs, `FALSE` is returned and `errno` is set.

7. `PROC file mode = (STRING file) BITS`
   Yields file mode of file `file`. If the file does not exists or if an error occurs, `2r0` is returned and `errno` is set.

8. `PROC get pwd = STRING`
   Returns the current working directory.

9. `PROC set pwd = (STRING dir) VOID`
   Sets the current working directory to `dir`, which must be a valid directory name.

10. `PROC get directory = (STRING dir) [] STRING`
    Retrieves the filenames from directory `dir`, which must be a valid directory name. Note that the order of entries in the returned `[] STRING` is arbitrary. Example:

```
PROC ls = (STRING dir name) VOID:
     IF is directory (dir name)
     THEN printf (($lg$, SORT get directory (dir name)))
     ELSE print (("’", dir name, "’ is not a directory"))
     FI
```

11.16.3 Processes

1. `PROC fork = INT`
   Creates a child process that differs from the parent process only in its `PID` and `PPID`, and that resource utilisations are set to zero. File locks and pending signals are not inherited. Under Linux, fork is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. On success, the `PID` of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and `errno` will be set appropriately.

2. `PROC wait pid = (INT pid) VOID`
   Waits until child process `pid` ends.

3. `PROC execve =`
   `(STRING filename, [] STRING argv, [] STRING envp) INT`
   Executes `filename` that must be either a binary executable, or a script starting with a line of the form

   ```
   #! interpreter [argv]
   ```

   In the latter case, the interpreter must be a valid path-name for an executable which is not itself a script, which will be invoked as `interpreter [arg] filename`. Argument `argv` is a row of argument strings that is passed to the new program. Argument `envp` is a row of strings, with elements conventionally of the form `KEY=VALUE`, which are passed as environment to the new program. The argument vector and environment can be accessed by the called program's `main` function, when it is defined as

   ```
   int main(int argc, char *argv[], char *envp[])
   ```

   The procedure does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's `PID`, and any open file descriptors that are not set to close on exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour. The `SIGCHLD` signal (when set to `SIG_IGN`) may or may not be reset to `SIG_DFL`.

4. `PROC execve child =`
   `(STRING filename, [] STRING argv, [] STRING envp) INT`
   Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`.

5. `PROC execve output =`
   `(STRING filename, [] STRING argv, [] STRING envp, REF STRING output)`

`INT`
Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`. Output written to `stdout` by the child process is collected in a string which is assigned to `output` unless `output` is `NIL`. The routine yields the return code of the child process on success or -1 on failure. Next code fragment shows how to execute a command in the shell and retrieve output from both `stdout` and `stderr`:

```
PROC get output = (STRING cmd) VOID:
     IF STRING sh cmd = "{ " + cmd + " ; } 2>&1";
        STRING output;
        execve output ("/bin/sh", ("sh", "-c", sh cmd), "", output) >= 0
     THEN print (output)
     FI;

get output ("ps | grep a68g")
```

6. `PROC execve child pipe =`
   `(STRING filename, [] STRING argv, [] STRING envp) PIPE`
   Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`. The child process redirects standard input and standard output to the pipe that is delivered to the parent process. Therefore, the parent can write to the `write` field of the pipe, and this data can be read by the child from standard input. The child can write to standard output, and this data can be read by the parent process from the `read` field of the pipe. The files in the pipe are opened. The channels assigned to the read - and write end of the pipe are `standin channel` and `standout channel`, respectively. This procedure calls `execve` in the child process, for a description of the arguments refer to that procedure. For example:

```
PROC eop handler = (REF FILE f) BOOL:
     BEGIN put (stand error, ("end of pipe", new line));
           GO TO end
     END;

PIPE p = execve child pipe ("/bin/cat", ("cat", program idf), "");

IF pid OF p < 0
THEN put (stand error, ("pipe not valid: ", strerror (errno)));
     stop
FI;

on logical file end (read OF p, eop handler);

DO STRING line;
   get (read OF p, (line, newline));
   put (stand out, ("From pipe: """, line, """", newline))
OD;

end:
```

```
close (read OF p);
close (write OF p)
```

11.16.4 Fetching web page contents and sending requests

1. `PROC http content =`
   `(REF STRING content, STRING domain, STRING path, INT port) INT`
   Gets content from web page specified by `domain` and `path`, and stores it in `content`. Both `domain` and `path` must be properly `MIME` encoded. The routine connects to server `domain` using TCP and sends a straightforward `GET` request for `path` using HTTP 1.0, and reads the reply from the server. If port number `port` is 0 the default port number (80) will be used. The procedure returns 0 on success or an appropriate error code upon failure. For example:

```
PROC good page = (REF STRING page) BOOL:
     IF grep in string("^HTTP/[0-9.]* 200", page, NIL, NIL) = 0
     THEN TRUE
     ELSE IF INT start, end;
             grep in string("^HTTP/[0-9.]* [0-9]+ [a-zA-Z ]*", page,
                            start, end) = 0
          THEN print (page[start : end])
          ELSE print ("unknown error retrieving page")
          FI;
          FALSE
     FI;

IF STRING reply;
   INT rc = http content (reply, "www.gnu.org", "http://www.gnu.org/", 0);
   rc = 0 AND good page (reply)
THEN print (reply)
ELSE print (strerror (rc))
FI
```

2. `PROC tcp request =`
   `(REF STRING reply, STRING domain, STRING request, INT port) INT`
   Send request `request` to server specified by `domain`, and stores the server's reply in `reply`. Both `domain` and `request` must be properly `MIME` encoded. The routine connects to server `domain` using TCP. If port number `port` is 0 the default port number (80) will be used. The procedure returns 0 on success or an appropriate error code upon failure. For example:

```
STRING reply;
IF INT rc = tcp request (reply, "www.kernel.org",
            "GET / HTTP/1.0" + newline char +
            "Range: bytes=0-1023" + 2 * newline char, 0);
   rc = 0
THEN print (reply)
```

```
ELSE print (strerror (rc))
FI;
```

11.17 Regular expressions in string manipulation

1. `PROC grep in string =`
   `(STRING pattern, string, REF INT start, end) INT`
   Matches `pattern` with `string` and assigns first character position (counting from 1) and last character position of the widest leftmost matching sub string to `start` and `end` when they are not `NIL`. The routine yields 0 on match, 1 on no-match, 2 on out-of-memory error and 3 on other errors. If the string contains `newline character`, it is considered as end-of-line (”$”) and the following character is considered as start-of-line (”ˆ”). Compare this function with AWK function `match`.

2. `PROC sub in string =`
   `(STRING pattern, replacement, REF STRING string) INT`
   Matches `pattern` with `string` and replaces the widest leftmost matching sub string with `replacement`. The routine yields 0 on match, 1 on no-match, 2 on out-of-memory error and 3 on other errors. If the string contains `newline character`, it is considered as end-of-line (”$”) and the following character is considered as start-of-line (”ˆ”). Compare this function with AWK function `sub`. An equivalent to AWK function `gsub` can be programmed easily in Algol 68 Genie:

```
PROC awk gsub = (STRING pattern, replacement, REF STRING string) INT:
     BEGIN INT return code;
           DO return code := sub in string (pattern, replacement, string);
              UNTIL return code /= 0
           OD;
           return code
     END
```

`grep in string` and `sub in string` employ extended POSIX syntax for regular expressions. Traditional UNIX regular expression syntax is obsoleted by POSIX (but is still widely used for the purposes of backwards compatibility). POSIX extended regular expressions are similar to traditional UNIX regular expressions. Most characters are treated as literals that match themselves: ”a” matches ”a”, ”(bc” matches ”(bc”, et cetera. Metacharacters ”(”, ”)”, ”[”, ”]”, ””, ””, ”.”, ”*”, ”?”, ”+”, ”ˆ” and ”$” are used as special symbols that have to be marked by a preceding ”ẅhen they are to be matched literally.

11.17.1 Definition of metacharacters

1. `.`
   Matches any single character.
2. `*`
   Matches the preceding expression zero or more times. For example, "[xyz]*" matches "", "x", "y", "zx", "zyx", and so on.
3. `+`
   Matches the preceding expression one or more times - "ba+" matches "ba", "baa", "baaa" et cetera.
4. `?`
   Matches the preceding expression zero or one times - "ba?" matches "b" or "ba".
5. {`x, y` } Matches the preceding expression at least `x` and not more than `y` times. For example, "a{3, 5 }" matches "aaa", "aaaa" or "aaaaa". Note that this is not implemented on all platforms.
6. `|`
   The choice (or set union) operator: match either the expression before or the expression after the operator - "abc—def" matches "abc" or "def".
7. `[ ]`
   Matches characters that contained within [ ]. For example, [abc] matches "a", "b", or "c", and [a-z] matches any lower-case letter. These can be mixed: [abcq-z] matches a, b, c, q, r, s, t, u, v, w, x, y, z, and so does [a-cq-z]. A "-" character is a literal only if it is the first or last character within the brackets: [abc-] or [-abc]. To match an "[" or "]" character, put the closing bracket first in the enclosing square brackets: [][ab] matches "]", "[", "a" or "b".
8. `[ˆ]`
   Matches a single character that is not contained within [ ]. For example, [ˆabc] matches any character other than "a", "b", or "c". [ˆa-z] matches any single character that is not a lower-case letter. As with [ ], these can be mixed.
9. ˆ
   Matches start of the line.
10. `$`
    Matches end of the line.
11. `( )`
    Matches a sub-expression. A sub-expression followed by "*" is invalid on most platforms.

Since many ranges of characters depend on the chosen locale setting (e.g., in some settings letters are organised as abc..yzABC..YZ while in some others as aAbBcC..yYzZ) the POSIX standard defines categories of characters as shown in the following table:

Category	Compare to	Matches
`[:upper:]`	`[A-Z]`	Upper-case letters
`[:lower:]`	`[a-z]`	Lower-case letters
`[:alpha:]`	`[A-Za-z]`	Upper- and lower-case letters
`[:alnum:]`	`[A-Za-z0-9]`	Digits, upper- and lower-case letters
`[:digit:]`	`[0-9]`	Digits
`[:xdigit:]`	`[0-9A-Fa-f]`	Hexadecimal digits
`[:punct:]`	`[.,!?:...]`	Punctuation marks
`[:blank:]`	`[ \t]`	Space and Tab
`[:space:]`	`[ \t\n\r\f\v]`	Typographical display features
`[:cntrl:]`	—	Control characters
`[:graph:]`	`[ˆ \t\n\r\f\v]`	Printed characters
`[:print:]`	`[ˆ \t\n\r\f\v]`	Printed characters and space

Examples:

1. `".at"` matches any three-character string ending with `at`,
2. `"[hc]at"` matches `hat` and `cat`,
3. `"[ˆ]at"` matches any three-character string ending with `at` and not beginning with `b`,
4. `"ˆ[hc]at"` matches `hat` and `cat` but only at the beginning of a line,
5. `"[hc]at\$"` matches `hat` and `cat` but only at the end of a line,
6. `"[hc]+at"` matches with ”hat”, ”cat”, ”hhat”, ”chat”, ”hcat”, ”ccchat” et cetera,
7. `"[hc]?at"` matches ”hat”, ”cat” and ”at”,
8. `"([cC]at)`([dD]og)”— matches ”cat”, ”Cat”, ”dog” and ”Dog”,
9. `[[:upper:]ab]` matches upper-case letters and ”a”, ”b”.

11.18 Curses support

On systems that have installed the curses library, and most systems do, *a68g* can link with that library to extend the standard-prelude. Curses support is very basic.

1. `PROC curses start = VOID`
   Starts curses support.
2. `PROC curses end = VOID`
   Stops curses support.

3. `PROC curses clear = VOID`
   Clears the screen.

4. `PROC curses refresh = VOID`
   Refreshes the screen (by writing all changes since the last call to this routine).

5. `PROC curses getchar = CHAR`
   Check if a key was pressed on the keyboard. If so, the routine returns the corresponding character, if not, on UNIX it returns `null char`.

6. `PROC curses putchar = (CHAR ch) VOID`
   Writes `ch` at the current cursor position.

7. `PROC curses move = (INT line, column) VOID`
   Sets the cursor position to `line` and `column`.

8. `PROC curses lines = INT`
   Returns the number of lines on the screen.

9. `PROC curses columns = INT`
   Returns the number of columns on the screen.

11.19 PostgreSQL support

PostgreSQL (*http://www.postgresql.org*) is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features. PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs)

1. A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called postmaster.

2. The user's client (front-end) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialised database maintenance tool.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts (”forks”) a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original postmaster process. Thus, the postmaster is always running, waiting for client connections, whereas client and associated server processes come and go.

The routines in this section enable basic client interactions from Algol 68 with a PostgreSQL backend server. These routines allow client programs to pass queries to a PostgreSQL backend server and to receive the results of these queries.

Algol 68 Genie uses `libpq`, the C application programmer’s interface to PostgreSQL (version 8.1.4 or compatible). The `libpq` library is also used in other PostgreSQL application interfaces, including those for C++, Perl, Python, Tcl and ECPG.

Connection to a database server is established through a `REF FILE` variable. String results are communicated through a `REF STRING` buffer that is associated with the `REF FILE` variable upon establishing a connection with a PostgreSQL server. This allows for easy mode conversion of text-formatted data from tables using `get`. Currently, Algol 68 Genie does not support retrieving data in binary format.

Next routines return 0 or positive on success, -1 when no connection was established, -2 when no result was available, or -3 on other errors.

11.19.1 Connecting to a server

1. `PROC pq connect db =`
   `(REF FILE f, STRING conninfo, REF STRING buffer) INT`
   Connect to a PostgreSQL backend server and open a new database connection in a way specified by `conninfo`, associate the connection with file `f` and associate `buffer` with file `f`.

   The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by white space. Each parameter setting is in the form keyword = value. Spaces around the equal sign are optional. To write an empty value or a value containing spaces, surround it with single quotes, e.g., keyword = ’a value’. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.

   The currently recognised parameter keywords are:

   `host`
   Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default behaviour when host is not specified is to connect to a Unix-domain socket in /tmp (or whatever socket directory was specified when PostgreSQL was built). On machines without Unix-domain sockets, the default is

to connect to localhost.

`hostaddr`
Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a non-empty string is specified for this parameter. Using hostaddr instead of host allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies: If host is specified without hostaddr, a host name lookup occurs. If hostaddr is specified without host, the value for hostaddr gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both host and hostaddr are specified, the value for hostaddr gives the remote address; the value for host is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if `libpq` is passed a host name that is not the name of the machine at hostaddr.) Also, host rather than hostaddr is used to identify the connection in `~/.pgpass`. Without either a host name or host address, `libpq` will connect using a local Unix-domain socket; or on machines without Unix-domain sockets, it will attempt to connect to localhost.

`port`
Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`
The database name. Defaults to be the same as the user name.

`user`
PostgreSQL user name to connect as. Defaults to be the same as the operating system name of the user running the application.

`password`
Password to be used if the server demands password authentication.

`connect_timeout`
Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a time-out of less than 2 seconds.

`options`
Command-line options to be sent to the server.

`sslmode`
This option determines whether or with what priority an SSL connection will be negotiated with the server. There are four modes: disable will attempt only an unencrypted SSL connection; allow will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; prefer (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; require will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option require will

cause an error, while options allow and prefer will be accepted but `libpq` will not in fact attempt an SSL connection.

`requiressl`
This option is deprecated in favour of the sslmode setting. If set to 1, an SSL connection to the server is required (this is equivalent to sslmode require). `libpq` will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), `libpq` will negotiate the connection type with the server (equivalent to sslmode prefer). This option is only available if PostgreSQL is compiled with SSL support.

`krbsrvname`
Kerberos service name to use when authenticating with Kerberos 5. This must match the service name specified in the server configuration for Kerberos authentication to succeed.

`service`
Service name to use for additional parameters. It specifies a service name in
`pg_service.conf`
that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See
`share/pg_service.conf.sample`
in the installation directory for information on how to set up the file. If any parameter is unspecified, then the corresponding environment variable (see Section 28.11) is checked. If the environment variable is not set either, then the indicated built-in defaults are used. The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` if no value is directly specified by the calling code. These are useful to avoid hard-coding database connection information into simple client applications, for example.

`PGHOST`
`PGHOST` sets the database server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is then the name of the directory in which the socket file is stored (in a default installation setup this would be `/tmp`).

`PGHOSTADDR`
`PGHOSTADDR` specifies the numeric IP address of the database server. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead. See the documentation of these parameters, under `PQconnectdb` above, for details on their interaction. When neither `PGHOST` nor `PGHOSTADDR` is set, the default behaviour is to connect using a local Unix-domain socket; or on machines without Unix-domain sockets, `libpq` will attempt to connect to localhost.

`PGPORT`
`PGPORT` sets the TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL server.

`PGDATABASE`
`PGDATABASE` sets the PostgreSQL database name.

`PGUSER`
`PGUSER` sets the user name used to connect to the database.

`PGPASSWORD`
`PGPASSWORD` sets the password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via `ps`); instead consider using the `/.pgpass` file.

`PGPASSFILE`
`PGPASSFILE` specifies the name of the password file to use for lookups. If not set, it defaults to `/.pgpass`.

`PGSERVICE`
`PGSERVICE` sets the service name to be looked up in `pg_service.conf`. This offers a shorthand way of setting all the parameters.

`PGREALM`
`PGREALM` sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, `libpq` applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the server.

`PGOPTIONS`
`PGOPTIONS` sets additional run-time options for the PostgreSQL server.

`PGSSLMODE`
`PGSSLMODE` determines whether and with what priority an SSL connection will be negotiated with the server. There are four modes: disable will attempt only an unencrypted SSL connection; allow will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; prefer (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; require will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option require will cause an error, while options allow and prefer will be accepted but `libpq` will not in fact attempt an SSL connection.

`PGREQUIRESSL`
`PGREQUIRESSL` sets whether or not the connection must be made over SSL. If set to 1, `libpq` will refuse to connect if the server does not accept an SSL connection (equivalent to sslmode prefer). This option is deprecated in favour of the sslmode setting, and is only available if PostgreSQL is compiled with SSL support.

`PGKRBSRVNAME`
`PGKRBSRVNAME` sets the Kerberos service name to use when authenticating with Kerberos 5.

`PGCONNECT_TIMEOUT`
`PGCONNECT_TIMEOUT` sets the maximum number of seconds that `libpq` will wait when attempting to connect to the PostgreSQL server. If unset or set to zero, `libpq` will wait indefinitely. It is not recommended to set the time-out to less than 2 seconds.

2. `PROC pq finish = (REF FILE f) INT`
   Closes the connection to the server. Also frees memory used. Note that even if the server connection attempt fails, the application should call `pq finish` to free memory used. The connection must not be used again after `pq finish` has been called.

3. `PROC pq reset = (REF FILE f) INT`
   Resets the communication channel to the server. This function will close the connection to the server and attempt to re-establish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

4. `PROC pq parameter status = (REF FILE f, STRING parameter) INT`
   Assigns current parameter value to the string associated with `f`. Certain parameter values are reported by the server automatically at connection start-up or whenever their values change. `pq parameter status` can be used to interrogate these settings. It returns the current value of a parameter if known, or an empty string if the parameter is not known.

   Parameters reported as of the current release include `server_version`,
   `server_encoding`,
   `client_encoding`,
   `is_superuser`,
   `session_authorization`,
   `DateStyle`,
   `TimeZone`,
   `integer_datetimes`, and
   `standard_conforming_strings`.

   Note that `server_version`,
   `server_encoding` and
   `integer_datetimes`
   cannot change after start-up.

   Pre-3.0-protocol servers do not report parameter settings, but `libpq` includes logic to obtain values for `server_version` and `client_encoding` anyway. Note that on a pre-3.0 connection, changing `client_encoding` via `SET` after connection start-up will not be reflected by `pq parameter status`.

   If no value for `standard_conforming_strings` is reported, applications may assume it is `false`, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter may be taken as an indication that the escape string syntax (`E'...'`) is accepted.

11.19.2 Sending queries and retrieving results

1. `PROC pq exec = (REF FILE f, STRING query) INT`
   Submits a command to the server and waits for the result. A zero result will generally be returned except in out-of-memory conditions or serious errors such as inability to send the command to the server. If a zero result is returned, it should be treated like a fatal error. Use `pq error message` to get more information about such errors.

   It is allowed to include multiple SQL commands (separated by semicolons) in the command string. Multiple queries sent in a single `pq exec` call are processed in a single transaction, unless there are explicit BEGIN/COMMIT commands included in the query string to divide it into multiple transactions. Note however that the returned PGresult structure describes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned PGresult describes the error condition.

2. `PROC pq ntuples = (REF FILE f) INT`
   Returns the number of rows (tuples) in the query result.

3. `PROC pq nfields = (REF FILE f) INT`
   Returns the number of columns (fields) in each row of the query result.

4. `PROC pq fname = (REF FILE f, INT index) INT`
   Returns the column name associated with the given column number. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element).

5. `PROC pq fnumber = (REF FILE f, INT index) INT`
   Returns the column number associated with the given column name. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element). The given name is treated like an identifier in an SQL command, that is, it is down-cased unless quoted.

6. `PROC pq fformat = (REF FILE f, INT index) INT`
   Returns the format code indicating the format of the given column. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element). Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.) Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element).

7. `PROC pq get is null = (REF FILE f, INT row, column) INT`
   Tests a field for a null value. Row and column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element). This function returns 1 if the field is null and 0 if it contains a non-null value.

8. `PROC pq get value = (REF FILE f, INT row, column) INT`
   Assigns a single field value (in text format) of one row to the string associated with `f`.

Row and column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscripts the first element).

Currently, `pq get value` does not support retrieving data in binary format.

An empty string is returned if the field value is null.

File `f` is reset, hence after a call to `pq get value`, conversion of text data to other data types than row-of-character can be easily accomplished using `get`.

9. `PROC pq cmd status = (REF FILE f) INT`
   Assigns the command status tag of the last SQL command sent to `f` to the string associated with `f`. Commonly this is just the name of the command, but it may include additional data such as the number of rows processed.

10. `PROC pq cmd tuples = (REF FILE f) INT`
    Assigns the number of rows affected by the last SQL command sent to `f` to the string associated with `f`, following the execution of an `INSERT`, `UPDATE`, `DELETE`, `MOVE`, or `FETCH` statement.

11.19.3 Connection status information

1. `PROC pq error message = (REF FILE f) INT`
   Assigns the error message most recently generated by an operation on the connection to the string associated with `f`.

2. `PROC pq result error message = (REF FILE f) INT`
   Assigns the error message associated with the command (or an empty string if there was no error) to the string associated with `f`. Immediately following a `pq exec call`, `pq error message` (on the connection) will return the same string as `pq result error message` (on the result). However, a query result will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done.

3. `PROC pq db = (REF FILE f) INT`
   Assigns the database name of the connection to the string associated with `f`.

4. `PROC pq user = (REF FILE f) INT`
   Assigns the user name of the connection to the string associated with `f`.

5. `PROC pq pass(REF FILE f) INT`
   Assigns the password of the connection to the string associated with `f`.

6. `PROC pq host = (REF FILE f) INT`
   Assigns the server host name to the string associated with `f`.

7. `PROC pq port = (REF FILE f) INT`
   Assigns the port of the connection to the string associated with `f`.

8. `PROC pq options = (REF FILE f) INT`
   Assigns the command-line options passed in the connection request to the string associated with `f`.

9. `PROC pq protocol version = (REF FILE f) INT`
   Returns the front-end/back-end protocol being used. Applications may wish to use this to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). This will not change after connection start-up is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by libpq.)

10. `PROC pq server version = (REF FILE f) INT`
    Returns an integer representing the backend version. Applications may use this to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 7.4.2 will be returned as 70402, and version 8.1 will be returned as 80100 (leading zeroes are not shown). Zero is returned if the connection is bad

11. `PROC pq socket = (REF FILE f) INT`
    Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection setup or reset.)

12. `PROC pq backend pid = (REF FILE f) INT`
    Returns the process ID (PID) of the backend server process handling this connection. The backend PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying backend process). Note that the PID belongs to a process executing on the database server host, `not` the local host.

11.19.4 Behaviour in threaded programs

`libpq` is re-entrant and thread-safe if the configure command-line option

```
--enable-thread-safety
```

was used when the PostgreSQL distribution was built.

One restriction is that no two threads attempt to manipulate the same PGconn object at the same time. In particular, you cannot issue concurrent commands from different threads through the same connection object. If you need to run concurrent commands, use multiple connections. PGresult objects are read-only after creation, and so can be passed around freely between threads.

In Algol 68 Genie, this means that in a parallel-clause, two concurrent units should not manipulate a same `FILE` variable that is used to hold a connection to a database. Use multiple `FILE` variables to set up multiple connections.

For more information please refer to the PostgreSQL documentation.

11.19.5 Example code

```
#
This example assumes a database "mydb" owned by postgres user "marcel",
and a table "weather" constructed following the example in the
PostgreSQL 8.1 manual, chapter 2.
#

FILE z;     # Holds the connection #
STRING str; # The string associated with ’z’ to do IO with the server #

# Connect to the server #

IF pq connectdb (z, "dbname=mydb user=marcel", str) ~= 0
THEN print ((pq result error message (z); str));
     exit
ELSE printf (($"protocol="g(0)x"server="g(0)x"socket="g(0)x"pid="g(0)/$,
              pq protocol version (z),
              pq server version (z),
              pq socket (z),
              pq backend pid (z)))
FI;

# Get the complete table #

IF pq exec (z, "SELECT * FROM weather") ~= 0
THEN print ((pq result error message (z); str));
     exit
FI;

printf (($"tuples="g(0)x"fields="g(0)/$,
         pq ntuples (z), pq nfields (z)));

# Print column names #

FOR i TO pq nfields (z)
DO IF pq fname (z, i) = 0
   THEN print ((str, blank))
   FI
OD;
```

```
newline (standout);

# Print row entries #

FOR i TO pq ntuples (z)
DO FOR k TO pq nfields (z)
   DO IF pq getisnull (z, i, k) ~= 0
      THEN print ("(null)")
      ELIF pq getvalue (z, i, k) = 0
      THEN print ((str, blank))
      ELSE print ("error")
      FI
   OD;
   newline (standout)
OD;

# Print average high temperature, convert mode as a demo #

IF pq exec (z, "SELECT avg (temp_hi) FROM weather") ~= 0
THEN print ((pq result error message (z); str));
     exit
FI;

printf (($"tuples="g(0)x"fields="g(0)/$, pq ntuples (z), pq nfields (z)));

IF pq getisnull (z, 1, 1) ~= 0
THEN print ("(null)")
ELIF pq getvalue (z, 1, 1) = 0
THEN print ((REAL avg temp hi; get (z, avg temp hi); avg temp hi))
ELSE print ("error")
FI;

# Close connection to the server #
exit:
pq finish (z)
```

11.20 PCM Sounds

Algol 68 Genie has basic support for manipulating sounds. In *a68g* , `SOUND` values can be generated, read and written and the sampled sound data can be accessed and altered. You can for example write an application to concatenate sound files, or to filter a sound using Fourier transforms. Currently Algol 68 Genie supports linearly encoded PCM (for example used for CD audio). Should you wish to manipulate sound data in another format, then one option is to convert formats using a tool as for instance *SoX*.

11.20.1 Data structure and routines

a68g allows the declaration of values of mode `SOUND` that will contain a sound. The standard-prelude contains a declaration

```
MODE SOUND = STRUCT ( ... );
```

with structured fields that cannot be directly accessed. The actual sample data is stored in the heap. *a68g* takes care of proper copying of sample data in case of assignation of `SOUND` values, or manipulation of stowed objects containing `SOUND` values, et cetera. Currently next routines are declared for values of mode `SOUND`:

1. `PROC new sound = (INT resolution, rate, channels, samples) SOUND`
   Returns a sound value where `resolution` is the number of bits per sample (8, 16, ...), `rate` is the sample rate, `channels` is the number of channels (1 for mono, 2 for stereo) and `samples` is the number of samples per channel.

2. `OP RESOLUTION = (SOUND s) INT`
   Returns the number of bits in a sample of `s`.

3. `OP CHANNELS = (SOUND s) INT`
   Returns the number of channels of `s`.

4. `OP RATE = (SOUND s) INT`
   Returns the sample rate (as samples per second per channel) of `s`. Common values are 8000, 11025, 22050 or 44100.

5. `OP SAMPLES = (SOUND s) INT`
   Returns the number of samples per channel of `s`.

6. `PROC get sound = (SOUND s, INT channel, sample) INT`
   Returns the sample indexed by `channel` and `sample` of `s`. Note that under RIFF, samples with size up to 8 bits are normally unsigned while samples of larger size are signed.

7. `PROC set sound = (SOUND s, INT channel, sample, value) INT`
   Sets the sample indexed by `channel` and `sample` of `s` to `value`. Note that under RIFF, samples with size up to 8 bits are normally unsigned while samples of larger size are signed.

11.20.2 Transput of `SOUND` values

`SOUND` values can be read and written using standard-prelude transput routines. Formatted transput of `SOUND` values is undefined. Currently, Algol 68 Genie supports transput of files according to the RIFF PCM/WAVE format specification. Note that chunks other than `WAVE`, `fmt` or `data` are stripped upon reading, and do not appear upon writing.

An example of transput of a `SOUND` value is:

```
SOUND s;
FILE in, out;
open (in, "source.wav", standback channel);
open (out, "destination.wav", standback channel);
get bin (in, s);
put bin (out, s);
close (in);
close (out);
```

11.20.3 Playing sounds

Playing a sound depends on actual hardware details of a platform. Therefore, Algol 68 Genie does not have a standard procedure to play `SOUND` values. It is however possible to write a procedure to play a sound in Algol 68 Genie, making use of its extensions to print to pipes and to fork, as is demonstrated in next example that makes a `SOUND` value to be played by the `play` program that comes with *SoX*:

```
PR heap=256M PR # ... to play BIG audio tracks #

PROC play sound = (SOUND s) VOID:
     IF PIPE p = execve child pipe("/usr/bin/play",
                    ("play", "-t", "wav", "-q", "-"), "");
        pid OF p < 0
     THEN put(stand error, ("pipe not valid: ", strerror(errno)));
     ELSE put(write OF p, s); # put must wait for play to empty pipe #
          close(read OF p);
          close(write OF p)
     FI;

print("File: ");
STRING name = read string;
open(standin, name + ".wav", standin channel);
SOUND s;
read (s);
close(standin);
play sound(s)
```

11.20.4 Example application for `SOUND` values

Next application demonstrates how to program applications manipulating sound values. This particular example filters a sound by Fourier-transformation using routines described later in this chapter.

```
COMMENT
  Band-pass filter for a sound file on a second-per-second basis using FFT.
  Limitations:
  (1) sound duration must be a whole number of seconds
  (2) sample rate must be even
COMMENT

#  Get the sound file #
SOUND w, FILE in, out;
print ((newline, "File: "));
STRING fn = read string;
open (in, fn + ".wav", standback channel);
get bin (in, w);
close (in);
print ((newline, "rate=", RATE w));
print ((newline, "samples=", SAMPLES w));
print ((newline, "bits=", RESOLUTION w));
print ((newline, "channels=", CHANNELS w));
INT secs = SAMPLES w OVER RATE w;
print ((newline, "duration=", secs));
# If next number is large, FFT is inefficient #
print ((newline, "prime factors=", UPB prime factors (RATE w)));
# Get filter parameters #
print ((newline, "low frequency: "));
REAL low f = read real;
print ((newline, "high frequency: "));
REAL high f = read real;
# Apply a band filter.
  Not the most efficient code, but obvious #
FOR i TO secs
DO print ((newline, i, collections, garbage));
   FOR j TO CHANNELS w
   DO # Get data for this second #
      [RATE w] REAL samples;
      FOR k TO RATE w
      DO samples[k] := get sound (w, j, (i - 1) * RATE w + k)
      OD;
      # Forward transform to determine spectrum #
      [RATE w] COMPLEX spectrum := fft forward (samples);
      # Apply filter.
        Simplifications apply since the FFT is over one second,
        hence N*dt=1 and the frequencies are whole numbers #
      FOR m TO RATE w OVER 2 + 1
      DO # f2 is the corresponding negative frequency with f1 #
         INT f1 = m, f2 = (f1 = 1 | 1 | RATE w - f1 + 2);
         IF f1 < low f OR f1 > high f
         THEN spectrum[f1] := spectrum[f2] := 0
         FI
      OD;
      # Restore sample data for this second #
      samples := fft inverse (spectrum);
      FOR k TO RATE w
      DO set sound (w, j, (i - 1) * RATE w + k, ENTIER samples[k])
      OD
   OD
OD;
# Put the filtered sound file #
open (out, fn + ".fft.wav", standback channel);
put bin (out, w);
close (out)
```

Chapter 12

Example programs

12.1 Lucas numbers

```
CO Using refinements CO

determine first generation;
WHILE can represent next generation
DO calculate next generation;
   print next generation
OD.

   determine first generation:
      INT previous := 1, current := 3.

   can represent next generation:
      current <= max int - previous.

   calculate next generation:
      INT new = current + previous;
      previous := current;
      current := new.

   print next generation:
      printf (($lz","3z","3z","2z-d$, current,
               $xz","3z","3z","2z-d$, previous,
               $xd.n(real width - 1)d$, current / previous)).
```

12.2 Calendar

```
PROC weekday = (INT year, month, day) INT:
     # Day of the week by Zeller's Congruence algorithm from 1887 #
     BEGIN INT y := year, m := month, d := day, c;
           IF m <= 2
           THEN m +:= 12;
                y -:= 1
           FI;
           c := y OVER 100;
           y %*:= 100;
           (d + ((m + 1) * 26) OVER 10 + y + y OVER 4 + c OVER 4 - 2 * c) MOD 7
     END;

# Print a calendar for this year #

INT year = utc time[1];
FOR month TO 12
DO # Assign day numbers to weekdays in this month #
   INT days = (month | 31, 28 + (year MOD 4 = 0 AND year MOD 400 /= 0 | 1 | 0),
                       31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   [1 : days + 7] STRING s;
   FOR k TO UPB s
   DO s[k] := 3 * blank
   OD;
   FOR k TO days
   DO s[weekday (year, month, 1) + k] := whole (k, -3)
   OD;
   # Print this month #
   on format error (standout, (REF FILE f) BOOL: continue);
   print f (($llc("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")xg(0)l
                 "Sat Sun Mon Tue Wed Thu Fri"$, month, year));
   printf (($l7(aaax)$, s));
   continue: SKIP
OD
```

12.3 Whetstone benchmark

```
CO

Synthetic benchmark following Curnow & Wichmann.

Some time in the 1960's, NPL (UK) had a computer with an
Algol 60 implementation called "Whetstone translator-
interpreter", whence the name.

Some Algol 60 results (MWHIPS):
IBM 3090:       Algol 60 Compiler       5.0

Some Algol 68 results (MWHIPS):
```

```
ICL 1906A       A68R Compiler           0.6
I686 3GHz       A68G Interpreter        44

CO

[1 : 4] REAL e1,
REAL t, t1, t2, cpu1, time, x1, x2, x3, x4, x, y, z,
INT j, k, l, i, ii;

PROC pa = (REF [] REAL e) VOID:
     TO 6
     DO e[1] := (e[1] + e[2] + e[3] - e[4]) * t;
        e[2] := (e[1] + e[2] - e[3] + e[4]) * t;
        e[3] := (e[1] - e[2] + e[3] + e[4]) * t;
        e[4] := (- e[1] + e[2] + e[3] + e[4]) / t2
     OD;

PROC po = VOID:
     (e1[j] := e1[k]; e1[k] := e1[l]; e1[l] := e1[j]);

PROC p3 = (REF REAL x, y, z) VOID:
     (x := t * (x + y); y := t * (x + y); z := (x + y) / t2);

cpu1 := seconds;
INT max = 10;
FOR cycles TO max
DO # Initialise constants #

   t := 0.499975; t1 := 0.50025; t2 := 2.0;

   # If i = 10 we have one million whetstone instructions
     per cycle #

   i := 100; ii := i;
   INT n2 = 12 * i, n3 = 14 * i, n4 = 345 * i,  n6 = 210 * i,
       n7 = 32 * i, n8 = 899 * i, n9 = 616 * i, n11 = 93 * i;

   # MODULE 1. Simple identifiers #

   x1 := 1.0; x2 := x3 := x4 := -1.0;

   # MODULE 2. Array elements #

   e1[1] := 1.0; e1[2] := e1[3] := e1[4] := -1.0;

   TO n2
   DO e1[1] := (e1[1] + e1[2] + e1[3] - e1[4]) * t;
      e1[2] := (e1[1] + e1[2] - e1[3] + e1[4]) * t;
      e1[3] := (e1[1] - e1[2] + e1[3] + e1[4]) * t;
      e1[4] := (- e1[1] + e1[2] + e1[3] + e1[4]) * t
   OD;

   # MODULE 3. Array parameters #

   TO n3 DO pa(e1) OD;

   # MODULE 4. Conditional jumps #

   j := 1;
   TO n4
   DO (j = 1 | j := 2 | j := 3);
      (j > 2 | j := 0 | j := 1);
      (j < 1 | j := 1 | j := 0)
   OD;

   # MODULE 5. Omitted #
   # MODULE 6. Integers #

   j := 1; k := 2; l := 3;
   TO n6
   DO j := j * (k - j) * (l - k);
      k := l * k - (l - j) * k;
      l := (l - k) * (k + j);
      e1[l - 1] := j + k + l;
      e1[k - 1] := j * k * l
   OD;

   # MODULE 7. Trigonometry #

   x := y := 0.5;

   TO n7
   DO x := t * arctan(t2 * sin(x) * cos(x) /
                  (cos (x + y) + cos(x - y) - 1.0));
      y := t * arctan(t2 * sin(y) * cos(y) /
                  (cos (x + y) + cos(x - y) - 1.0))
   OD;

   # MODULE 8. Calls #

   x := y := z := 1.0;

   TO n8
   DO p3(x, y, z)
   OD;

   # MODULE 9. Array references #

   j := 1; k := 2; l := 3;
   e1[1] := 1.0; e1[2] := 2.0; e1[3] := 3.0;
   TO n9
   DO po
   OD;

   # MODULE 11. Standard functions #

   x := 0.75;
   TO n11
   DO x := sqrt(exp(ln(x) / t1))
   OD;

   time := (seconds - cpu1) / cycles;
   printf (($lz-d.2d, xz-d.2d$, time, 1 / (time / (ii / 10))))
OD;
print ((new line, "collections: ", collections))
```

12.4 Primality tests

12.4.1 Rabin-Miller

```
COMMENT
   This implements the Rabin-Miller primality test in Algol 68.
   If a number does not pass, it is not prime.
   If a number passes, it is prime with approximate probability
   1 - (1 / 4) ^ max trials. Literature gives better choices for
   test numbers 'a' than used here.
COMMENT

INT max trials = 10,
UNT p, m, pow := 2, INT b, MODE UNT = LONG LONG INT;

read (p);

# Dissect the number under test in form p = (2 ^ b) * m + 1 #

FOR n WHILE pow < p
DO IF (p - 1) MOD pow = 0
   THEN b := n;
        m := (p - 1) OVER pow
   FI;
   pow *:= 2
OD;

printf (($lg(0), " = 2 ^ ", g(0), " * ", g(0), " + 1"$, p, b, m));

# Test primality #

BOOL is prime := TRUE;

TO max trials WHILE is prime
DO UNT a = 1 + ENTIER (next random * (p - 1));
   IF OP GCD = (UNT a, b) UNT:
         IF b = 0
         THEN ABS a
         ELSE b GCD (a MOD b)
         FI;
      PRIO GCD = 6;
      p GCD a ˜= 1
   THEN is prime := FALSE
   ELIF PROC up mod = (UNT a, b, m) UNT: # (a ˆ b) MOD m #
        (UNT prod := 1, expo := b, pow := a MOD m;
         WHILE expo ˜= 0
         DO IF expo MOD 2 = 1
            THEN prod := (prod * pow) MOD m
            FI;
            pow := pow ˆ 2 MOD m;
            expo OVERAB 2
         OD;
         prod
        );
        UNT z := up mod (a, m, p);
        z = 1 OR z = p - 1
   THEN SKIP # pass #
   ELSE BOOL pass := FALSE;
        TO b WHILE NOT pass
        DO z := up mod (z, 2, p);
           pass := z = p - 1
        OD;
        is prime := is prime AND pass
   FI
OD;

IF is prime
THEN printf (($lg(0)" is prime with approximate probability ",
        h$, p, 1.0 - 0.25 ˆ max trials))
ELSE printf (($lg(0), " is a composite number"$, p))
FI
```

12.4.2 Lucas-Lehmer

```
CO
Lucas-Lehmer Test: for p a prime, the Mersenne number
2 ** p - 1 is prime if and only if 2 ** p - 1 divides S(p - 1)
where S(n + 1) = S(n) ** 2 - 2, and S(1) = 4.

This program verifies that
   M_11213 = 2 ** 11213 - 1
is prime.
CO

INT p = 11213;

printf (($l"M_", g(0), b(" is", " is not"), " prime"$, p,
         (
           LONG LONG INT m p =  LONG LONG 2 ** p - 1,
           LONG LONG INT s := 4;
           FROM 3 TO p
           DO s := (s * s - 2) MOD m p
```

```
          OD;
          s = 0
        )
      ))

PR precision=6800 timelimit=120 PR
```

12.5 Hilbert matrix using fractions

```
COMMENT
An application for multi-precision LONG LONG INT.
Calculate exact determinant of Hilbert matrices using fractions.
COMMENT

#
Fraction data structure and denotation through the "DIV" operator.
A fraction has positive denominator; the nominator holds the sign.
#

MODE FRAC = STRUCT (NUM nom, den), NUM = LONG LONG INT;

OP N = (FRAC u) NUM: nom OF u,
   D = (FRAC u) NUM: den OF u;

PR precision=101 PR # NUM can hold a googol! #

FRAC zero = 0 DIV 1, one = 1 DIV 1;

OP DIV = (NUM n, d) FRAC:
   IF d = 0
   THEN print ("Zero denominator"); stop
   ELSE NUM gcd = ABS n GCD ABS d;
              (SIGN n * SIGN d * ABS n OVER gcd, ABS d OVER gcd)
   FI;

OP DIV = (INT n, d) FRAC: NUM (n) DIV NUM (d);

PRIO DIV = 2;

OP GCD = (NUM a, b) NUM:
   IF b = 0
   THEN ABS a
   ELSE b GCD (a MOD b)
   FI;

PRIO GCD = 8;

# Basic operators for fractions. #

OP - = (FRAC u) FRAC: - N u DIV D u;

OP + = (FRAC u, v) FRAC:
   N u * D v + N v * D u DIV D u * D v;

OP - = (FRAC u, v) FRAC: u + - v;

OP * = (FRAC u, v) FRAC: N u * N v DIV D u * D v;

OP / = (FRAC u, v) FRAC: u * (D v DIV N v);

OP +:= = (REF FRAC u, FRAC v) REF FRAC: u := u + v;

OP -:= = (REF FRAC u, FRAC v) REF FRAC: u := u - v;

OP *:= = (REF FRAC u, FRAC v) REF FRAC: u := u * v;

OP /:= = (REF FRAC u, FRAC v) REF FRAC: u := u / v;

OP = = (FRAC u, v) BOOL: N u = N v ANDF D u = D v;

OP /= = (FRAC u, v) BOOL: NOT (u = v);

# Matrix algebra. #

OP INNER = ([] FRAC u, v) FRAC:
   # Innerproduct of two arrays of rationals #
   BEGIN FRAC s := zero;
         FOR i TO UPB u
         DO s +:= u[i] * v[i]
         OD;
         s
   END;

PRIO INNER = 8;

PROC lu decomposition = (REF [, ] FRAC a, REF [] INT p) VOID:
     # LU-decomposition cf. Crout, of a matrix of rationals. #
     BEGIN INT n = 1 UPB a;
           FOR k TO n
           DO FRAC piv := zero, INT k1 := k - 1;
              REF INT pk = p[k];
              REF [] FRAC aik = a[, k], aki = a[k,];
              FOR i FROM k TO n
              DO aik[i] -:= a[i, 1 : k1] INNER aik[1 : k1];
                 IF piv = zero
                 THEF aik[i] /= zero
                 THEN piv := aik[i];
                      pk := i
                 FI
              OD;
              IF piv = zero
              THEN print((new line, "Singular matrix")); stop
              FI;
              IF pk /= k
              THEN FOR i TO n
                   DO FRAC r = aki[i];
                      aki[i] := a[pk, i];
                      a[pk, i] := -r
                   OD
              FI;
              FOR i FROM k + 1 TO n
              DO aki[i] -:=
                    aki[1 : k1] INNER a[1 : k1, i] /:= piv
              OD
           OD
      END;

PROC determinant = ([, ] FRAC a) FRAC:
     # Determinant of a decomposed matrix is its trace. #
     BEGIN FRAC d := one;
           FOR i TO 1 UPB a
           DO d *:= a[i, i]
           OD;
           d
     END;

# Table of required results. #

[] NUM table = BEGIN
   1,
   12,
   2 160,
   6 048 000,
   266 716 800 000,
   186 313 420 339 200 000,
   2 067 909 047 925 770 649 600 000,
   365 356 847 125 734 485 878 112 256 000 000,
   1 028 781 784 378 569 697 887 052 962 909 388 800 000 000
   46 206 893 947 914 691 316 295 628 839 036 278 726 983 680 000 000 000
END;

# Compute determinant of Hilbert matrix of increasing rank. #

FOR n TO UPB table
DO [1 : n, 1 : n] FRAC a;
   FOR i TO n
   DO a[i, i] := 1 DIV 2 * i - 1;
      FOR j FROM i + 1 TO n
      DO a[i, j] := a[j, i] := 1 DIV i + j - 1
      OD
   OD;
   lu decomposition(a, LOC [1 : n] INT);
   FRAC det = determinant (a);
   print(("Rank ", whole (n, 0), ", determinant ",
          whole (N det, 0), " / ", whole (D det, 0),
          IF N det = 1 AND D det = table[n]
          THEN ", ok"
          ELSE ", not ok"
          FI, new line))
OD
```

12.6 Mastermind code breaker

```
# This breaks a code of 'pegs' unique digits that you think of. #

INT pegs = 4, max digit = 6;

MODE LIST = FLEX [1 : 0] COMBINATION,
     COMBINATION = [pegs] DIGIT,
     DIGIT = INT;

OP +:= = (REF LIST u, COMBINATION v) REF LIST:
   # Add one combination to a list. #
   (sweep heap;
    [UPB u + 1] COMBINATION w;
    w[ : UPB u] := u;
    w[UPB w] := v;
    u := w
   );

PROC gen = (REF COMBINATION part, INT peg) VOID:
     # Generate all unique [max digit!/(max digit-pegs)!] combinations. #
     IF peg > pegs
     THEN all combs +:= part
     ELSE FOR i TO max digit
          DO IF BOOL unique := TRUE;
                FOR j TO peg - 1 WHILE unique
                DO unique := part[j] ~= i
                OD;
                unique
             THEN part[peg] := i;
                  gen (part, peg + 1)
             FI
          OD
     FI;

LIST all combs;
gen (LOC COMBINATION, 1);

PROC break code = (LIST sieved) VOID:
     # Present a trial and sieve the list with entered score. #
     CASE UPB sieved + 1
     IN printf ($"Inconsistent scores"$),
        printf (($"Solution is "4(d)$, sieved[1]))
     OUT printf (($g(0), " solutions in set; Guess is "4(d)": "$,
                 UPB sieved, sieved[1]));
         # Read the score as a sequence of "w" and "b" #
         INT col ok := 0, pos ok := 0, STRING z := "";
         WHILE z = ""
         DO read ((z, new line))
         OD;
         FOR i TO UPB z
         DO (z[i] = "w" | col ok |: z[i] = "b" | pos ok) +:= 1
         OD;
         (pos ok = pegs | stop);
         # Survivors are combinations with score as entered. #
         LIST survivors;
         FOR i FROM 2 TO UPB sieved
         DO INT col ok i := 0, pos ok i := 0;
            FOR u TO pegs
            DO FOR v TO pegs
               DO IF sieved[1][u] = sieved[i][v]
                  THEN (u = v | pos ok i | col ok i) +:= 1
                  FI
               OD
            OD;
            IF col ok = col ok i AND pos ok = pos ok i
            THEN survivors +:= sieved[i]
            FI
         OD;
         break code (survivors)
     ESAC;

printf (($"This breaks a unique code of ", g(0),
          " digits in range 1..", g(0),
          " that you think of.", 2l,
          "Enter score as a series of w(hite) and b(lack)", l,
          "for instance www, wbw or bbbb.", 2l
         $, pegs, max digit));

break code (all combs)
```

12.7 Mandelbrot set

```
COMMENT
Plot (part of) the Mandelbrot set, that are points z[0] in the
complex plane for which the sequence
   z[n+1] := z[n] ** 2 + z[0] (n >= 0)
is bounded.
COMMENT

INT pix = 150, max iter = 256, REAL zoom = 0.33 / pix;
[-pix : pix, -pix : pix] INT plane;
COMPL ctr = 0.05 I 0.75 # center of set #;

# Compute the length of an orbit. #
PROC iterate = (COMPL z0) INT:
  BEGIN COMPL z := z0, INT iter := 1;
```

```
        WHILE (iter +:= 1) < max iter # not converged # AND
              ABS z < 2 # not diverged #
        DO z := z * z + z0
        OD;
        iter
  END;

# Compute set and find maximum orbit length. #
INT max col := 0;
FOR x FROM -pix TO pix
DO FOR y FROM -pix TO pix
   DO COMPL z0 = ctr + (x * zoom) I (y * zoom);
      IF (plane [x, y] := iterate (z0)) < max iter
      THEN (plane [x, y] > max col | max col := plane [x, y])
      FI
   OD
OD;

# Make a plot. #
FILE plot;
INT num pix = 2 * pix + 1;
make device (plot, "gif", whole (num pix, 0) + "x" + whole (num pix, 0));
open (plot, "mandelbrot.gif", stand draw channel);
FOR x FROM -pix TO pix
DO FOR y FROM -pix TO pix
   DO INT col = (plane [x, y] > max col | max col | plane [x, y]);
      REAL c = sqrt (1- col / max col); # sqrt to enhance contrast #
      draw colour (plot, c, c, c);
      draw point (plot, (x + pix) / (num pix - 1),
                        (y + pix) / (num pix  - 1))
   OD
OD;
close (plot)
```

12.8 Knight traversing a chess board

```
# Show paths by which a knight can traverse a
  n*n board without passing a square twice. #

PR timelimit=60 PR

INT n = 5, empty = 0;
[n, n] INT board;

FOR i TO n
DO FOR j TO n
   DO board[i, j] := empty
   OD
OD;

```

```
INT solutions := 0;

PROC go to square = (INT u, v, move) VOID:
  # Backtracking routine to traverse board. #
  IF (u >= 1 AND u <= n ANDF v >= 1 AND v <= n) ANDF board[u, v] = empty
  THEN board[u, v] := move;
       IF move = n * n
       THEN printf (($l"solution"xg(0)n(n)(ln(n)(x2d))$, solutions +:= 1, board))
       ELSE go to square (u + 1, v + 2, move + 1);
            go to square (u + 2, v + 1, move + 1);
            go to square (u + 2, v - 1, move + 1);
            go to square (u + 1, v - 2, move + 1);
            go to square (u - 1, v - 2, move + 1);
            go to square (u - 2, v - 1, move + 1);
            go to square (u - 1, v + 2, move + 1);
            go to square (u - 2, v + 1, move + 1)
       FI;
       board[u, v] := empty
  FI;

go to square (1, 1, 1)
```

12.9 Formula manipulation

```
# This example is based on example 11.10 in the Revised Report on Algol 68 #

# Data structure #

MODE FORMULA = UNION (REF CONST, REF VAR, REF TRIPLE, REF CALL),
     TRIPLE = STRUCT (FORMULA lhs, INT operator, FORMULA rhs),
     CALL = STRUCT (REF FUNCTION name, FORMULA parameter),
     FUNCTION = STRUCT (REF VAR bound var, FORMULA body),
     VAR = STRUCT (STRING name, NUMBER value),
     CONST = STRUCT (NUMBER value),
     NUMBER = LONG LONG REAL;

# Access operators #

OP VALUE = (REF CONST c) REF NUMBER: value OF c,
   VALUE = (REF VAR v) REF NUMBER: value OF v,
   NAME = (REF VAR v) REF STRING: name OF v,
   NAME = (REF CALL c) REF FUNCTION: name OF c,
   PARAMETER = (REF CALL c) REF FORMULA: parameter OF c,
   LHS = (REF TRIPLE t) REF FORMULA: lhs OF t,
   RHS = (REF TRIPLE t) REF FORMULA: rhs OF t,
   OPERATOR = (REF TRIPLE t) REF INT: operator OF t,
   BOUND = (REF FUNCTION f) REF REF VAR: bound var OF f,
   BODY = (REF FUNCTION f) REF FORMULA: body OF f;

# Generate objects #
```

```

PROC make triple = (FORMULA u, INT op, FORMULA v) REF TRIPLE:
     HEAP TRIPLE := (u, op, v);

PROC make call = (REF FUNCTION name, FORMULA parameter) REF CALL:
     HEAP CALL := (name, parameter);

PROC make function = (REF VAR bound var, FORMULA body) REF FUNCTION:
     HEAP FUNCTION := (bound var, body);

PROC make var = (STRING name, NUMBER value) REF VAR: HEAP VAR := (name, value);

PROC make const = (NUMBER x) REF CONST: (HEAP CONST c; VALUE c := x; c);

REF CONST zero = make const (0), one = make const (1), two = make const (2);

# Basic routines and operators #

PROC is var = (FORMULA f) BOOL: (f | (REF VAR): TRUE | FALSE);
PROC is const = (FORMULA f) BOOL: (f | (REF CONST): TRUE | FALSE);

PROC var name = (FORMULA f) STRING: (f | (REF VAR v): NAME v | SKIP);

INT plus = 1, minus = 2, times = 3, divide = 4, power = 5;

OP = = (FORMULA a, REF CONST b) BOOL:
   (a | (REF CONST c): c IS b | FALSE);

OP + = (FORMULA a, b) FORMULA:
   (a = zero | b |: b = zero | a | make triple (a, plus, b));

OP - = (FORMULA a, b) FORMULA:
   (b = zero | a | make triple (a, minus, b));

OP * = (FORMULA a, b) FORMULA:
   IF a = zero OR b = zero
   THEN zero
   ELSE (a = one | b |: b = one | a | make triple (a, times, b))
   FI;

OP / = (FORMULA a, b) FORMULA:
   IF a = zero AND NOT (b = zero)
   THEN zero
   ELSE (b = one | a | make triple (a, divide, b))
   FI;

OP ˆ = (FORMULA a, REF CONST b) FORMULA:
   IF a = one OR (b IS zero)
   THEN one
   ELSE (b IS one | a | make triple (a, power, b))
   FI;

# Applications of above definitions: derivative, evaluation, simplification #

```

```
PROC derivative = (FORMULA e, # with respect to # REF VAR x) FORMULA:
     # derivative a formula #
     CASE e
     IN (REF CONST): zero,
        (REF VAR v): (v IS x | one | zero),
        (REF TRIPLE f):
           CASE FORMULA u = LHS f, v = RHS f;
                FORMULA deriv u = derivative (u, x),
                        deriv v = derivative (v, x);
                OPERATOR f
           IN deriv u + deriv v,
              deriv u - deriv v,
              u * deriv v + deriv u * v,
              (deriv u - f * deriv v) / v,
              CASE v
              IN (REF CONST c):
                    v * u ^ make const (VALUE c - 1) * deriv u
              ESAC
           ESAC,
        (REF CALL c):
           BEGIN
              REF FUNCTION f = NAME c;
              FORMULA g = PARAMETER c;
              REF VAR y = BOUND f;
              REF FUNCTION deriv f = make function (y, derivative (BODY f, y));
              (make call (deriv f, g)) * derivative (g, x)
           END
     ESAC;

PROC evaluate = (FORMULA e) NUMBER:
     # Value of a formula #
     CASE e
     IN (REF CONST c): VALUE c,
        (REF VAR v): VALUE v,
        (REF TRIPLE f):
           CASE NUMBER u = evaluate (LHS f), v = evaluate (RHS f);
                OPERATOR f
           IN u + v, u - v, u * v, u / v, long long exp (v * long long ln (u))
           ESAC,
        (REF CALL c):
           BEGIN
              REF FUNCTION f = NAME c;
              VALUE BOUND f := evaluate (PARAMETER c);
              evaluate (BODY f)
           END
     ESAC;

PROC simplify = (REF FORMULA e) VOID:
     # Example simplifications - extend as you see fit #
     CASE e
     IN (REF CALL c): (simplify (BODY NAME c), simplify (PARAMETER c)),
        (REF TRIPLE f):
           IF REF FORMULA u = LHS f, v = RHS f;
              (simplify (u), simplify (v));
              OPERATOR f = plus
           THEN IF (is var (u) AND is var (v))
                THEF (var name (u) = var name (v))
                THEN e := two * u
                FI
           ELIF OPERATOR f = times
           THEN IF is const (v)
                THEN e := make triple (v, times, u);
                     simplify (e)
                FI;
                IF is const (u) ANDF (u = zero)
                THEN e := zero
                FI
           FI
     ESAC;

# A small demonstration #

OP FMT = (NUMBER x) STRING:
   IF x = ENTIER x
   THEN whole (x, 0)
   ELSE fixed (x, 0, 4)
   FI;

PROC write = (FORMULA e) VOID:
     CASE e
     IN (REF CONST c): print (FMT VALUE c),
        (REF VAR v): print (NAME v),
        (REF TRIPLE f):
           BEGIN
              print ("(");
              write (LHS f);
              print ((OPERATOR f | " + ", " - ", " * ", " / ", " ^ "));
              write (RHS f);
              print (")")
           END
     ESAC;

REF VAR x = make var ("x", -1), y = make var ("y", 1);

PROC demo = (REF FORMULA f, REF VAR z) VOID:
     BEGIN
       print (("x = ", FMT VALUE x, "; y = ", FMT VALUE y, new line, "f = "));
       write (f);
       simplify (f);
       print ((newline, "  = "));
       write (f);
       print ((" = ", FMT evaluate (f), new line, "df/d", NAME z, " = "));
       FORMULA g := derivative (f, z);
       write (g);
       print ((newline, "  = "));
       simplify (g);
       write (g);
       print ((" = ", FMT evaluate (g), new line, new line))
```

```
    END;

demo (LOC FORMULA := x + x + zero * y, x);
demo (LOC FORMULA := x * two, x);
demo (LOC FORMULA := x * x + y * y, x);
demo (LOC FORMULA := x * x + two * x * y + y * y, x);
demo (LOC FORMULA := x + y / x, x)

COMMENT Output produced:
x = -1; y = 1
f = (x + x)
  = (2 * x) = -2
df/dx = 2
  = 2 = 2

x = -1; y = 1
f = (x * 2)
  = (2 * x) = -2
df/dx = 2
  = 2 = 2

x = -1; y = 1
f = ((x * x) + (y * y))
  = ((x * x) + (y * y)) = 2
df/dx = (x + x)
  = (2 * x) = -2

x = -1; y = 1
f = (((x * x) + ((2 * x) * y)) + (y * y))
  = (((x * x) + ((2 * x) * y)) + (y * y)) = 0
df/dx = ((x + x) + (2 * y))
  = ((2 * x) + (2 * y)) = 0

x = -1; y = 1
f = (x + (y / x))
  = (x + (y / x)) = -2
df/dx = (1 + ((0 - (y / x)) / x))
  = (1 + ((0 - (y / x)) / x)) = 0

COMMENT
```

12.10 Fibonacci grammar

```
#
This program computes Fibonacci numbers by counting the number of
derivations of the "Fibonacci grammar":

    fib: "a"; "a", fib; "aa", fib.

The purpose for this is to illustrate the use of procedure closures
```

```
which we call continuations. We use this to generate a recursive
descent with backup parser following a simple translation from grammar
rules to procedures.

Contributed by Eric Voss.
#

MODE CONT = PROC (INT) VOID;

PROC grammar fib = (INT i, CONT q) VOID:
     BEGIN terminal (i, "a", q);
           terminal (i, "a", (INT j) VOID: grammar fib (j,  q));
           terminal (i, "aa", (INT j) VOID: grammar fib (j,  q))
     END;

PROC terminal = (INT i, STRING a, CONT q) VOID:
     IF INT u = i + UPB a - 1;
        u > UPB sentence
     THEN SKIP
     ELIF sentence [i : u] = a
     THEN q (u + 1)
     FI;

STRING sentence;
FOR k TO 10
DO sentence := k * "a";
   INT nr derivations := 0;
   grammar fib (1, (INT j) VOID: (j > UPB sentence | nr derivations +:= 1));
   print (("Fibonacci number ", UPB sentence,  " = ", nr derivations, new line))
OD
```

12.11 Monte Carlo simulation

```
#! ./a68g
#! ./a68g
COMMENT

Copyright (C) 2006-2008 J. Marcel van der Veer <algol68g@xs4all.nl>.

This is a simple program that generates trajectories for atoms in a binary
Lennard-Jones TVN ensemble through the Metropolis Monte Carlo method.

Instead of a Verlet table this program uses a sorted list along the z-axis. This
gives a comparable set estimate for interacting neighbours.

The program stores the energy for each particle to calculate initial energie
quickly. If a state is accepted then the energies are updated. This saves one
energy calculation if a state is not accepted.

COMMENT
```

```

PR heap=256M nowarnings PR

BOOL continuing = FALSE,          # FALSE for new, TRUE for continuing #
STRING jobname = "triple_point", # Job title                          #
INT states = 50,                  # States per particle                #
INT n x = 8, n y = 8, n z = 8,   # Unit cells per axis                #
REAL density = 0.85, cut = 2.5,  # Density and Lennard-Jones cut-off  #
INT res = 1000;                   # Graphics resolution                #

CO
Next setting Lennard-Jones parameters makes a binary system behave like a
one-component system.
CO

[, ] REAL ekt = ((0.72, 0.72),   # E/kT between species i and j       #
                 (0.72, 0.72)),  # Sigma between species i and j      #
[, ] REAL sig = ((1.0, 1.0),
                 (1.0, 1.0));

INT cells = n x * ny * nz;
INT no atoms = 4 * cells;

CO
Basic vector and matrix stuff declared in a bottom-up way.
This code does not use the GSL extensions to a68g, which would make it faster.
CO

MODE VECTOR = STRUCT (REAL x, y, z);

VECTOR zero = (0, 0, 0);

OP + = (VECTOR u, v) VECTOR: (x OF u + x OF v, y OF u + y OF v, z OF u + z OF v);
OP +:= = (REF VECTOR u, VECTOR v) REF VECTOR: u := u + v;

OP - = (VECTOR u, v) VECTOR: (x OF u - x OF v, y OF u - y OF v, z OF u - z OF v);
OP -:= = (REF VECTOR u, VECTOR v) REF VECTOR: u := u - v;

OP * = (VECTOR u, REAL s) VECTOR: (s * x OF u, s * y OF u, s * z OF u);
OP *:= = (REF VECTOR u, REAL s) REF VECTOR: u := u * s;
OP * = (VECTOR u, v) REAL: x OF u * x OF v + y OF u * y OF v + z OF u * z OF v;

OP / = (VECTOR u, REAL v) VECTOR: u * (1 / v);

OP NORM = (VECTOR u) REAL: sqrt (u * u);
OP SQR = (VECTOR u) VECTOR: (x OF u ** 2, y OF u ** 2, z OF u ** 2);
OP VOL = (VECTOR u) REAL: x OF u * y OF u * z OF u;
OP TRACE = (VECTOR u) REAL: x OF u + y OF u + z OF u;

# Operators for periodic boundary conditions #

OP NEAREST = (REAL r, REAL l) REAL:
   (REAL hl = l * 0.5; r > hl | r - l |: r < - hl | r + l | r);

```

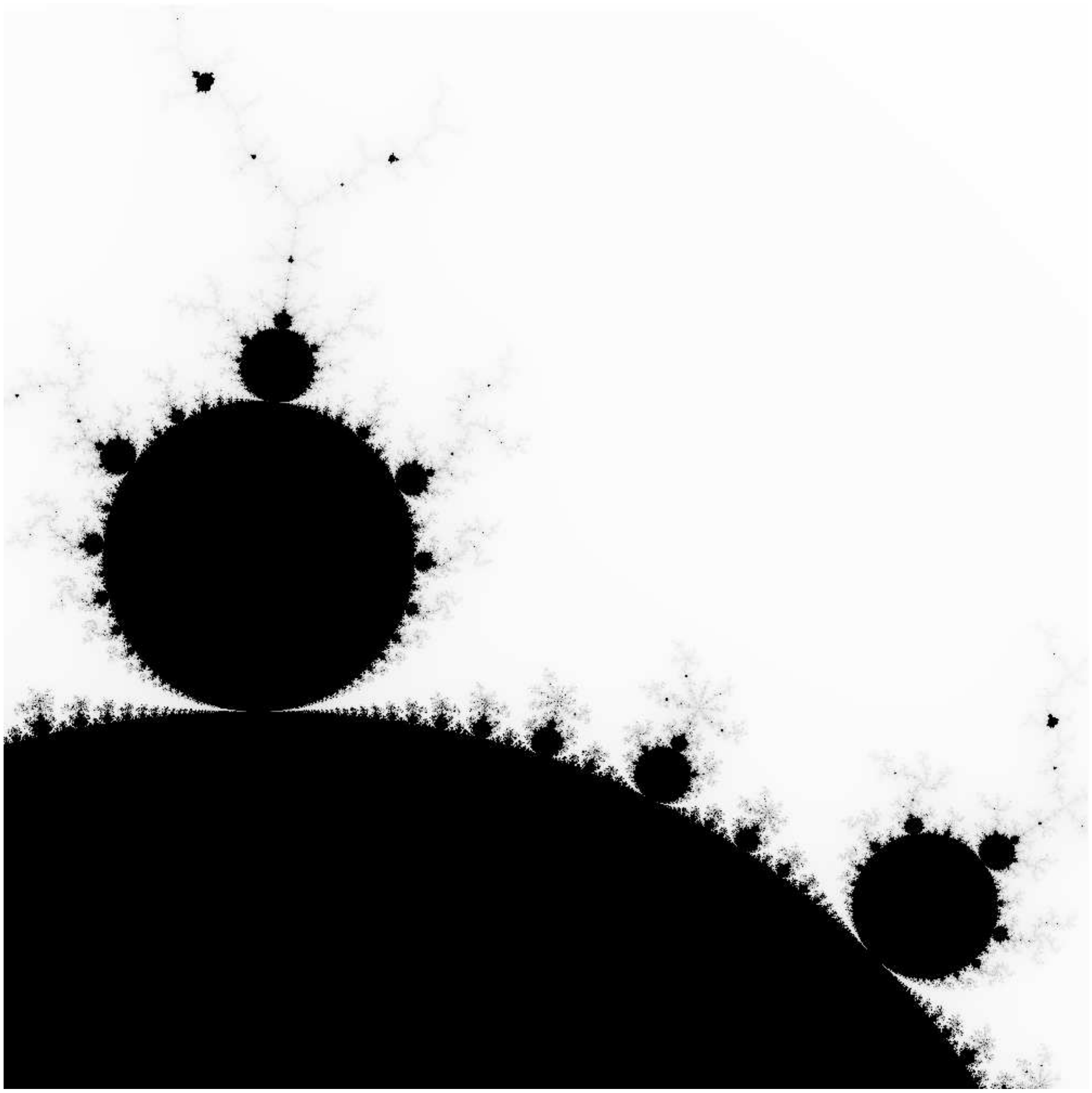
```
OP NEAREST = (VECTOR r, VECTOR l) VECTOR:
   (x OF r NEAREST x OF l, y OF r NEAREST y OF l, z OF r NEAREST z OF l);

OP INSIDE = (REAL r, REAL l) REAL:
   (r < 0 | r + l |: r >= l | r - l | r);

OP INSIDE = (VECTOR r, VECTOR l) VECTOR:
   (x OF r INSIDE x OF l, y OF r INSIDE y OF l, z OF r INSIDE z OF l);

PRIO NEAREST = 2, INSIDE = 2;

# Atoms, ensembles, etcetera. #

MODE ATOM =  STRUCT (VECTOR pos, INT species, LONG REAL u, VECTOR t);

MODE ENSEMBLE = STRUCT ([no atoms] ATOM parts,
                        [no atoms] INT admin, link,
                        VECTOR box,
                        INT steps, steps accepted,
                        INT swaps, swaps accepted,
                        INT resizes, resizes accepted,
                        REAL step, prev diff,
                        REAL energy,
                        VECTOR stress),

     NEIGHBOUR = STRUCT (INT n, VECTOR dr, REAL r, REAL u);

REAL min exp arg = - ln (max real);

PROC expf = (REAL x) REAL:
  IF x > min exp arg
  THEN exp (x)
  ELSE 0
  FI;

# Basic routines to work with an ensemble #

PROC print stats = (REF FILE f, REF ENSEMBLE s) VOID:
  (putf (f, ($l"steps/N="g(10)x"accept rate="g(0, 6)x"step="g(0, 6)x
              "diffusion="g(0, 6)$,
            steps OF ensemble / no atoms,
            steps accepted OF ensemble / steps OF ensemble,
            step OF ensemble, prev diff OF ensemble));
   putf (f, ($l"swaps="g(10)x"accept rate="g(0, 6)$,
            swaps OF ensemble, swaps accepted OF ensemble / swaps OF ensemble));
   putf (f, ($l"collections="g(10)$, collections))
  );

PROC print physics = (REF FILE f, REF ENSEMBLE s) VOID:
  (putf (f, ($l"atoms="g(0)$, no atoms));
   putf (f, ($l"box size="3(xg(10,6))x"volume="xg(10,6)$, box OF s,
             VOL box OF s));
   putf (f, ($l"E/V="xg(10,6)$, energy OF s / VOL box OF s));
   putf (f, ($l"T="3(xg(10,6))$, stress OF s / VOL box OF s));
   putf (f, ($l"p="xg(10,6)$, - TRACE stress OF s / VOL box OF s / 3))
  );

PROC print ensemble = (REF ENSEMBLE s) VOID:
  (FILE out;
   open (out, job name + ".txt", standout channel);
   print stats (out, s);
   FOR i TO no atoms
   DO putf (out, ($lg(-5)3(xg(10,6))xg(0)xg(0,12)x3(xg(10, 6))$, i,
                  (parts OF ensemble)[i]))
   OD;
   close (out)
  );

PROC gen fcc = (REF ENSEMBLE s, INT n x, n y, n z) VOID:
  (INT m := 0;
   REF [] ATOM p = parts OF s;
   REAL frac = (4 / density) ** (1 / 3);
   box OF s := (n x * frac, n y * frac, n z * frac);
   FOR k FROM 0 TO n z - 1
   DO FOR i FROM 0 TO n x - 1
      DO FOR j FROM 0 TO n y - 1
         DO PROC solid = (INT n) BOOL: (n <= n z OVER 2);
            p[m +:= 1] := ((i, j, k), (solid (k + 1) | 1 | 2), 0, zero);
            p[m +:= 1] := ((i + 0.5, j + 0.5, k),
                           (solid (k + 1) | 1 | 2), 0, zero);
            p[m +:= 1] := ((i + 0.5, j, k + 0.5),
                           (solid (k + 1) | 1 | 2), 0, zero);
            p[m +:= 1] := ((i, j + 0.5, k + 0.5),
                           (solid (k + 1) | 1 | 2), 0, zero)
         OD
      OD
   OD;
   FOR i TO m
   DO pos OF p[i] +:= VECTOR (0.25, 0.25, 0.25);
      pos OF p[i] *:= frac
   OD
  );

PROC dump box = (REF ENSEMBLE s) VOID:
  (FILE out;
   open (out, job name + ".bin", stand back channel);
   put bin (out, s);
   close (out)
  );

PROC read box = (REF ENSEMBLE s) VOID:
  (FILE in;
   open (in, job name + ".bin", stand back channel);
   get bin (in, s);
   close (in)
  );

PROC plot = (REF FILE f, REF ENSEMBLE s) VOID:
  (REF [] ATOM p = parts OF s;
   [UPB p] INT key;
   sort (key, NIL, y OF pos OF p);
   VECTOR box = box OF s;
   # Draw background and outline central cell #
   draw backgroundcolour (f, 1, 1, 1);
   draw erase (f);
   # Draw atoms and near images #
   REAL fact = z OF box / x OF box;
   FOR k TO no atoms
   DO VECTOR q = pos OF p[key[k]];
      VECTOR r = (x OF q / x OF box, y OF q / y OF box, fact * z OF q / z OF box);
      REAL c = 0.3 + 0.6 * y OF r;
      INT spec = species OF p[key[k]];
      IF spec = 1
      THEN draw colour (f, 0, c, c);
           draw ball (f, z OF r, x OF r, sig[spec, spec] * fact * 16 / res)
      ELSE draw colour (f, c, c, c);
           draw ball (f, z OF r, x OF r, sig[spec, spec] * fact * 16 / res)
      FI
   OD
  );

# Routines to work with a neighbour list #

PROC swap = (REF INT i, REF INT j) VOID:
  (INT k = i;
   i := j;
   j := k
  );

PROC sort = (REF [] INT index, REF [] INT link, [] REAL z) VOID:
  (FOR i TO UPB index
   DO index[i] := i
   OD;
   FOR i TO UPB z - 1
   DO FOR j FROM i + 1 TO UPB z
      DO IF z[index[i]] > z[index[j]]
         THEN swap (index[i], index[j])
         FI
      OD
   OD;
   IF link ISNT NIL
   THEN FOR i TO UPB z
        DO link[index[i]] := i
        OD
   FI
  );

PROC resort = (REF ENSEMBLE s, INT m) VOID:
  (# Rearrange the neighbour list assumpting that displacements are small #
   REF [] REAL z = z OF pos OF parts OF s;
   REAL z m = z[m];
   REF [] INT admin = admin OF s, link = link OF s;
```

```
    INT i := link[m];
    # Shift particle up the list #
    WHILE i < no atoms ANDF z m > z[admin[i + 1]]
    DO admin[i] := admin[i + 1];
       link[admin[i]] := i;
       admin[i + 1] := m;
       link[m] := i + 1;
       i +:= 1
    OD;
    # Shift particle down the list #
    WHILE i > 1 ANDF z m < z[admin[i - 1]]
    DO admin[i] := admin[i - 1];
       link[admin[i]] := i;
       admin[i - 1] := m;
       link[m] := i - 1;
       i -:= 1
    OD
   );

# Routines to work with energy tables #

PROC atom energy = (REF ENSEMBLE s, REF [] NEIGHBOUR list, INT cent) LONG REAL:
  (# Total energy of particle "cent" with neighbour list "list" #
   LONG REAL sum := 0;
   REF [] ATOM p = parts OF s;
   INT spec cent = species OF p[cent];
   FOR i TO UPB list
   DO REF NEIGHBOUR neigh = list[i];
      ATOM a = p[n OF neigh];
      IF r OF neigh < cut
      THEN sum +:= (u OF neigh := lj e 12 6 (ekt[spec cent, species OF a],
                    sig[spec cent, species OF a], r OF neigh))
      ELSE u OF neigh := 0
      FI
   OD;
   sum
  );

PROC atom stress = (REF ENSEMBLE s, REF [] NEIGHBOUR list, INT cent) VECTOR:
  (# Stress of particle "cent" with neighbour list "list" #
   VECTOR sum := zero;
   REF [] ATOM p = parts OF s;
   INT spec cent = species OF p[cent];
   FOR i TO UPB list
   DO REF NEIGHBOUR neigh = list[i];
      ATOM a = p[n OF neigh];
      IF r OF neigh < cut
      THEN # diagonal elements only #
           sum +:= SQR (dr OF neigh) * lj f 12 6 (ekt[spec cent, species OF a],
                    sig[spec cent, species OF a], r OF neigh)
      FI
   OD;
   sum
  );
```

```

PROC collect = (REF ENSEMBLE s, INT cent) REF [] NEIGHBOUR:
  (REF [] ATOM p = parts OF s;
   HEAP [no atoms] NEIGHBOUR list;
   BOOL continue, INT k, n := 0;
   # Work down the neighbour list #
   k := ((link OF s)[cent] = 1 | no atoms | (link OF s)[cent] - 1);
   continue := TRUE;
   WHILE continue
   DO VECTOR dr = (pos OF p[cent] - pos OF p[ (admin OF s)[k]]) NEAREST box OF s;
      IF continue := ABS z OF dr < cut
      THEN IF REAL r := NORM dr;
              r < cut
           THEN list[n +:= 1] := ((admin OF s)[k], dr, r, SKIP)
           FI
      FI;
      k := (k = 1 | no atoms | k - 1)
   OD;
   # Work up the neighbour list #
   continue := TRUE;
   k := ((link OF s)[cent] = no atoms | 1 | (link OF s)[cent] + 1);
   WHILE continue
   DO VECTOR dr = (pos OF p[cent] - pos OF p[ (admin OF s)[k]]) NEAREST box OF s;
      IF continue := ABS z OF dr < cut
      THEN IF REAL r := NORM dr;
              r < cut
           THEN list[n +:= 1] := ((admin OF s)[k], dr, r, SKIP)
           FI
      FI;
      k := (k = no atoms | 1 | k + 1)
   OD;
   list[1 .. n]
  );

PROC energy and stress = (REF ENSEMBLE s) VOID:
  (REF [] ATOM p = parts OF s;
   energy OF s := 0;
   stress OF s := zero;
   FOR i TO no atoms
   DO REF [] NEIGHBOUR list = collect (s, i);
      energy OF s +:= SHORTEN (u OF p[i] := atom energy (s, list, i));
      stress OF s +:= (t OF p[i] := atom stress (s, list, i))
   OD;
   stress OF s := stress OF s * 0.5
  );

PROC rework u table = (REF ENSEMBLE s, REF [] NEIGHBOUR init, final) VOID:
  (REF [] ATOM p = parts OF s;
   FOR i TO UPB init
   DO REF NEIGHBOUR neigh = init[i];
      u OF p[n OF neigh] -:= u OF neigh
   OD;
   FOR i TO UPB final
   DO REF NEIGHBOUR neigh = final[i];
```

```
      u OF p[n OF neigh] +:= u OF neigh
   OD
  );

PROC mc step = (REF ENSEMBLE s) VOID:
  (# Generate a new state by moving all atoms a little bit #
   REF [] ATOM p = parts OF s;
   LONG REAL sum diff := 0, INT n diff := 0;
   TO no atoms
   DO INT k = ENTIER (1 + no atoms * random);
      LONG REAL u init = u OF p[k];
      PROC rnd step = REAL: (2 * random - 1) * step OF s;
      VECTOR pos init := pos OF p[k];
      # Move to new position #
      VECTOR disp = (rnd step, rnd step, rnd step);
      VECTOR pos final := (pos init + disp) INSIDE box OF s;
      pos OF p[k] := pos final;
      resort (s, k);
      REF [] NEIGHBOUR l final = collect (s, k);
      LONG REAL u final = atom energy (s, l final, k);
      # Restore old situation #
      pos OF p[k] := pos init;
      resort (s, k);
      # Do we accept the new state? Use Boltzmann"s criterion #
      REAL u diff = SHORTEN (u final - u init);
      IF u diff < 0 ORF expf (- u diff) > random
      THEN steps accepted OF s +:= 1;
           REF [] NEIGHBOUR l init = collect (s, k);
           LONG REAL u init 2 = atom energy (s, l init, k);
           IF ABS (u init - u init 2) > 0.01
           THEN print ((newline, "u table error: ", u init, u init 2))
           FI;
           rework u table (s, l init, l final);
           u OF p[k] := u final;
           pos OF p[k] := pos final;
           resort (s, k);
           sum diff +:= disp * disp;
           n diff +:= 1
      FI;
      steps OF s +:= 1
   OD;
   # Adapt step size according to optimum for hard sphere system #
   IF steps accepted OF s / steps OF s > 0.2
   THEN (step OF s < 1 | step OF s *:= 1.1)
   ELSE (step OF s > 1e-6 | step OF s /:= 1.1)
   FI;
   prev diff OF s := SHORTEN sum diff / n diff
  );

PROC mc swap = (REF ENSEMBLE s) VOID:
  (# Generate a new state by swapping two atoms of different species #
   REF [] ATOM p = parts OF s;
   INT j, k;
   DO j := ENTIER (1 + no atoms * random);
```

```
      k := ENTIER (1 + no atoms * random)
      UNTIL j /= k AND species OF p[j] /= species OF p[k]
   OD;
   swap (species OF p[j], species OF p[k]);
   LONG REAL u init = u OF p[j] + u OF p[k];
   REF [] NEIGHBOUR l final j = collect (s, j),
                    l final k = collect (s, k);
   LONG REAL u final j = atom energy (s, l final j, j);
   LONG REAL u final k = atom energy (s, l final k, k);
   LONG REAL u final = u final j + u final k;
   swap (species OF p[j], species OF p[k]);
   REAL u diff = SHORTEN (u final - u init);
   IF u diff < 0 ORF expf (- u diff) > random
   THEN REF [] NEIGHBOUR l init j = collect (s, j),
                         l init k = collect (s, k);
        atom energy (s, l init j, j);
        atom energy (s, l init k, k);
        rework u table (s, l init j, l final j);
        u OF p[j] := u final j;
        rework u table (s, l init k, l final k);
        u OF p[k] := u final k;
        swap (species OF p[j], species OF p[k]);
        swaps accepted OF ensemble +:= 1
   FI;
   swaps OF ensemble +:= 1
  );

ENSEMBLE ensemble;

# Prelude. Read old box or generate a new one #
IF continuing
THEN read box (ensemble)
ELSE steps OF ensemble := 0;
     steps accepted OF ensemble := 0;
     swaps OF ensemble := 0;
     swaps accepted OF ensemble := 0;
     resizes OF ensemble := 0;
     resizes accepted OF ensemble := 0;
     step OF ensemble := 0.5;
     prev diff OF ensemble := 0;
     gen fcc (ensemble, n x, n y, n z);
     sort (admin OF ensemble, link OF ensemble, z OF pos OF parts OF ensemble);
     energy and stress (ensemble)
FI;
# Prepare the plotter #
FILE f;
VECTOR box = box OF ensemble;
open (f, jobname, stand draw channel);
make device (f, "ps", "a4");
# Monte Carlo simulation #
print physics (standout, ensemble);
FOR i TO states
DO REAL t0 = clock;
   TO 5
   DO mc step (ensemble);
      mc swap (ensemble)
   OD;
   plot (f, ensemble);
   energy and stress (ensemble);
   print stats (standout, ensemble);
   print physics (standout, ensemble);
   printf (($l"rate="g(10)$, 5 * no atoms / (clock - t0)))
OD;
# Postlude #
plot (f, ensemble);
print ensemble (ensemble);
dump box (ensemble);
close (f)
```



Part of the Mandelbrot-set, plotted with a68g using program 12.7.

Appendix A

Reporting bugs

Your bug reports play an essential role in making Algol 68 Genie reliable and its documentation up-to-date. When you encounter a problem, then you should report the problem. Your feedback will be greatly appreciated.

A.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

1. Check using section 10.11 that your issue is not a known one.
2. If *a68g* gets a fatal signal, and for instance produces a core dump, for any input, that is most likely a bug. (But check with section 10.11 for some known conditions that can crash *a68g*).
3. If *a68g* produces an error message for valid input, that is a bug.
4. If *a68g* does not produce an error message for invalid input, that is a bug. However, you should note that your idea of ”invalid input” might be someone else’s idea of ”an extension” or ”support for traditional practice”.
5. If *a68g* just terminates without producing expected output, this may indicate a bug. But please check that the bug is not an error in the Algol 68 source program.
6. If you are an experienced user of Algol 68, your suggestions for improvement of Algol 68 Genie are welcome in any case.

A.2 How and where to report bugs

If you want to report a bug please send an e-mail to

algol68g@xs4all.nl

Please include the version number of *a68g* and if possible submit the Algol 68 code that produced the bug.

Appendix B

Algol 68 Genie context-free syntax

B.1 Introduction

This syntax summary provides a quick reference for Algol 68 Genie syntax. The syntax described here is context-free. The advantage of presenting a context-free syntax is that the backbone of constructions can be explained briefly. The disadvantage is that a context-free grammar cannot reject programs that are semantically incorrect, for instance those that apply undeclared symbols. The two-level Algol 68 syntax is described in the Revised Report. The distribution of Algol 68 Genie contains a copy of the Revised Report in HTML.

B.2 Notions and notation

In this overview following notation is adopted: `( )` is used to group notions,
`[SOME]` means *optional SOME*
`a b` means *a followed by b*
`a:  b.` means *definition of a is b*
`a, b` means *a or b*
`‘SOME’` means *literal symbol SOME*
`SOME-list` means *SOME [‘,’ SOME-list]*
`SOME-sequence` means *SOME [SOME-sequence]*

B.3 Reserved symbols

Next symbols are reserved in Algol 68 Genie: `ANDF`, `ANDTH`, `ASSERT`, `ASSIGN`, `AT`, `BEGIN`, `BITS`, `COMMENT`, `PRAGMAT`, `BOOL`, `BY`, `BYTES`, `CASE`, `CHANNEL`, `CHAR`, `CODE`, `COL`, `COMPLEX`, `COMPL`, `DIAG`, `DO`, `DOWNTO`, `EDOC`, `ELIF`, `ELSE`, `ELSF`, `EMPTY`, `END`, `ENVIRON`, `ESAC`, `EXIT`, `FALSE`, `FILE`, `FI`, `FLEX`, `FORMAT`, `FOR`, `FROM`, `GO`, `GOTO`, `HEAP`, `IF`, `IN`, `INT`, `ISNT`, `IS`, `LOC`, `LONG`, `MODE`, `NIL`, `OD`, `OF`, `OP`, `OREL`, `ORF`, `OUSE`, `OUT`, `PAR`, `PIPE`, `PRIO`, `PROC`, `REAL`, `REF`, `ROW`, `SEMA`, `SHORT`, `SKIP`, `SOUND`, `STRING`, `STRUCT`, `THEF`, `THEN`, `TO`, `TRNSP`, `TRUE`, `UNION`, `UNTIL`, `VOID`, and `WHILE` .

B.4 Coercions

The "strength" of a context determines what coercions are possible in that context. A `strong` context allows all coercions, and typically requires that the resulting mode is known a priori; for example in the case of initial values assigned in a declaration or parameters in procedure calls.

```
strong:
   firm [widening-sequence] [rowing-sequence] [voiding].

firm:
   meek [uniting].

meek:
   (dereferencing, deproceduring)-sequence.
```

Weak coercions must yield a name:

```
weak:
   (dereferencing, deproceduring)-sequence.

soft:
   deproceduring-sequence.
```

B.5 Tags

B.5.1 Identifier tags

```
mode-indicant:
   upper-case-letter [(upper-case letter, digit, underscore)-sequence].

identifier:
   lower-case-letter [(lower-case letter, digit, underscore)-sequence].

label:
   identifier.

digit:
   ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’.

lower-case-letter:
   ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’.
   ‘k’, ‘l’, ‘m’, ‘n’, ‘o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘t’.
   ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’.

upper-case-letter:
   ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’.
   ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’.
   ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’.

underscore:
   ‘_’.
```

B.5.2 Operator tags

```
operator:
   monadic operator, dyadic operator.

monadic operator:
   upper-case-letter [(upper-case letter, digit, underscore)-sequence],
   monad[nomad][‘:=’, ‘=:’].

dyadic operator:
   upper-case-letter [(upper-case letter, digit, underscore)-sequence],
   (monad, nomad)[nomad][‘:=’, ‘=:’].

monad:
   ‘+’, ‘-’, ‘!’, ‘?’, ‘%’, ‘^’, ‘&’, ‘~’.

nomad:
   ‘<’, ‘>’, ‘/’, ‘=’, ‘*’.
```

B.6 Particular program

A particular program is the actual application, embedded in the standard environ.

```
particular-program:
   [(label ':')-sequence] enclosed-clause.
```

B.7 Clauses

Serial - and enquiry-clauses describe how declarations and units (”statements”) are put in sequence. Algol 68 requires clauses to yield a value. As declarations yield no value, serial- and enquiry-clauses cannot end in a declaration.

`EXIT` leaves a serial-clause, yielding the value of the preceding unit. If the unit following `EXIT` would not be labelled, it could never be reached. Hence Revised Report syntax for the serial-clause is more elaborate than presented here to require that the unit following `EXIT` is labelled. In a serial-clause there cannot be labelled units before declarations to prevent reentering declarations once they have been executed.

```
serial-clause:
   [initialiser-series ';'] labelled-unit-series.

labelled-unit-series:
   [([label ':'] unit (';', 'EXIT'))-sequence] unit.

initialiser-series:
   (unit, declaration-list) [';' initialiser-series].
```

An enquiry-clause provides a value to direct the conditional-clause, integer-case-clause, united-case-clause or while-part in a loop-clause. An enquiry-clause has no labels, so you cannot for instance jump back to the enquiry-clause at `IF` from the serial-clause at `THEN`.

```
enquiry-clause:
   [initialiser-series ';'] [(unit ';')-sequence] unit.
```

Enclosed-clauses provide structure for a particular program.

```
enclosed-clause:
   closed-clause,
   collateral-clause,
   parallel-clause,
   conditional-clause,
   integer-case-clause,
   united-case-clause,
   loop-clause.
```

```
closed-clause:
   ‘BEGIN’ serial-clause ‘END’.

collateral-clause:
   ‘BEGIN’ [unit-list] ‘END’.
```

In a parallel-clause the elaboration of units can be synchronised using semaphores.

```
parallel-clause:
   ‘PAR’ ‘BEGIN’ unit-list ‘END’.

conditional-clause:
   ‘IF’ meek-boolean-enquiry-clause
   ‘THEN’ serial-clause
   [(‘ELIF’) meek-boolean-enquiry-clause
     ‘THEN’ serial-clause)-sequence]
   [‘ELSE’ serial-clause]
   ‘FI’.

integer-case-clause:
   ‘CASE’ meek-integer-enquiry-clause
   ‘IN’ unit-list
   [(‘OUSE’ meek-integer-enquiry-clause
     ‘IN’ unit-list)-sequence]
   [‘OUT’ serial-clause]
   ‘ESAC’.

united-case-clause:
   ‘CASE’ meek-united-enquiry-clause
   ‘IN’ specified-unit-list
   [(‘OUSE’ meek-united-enquiry-clause
     ‘IN’ specified-unit-list)-sequence]
   [‘OUT’ serial-clause]
   ‘ESAC’.

specified-unit: ‘(’ (formal-declarer, ‘VOID’) [identifier] ‘)’ ‘:’ unit.

loop-clause:
   [‘FOR’ identifier]
   [‘FROM’ meek-integer-unit]
   [‘BY’ meek-integer-unit]
   [(‘TO’, ‘DOWNTO’) meek-integer-unit]
   [‘WHILE’ meek-boolean-enquiry-clause]
   ‘DO’ serial-clause [‘UNTIL’ meek-boolean-enquiry-clause],
        [‘UNTIL’ meek-boolean-enquiry-clause]
   ‘OD’.
```

Short-hand notations for enclosed-clauses are:

```
‘(’ ... ‘)’ for
   ‘BEGIN’ ... ‘END’

‘(’ ... ‘|’ ... ‘|’ ... ‘)’ for
   ‘IF’ ... ‘THEN’ ... ‘ELSE’ ... ‘FI’

‘(’ ... ‘|’ ... ‘|:’ ... ‘|’ ... ‘)’ for
   ‘IF’ ... ‘THEN’ ... ‘ELIF’ ... ‘THEN’ ... ‘FI’

‘(’ ... ‘|’ ... ‘|’ ... ‘)’ for
   ‘CASE’ ... ‘IN’ ... ‘OUT’ ... ‘ESAC’

‘(’ ... ‘|’ ... ‘|:’ ... ‘|’ ... ‘)’ for
   ‘CASE’ ... ‘IN’ ... ‘OUSE’ ... ‘OUT’ ... ‘ESAC’
```

B.8 Statements

Statements are orthogonal, for instance an enclosed-clause can be an operand in a formula. The constituent constructs of statements are primaries, secondaries, tertiaries and units.

B.8.1 Primaries

```
primary:
   enclosed-clause,
   identifier,
   call,
   slice,
   cast,
   format-text,
   denotation.
```

A call invokes a procedure, but can be partially parametrised: partial parametrisation adds arguments to a procedure's locale - when the locale is complete the procedure is called, otherwise currying takes place.

```
call:
   meek-primary ‘(’ actual-parameter-list’)’.
```

```
actual-parameter:
   [strong-unit].
```

A slice selects an element or a sub-row from a rowed value.

```
slice:
   weak-primary ‘[’ indexer-list ‘]’.

indexer:
   [meek-integer-unit] [‘:’]
   [meek-integer-unit] [revised-lower-bound].

revised-lower-bound:
   (‘AT’, ‘@’) meek-integer-unit.
```

A cast forces a strong context in which all coercions are allowed.

```
cast:
   (formal-declarer, ‘VOID’) strong-enclosed-clause.

denotation:
   integer-denotation,
   real-denotation,
   boolean-denotation,
   bits-denotation,
   character-denotation,
   row-of-character-denotation,
   VOID-denotation.

integer-denotation:
   [‘LONG’-sequence, ‘SHORT’-sequence] digit-sequence.

real-denotation:
   [‘LONG’-sequence, ‘SHORT’-sequence] digit-sequence ‘.’
   digit-sequence[‘E’ [‘+’, ‘-’] digit-sequence].

boolean-denotation:
   ‘TRUE’, ‘FALSE’.

bits-denotation:
   [‘LONG’-sequence, ‘SHORT’-sequence] digit-sequence ‘r’
   bits-digit-sequence.

bits-digit:
   ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’,
   ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’,
   ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’,

character-denotation:
   ‘"’ string-item ‘"’.

string-item:
   character, ‘""’.

row-of-character-denotation:
   ‘"’ string-item-sequence ‘"’.

VOID-denotation:
   ‘EMPTY’.
```

Formats

```
format-text:
   ‘$’ picture-list ‘$’.

picture:
   insertion,
   pattern,
   collection,
   replicator collection.

replicator:
   integer-denotation, ‘n’ meek-integer-enclosed-clause.

collection:
   ‘(’ picture-list ‘)’.

insertion:
   (replicator ‘k’,
   [replicator] ‘l’,
   [replicator] ‘/’,
   [replicator] ‘p’,
   [replicator] ‘x’,
   [replicator] ‘q’,
   [replicator] (row-of-character-denotation)-sequence.

pattern:
   general-pattern,
   integer-pattern,
   real-pattern,
   complex-pattern,
   bits-pattern,
   string-pattern,
   boolean-pattern,
   choice-pattern,
   format-pattern.

general-pattern:
   ‘g’[strong-row-of-integer-enclosed-clause],
   ‘h’[strong-row-of-integer-enclosed-clause].

integer-pattern:
   [sign-mould] integer-mould.

sign-mould:
   [replicator] [‘s’] (‘z’, ‘d’)
   [([replicator] [‘s’] (‘z’, ‘d’), insertion)-sequence]
   (‘+’, ‘-’).

integer-mould:
   [replicator] [‘s’] (‘z’, ‘d’)
   [([replicator] [‘s’] (‘z’, ‘d’), insertion)-sequence].

real-pattern:
   [sign-mould] [‘s’] ‘.’ [insertion] integer-mould
   [[‘s’]‘e’ [insertion] [sign-mould] integer-mould].

complex-pattern:
   real-pattern [‘s’]‘i’ [insertion] real-pattern.

bits-pattern:
   replicator ‘r’ integer-mould.

string-pattern:
   [replicator] [‘s’] ‘a’
   [([replicator] [‘s’] ‘a’, insertion)-sequence].

boolean-pattern:
   ‘b’,
   ‘b’ ‘(’ row-character-denotation ‘,’
           row-character-denotation ‘)’.

choice-pattern:
   ‘c’ ‘(’ row-character-denotation-list ‘)’.
```

```
format-pattern:
   ‘f’ meek-format-enclosed-clause.
```

B.8.2 Secondaries

```
secondary:
   primary,
   selection,
   generator.

selection:
   identifier ‘OF’ secondary.

generator:
   (‘HEAP’, ‘LOC’) actual-declarer.
```

B.8.3 Tertiaries

```
tertiary:
   secondary,
   nihil,
   formula,
   stowed-function.

nihil:
   ‘NIL’.

formula:
   monadic-operator-sequence firm-secondary,
   firm-factor dyadic-operator firm-factor.

factor:
   [monadic-operator-sequence] secondary, formula.

stowed-function:
   (‘DIAG’, ‘TRNSP’, ‘ROW’, ‘COL’) weak-tertiary,
   meek-integral-tertiary
   (‘DIAG’, ‘ROW’, ‘COL’) weak-tertiary.
```

B.8.4 Units

```
unit:
   tertiary,
   assignation,
   routine-text,
   identity-relation,
   jump,
   skip,
   assertion,
   conditional-function.

assignation:
   soft-tertiary ‘:=’ strong-unit.

identity-relation:
   soft-tertiary (‘IS’, ‘:=:’),
   (‘ISNT’, ‘:/=:’) soft-tertiary.

jump:
   [‘GOTO’] label.

skip:
   ‘SKIP’.

assertion:
   ‘ASSERT’ meek-boolean-enclosed-clause.

conditional-function:
   meek-boolean-tertiary
   ((‘THEF’, ‘ANDF’, ‘ANDTH’), (‘ELSF’, ‘ORF’, ‘OREL’))
   meek-boolean-tertiary.

routine-text:
   [‘(’ (formal-declarer identifier)-list ‘)’] formal-declarer ‘:’
   strong-unit.
```

B.9 Declarations

```
declaration:
   mode-declaration,
   identity-declaration,
   variable-declaration,
   procedure-declaration,
   procedure-variable-declaration,
   operator-declaration,
   priority-declaration.

mode-declaration:
   ‘MODE’ (mode-indicant ‘=’ actual-declarer)-list.

identity-declaration:
   formal-declarer (identifier ‘=’ strong-unit)-list.

variable-declaration:
   [‘HEAP’, ‘LOC’] actual-declarer
   (identifier [‘:=’ strong-unit])-list.

procedure-declaration:
   ‘PROC’ (identifier ‘=’ routine-text)-list.

procedure-variable-declaration:
   [‘HEAP’, ‘LOC’] ‘PROC’
   (identifier ‘:=’  routine-text)-list.

operator-declaration:
   ‘OP’ (operator ‘=’ routine-text)-list,
   ‘OP’ [‘(’ formal-declarer-list ‘)’ (formal-declarer, ‘VOID’)]
   (operator ‘=’ strong-unit)-list.

priority-declaration:
   ‘PRIO’ (operator ‘=’ priority-digit)-list.

priority-digit:
   ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’.
```

B.10 Declarers

Declarers describe modes. The context determines whether a mode must be formal, virtual, or actual. Formal- or virtual-declarers are needed where the size of rows is irrelevant. Actual declarers are needed where the size of rows must be known, for instance when allocating memory for rows.

```
victal:
   virtual,
   actual,
   formal.
```

```
victal-declarer:
   (‘LONG’-sequence, ‘SHORT’-sequence) primitive-declarer,
   mode-indicant,
   ‘REF’ virtual-declarer,
   ‘[’ victal-bounds-list ‘]’ victal-declarer,
   ‘FLEX’ ‘[’ victal-bounds-list ‘]’ victal-declarer,
   ‘STRUCT’ ‘(’ (victal-declarer identifier)-list ‘)’,
   ‘UNION’ ‘(’ (formal-declarer, ‘VOID’)-list ‘)’,
   ‘PROC’ [‘(’ formal-declarer-list ‘)’] (formal-declarer, ‘VOID’).

primitive-declarer:
   ‘INT’,
   ‘REAL’,
   ‘COMPL’,
   ‘COMPLEX’,
   ‘BOOL’,
   ‘CHAR’,
   ‘STRING’,
   ‘BITS’,
   ‘BYTES’,
   ‘FORMAT’,
   ‘FILE’,
   ‘PIPE’,
   ‘CHANNEL’,
   ‘SEMA’,
   ’SOUND’.

actual-bounds:
   [meek-integer-unit ‘:’] meek-integer-unit.

formal-bounds, virtual-bounds:
   [‘:’].
```

B.11 Pragments

Pragments are either pragmats or comments. Pragmats contain preprocessor directives, or set options from within an Algol 68 program.

```
pragment:
   pragmat, comment.

pragmat:
   pragmat-symbol pragmat-item-sequence pragmat-symbol.

pragmat-symbol:
   ‘PR’, ‘PRAGMAT’.

comment:
   comment-symbol character-sequence comment-symbol.

comment-symbol:
   ‘#’, ‘CO’, ‘COMMENT’.
```

B.12 Refinements

The refinement preprocessor is a low-level tool for top-down programming. It is not really a part of Algol 68 Genie syntax, but superimposed on top of it. Note that refinements interfere somewhat with labels.

```
refined program:
   paragraph ‘.’, refinement definition sequence.

refinement definition:
   identifier ‘:’  paragraph ‘.’.

paragraph:
   character-sequence.
```

Appendix C

Release history

Development of Algol 68 Genie started 1992. Algol 68 Genie was released under GNU GPL and made available for download in 2001.

Mark 14.1, November 2008

1. Implements zero replicators as required by the Revised Report
2. Decommissions (undocumented) VMS option syntax

Mark 14, October 2008

1. Adds first edition of the *Algol 68 Genie User Manual*
2. Adds Revised Report (HTML translation 1.2) to the documentation
3. Adds option `LINK` to the monitor

Mark 13, June 2008

1. Adds procedures to interrogate files and directories
2. Adds various operators and procedures
3. Adds `APROPOS` to command-line options and monitor options

Mark 12, May 2008

1. Improves code for the parallel clause

2. Improves interpreter efficiency
3. Improves breakpoint commands in the monitor
4. Adds options `UNTIL`, `FINISH` and `OUT` to the monitor
5. Adds Revised Report (HTML translation 1.1) to the documentation

Mark 11.1, March 2008

1. Adds options `RERUN`, `RESET` and `RESTART` to monitor
2. Adds Revised Report (HTML translation 1.0) to the documentation

Mark 11, November 2007

1. Adds support for sound
2. Adds support for Fourier transforms
3. Adds pseudo-operator `DIAG`
4. Adds pseudo-operator `TRNSP`
5. Adds pseudo-operator `COL`
6. Adds pseudo-operator `ROW`
7. Improves diagnostics in formatted transput
8. Improves monitor

Mark 10.2, April 2007

1. Adds procedure `real`
2. Adds format pattern `h`
3. Adds operator `OP SET = (INT, L BITS) L BITS`
4. Adds operator `OP CLEAR = (INT, L BITS) L BITS`

Mark 10.1, December 2006

1. Improves interpreter efficiency

Mark 10, August 2006

1. Adds basic linear algebra support
2. Adds procedures `arctan2`, `long arctan2` and `long long arctan2`
3. Adds procedure `execve output`
4. Improves diagnostics
5. Adds option [NO]BACKTRACE

Mark 9.2, July 2006

1. Adds typographical display features

Mark 9.1, May 2006

1. 64-bit safe interpreter
2. Improves monitor/debugger
3. Adds option MONITOR (or DEBUG)
4. Adds procedure `monitor` (or `debug`)

Mark 9, March 2006

1. Adds basic PostgreSQL support
2. Adds procedure `utc time` (UNIX)
3. Adds procedure `local time` (UNIX)
4. Improves diagnostic messages

Mark 8.1, November 2005

1. Adds procedure `sub in string` for substituting regular expressions in a string (UNIX)
2. Adds option `PORTCHECK` cf. standard hardware representation
3. Adds option `PEDANTIC`

Mark 8, July 2005

1. Adds procedure `http content` for fetching web page contents (UNIX)

2. Adds procedure `tcp request` for sending requests via TCP (UNIX)
3. Adds procedure `grep in string` for matching regular expressions in a string (UNIX)
4. Adds procedure `last char in string` to find last occurrence of a character in a string
5. Adds procedure `string in string` to find first occurrence of a string in a string
6. Adds keyword `UNTIL` to implement post-checked loop
7. Adds keyword `DOWNTO` to complement `TO`
8. Adds keyword `ANDTH` as alternative for `ANDF` and keyword `OREL` as alternative for `ORF`
9. Makes   a `SKIP` when used as a unit (cf. RR)

Mark 7, May 2005

1. Adds partial parametrisation following C. H. Lindsey's proposal
2. Adds elementary and trigonometric functions for complex numbers, independent of GSL

Mark 6, March 2005

1. Adds parallel clause on platforms that support POSIX threads

Mark 5, October 2004

1. Improves interpreter efficiency
2. Adds transput on `STRING` as well as `FILE` and `PIPE`
3. Adds dynamic scope checking
4. Adds basic preprocessor
5. Adds curses support

Mark 4, June 2004

1. Adds more GSL support
2. Adds modes `LONG BITS` and `LONG LONG BITS`
3. Makes parser accept `( .. )` as alternative for `[ .. ]`

4. Makes parser optionally treat `  ..   `, `[ ..   ]` and `( ..   )` as equivalent
5. Makes parser accept loop clause as encloses clause
6. Makes mode checker accept `UNION` with components relates through deflexing
7. Changes to thread-safe plotutils interface

Mark 3.2, March 2004

1. Adds overflow checks for primitive modes

Mark 3.1, January 2004

1. Adds optional system stack overflow handling to interpreter
2. Improves multiprecision library

Mark 3, September 2003

1. Adds formatted transput
2. Improves speed of multiprecision library

Mark 2, March 2003

1. Adds mode `BYTES` and `LONG BYTES`
2. Adds mode `FORMAT` and parsing of `FORMAT` texts
3. Adds straightening in unformatted transput
4. Adds mode `PIPE` and `UNIX` support
5. Changes to UNIX system level IO

Mark 1, November 2002

Beta releases, from November 2001 onward

Appendix D

Manual page

```
A68G(1)                  BSD General Commands Manual                 A68G(1)

NAME
     a68g - Algol 68 Genie, an Algol 68 interpreter

SYNOPSIS
     a68g [-apropos | -help | -info -[string]] [-assertions | -noassertions]
     [-backtrace | -nobacktrace] [-brackets] [-check | -norun]
     [-debug | -monitor] [-echo string] [-execute unit] [-extensive]
     [-frame number] [-handles number] [-heap number] [-listing] [-moids]
     [-overhead number] [-pragmats | -nopragmats] [-precision number]
     [-preludelisting] [-print unit] [-reductions] [-run]
     [-source | -nosource] [-stack number] [-statistics] [-terminal]
     [-timelimit number] [-trace | -notrace] [-tree | -notree] [-unused]
     [-verbose] [-version] [-warnings | -nowarnings] [-xref | -noxref]
     filename

DESCRIPTION
     Algol 68 Genie (Algol68G) is an Algol 68 interpreter. It can be used for
     executing Algol 68 programs or scripts. Algol 68 is a rather lean orthog-
     onal general-purpose language that is a beautiful means for denoting
     algorithms. Algol 68 was designed as a general-purpose programming lan-
     guage by IFIP Working Group 2.1 (Algorithmic Languages and Calculi) that
     has continuing responsibility for Algol 60 and Algol 68.

OPTIONS
     Options are passed to a68g either from the file .a68g.rc in the working
     directory, the environment variable A68G_OPTIONS, the command-line or
     from pragmats.

     Option precedence is as follows: pragmat options supersede command-line
     options, command-line options supersede options in environment variable
     A68G_OPTIONS, A68G_OPTIONS supersedes options in .a68g.rc.

     Listing options, tracing options and -pragmat, -nopragmat, take their
     effect when they are encountered in a left-to-right pass of the program
     text, and can thus be used, for example, to generate a cross reference
     for a particular part of the program.

     Options are not case sensitive. In the list below, uppercase letters are
     mandatory for recognition. For instance

         a68g -prec 200 -ti 60 ...

     means

         a68g -precision 200 -timelimit 60 ...

     Where numeric arguments are required, suffices k, M or G are allowed for
     multiplication with 2 ** 10, 2 ** 20 or 2 ** 30 respectively.

     -APropos | -Help | -INfo [string]
             Print info on options if string is omitted, or print info on
             string otherwise.

     -Assertions | -NOAssertions
             Control elaboration of assertions.

     -BACKtrace | -NOBACKtrace
             Control whether a stack backtrace is done in case a runtime-error
             occurs.

     -BRackets
             Consider [ .. ] and { .. } as being equivalent to ( .. ). Tradi-
             tional Algol 68 syntax allows ( .. ) to replace [ .. ] in bounds
             and slices.

     -Check | -NORun
             Check syntax only, the interpreter does not start.

     -DEBUG | -MONitor
             Start in the monitor. Invoke the monitor in case a runtime-error
             occurs; the program will pause in the monitor on the line that
             contains the error.

     -ECHO string
             Echo string to standout.

     -Execute unit
             Execute the Algol 68 unit. In this way one-liners can be executed
             from the command line.

     -EXTensive
             Generate an extensive listing.

     -FRAME number
             Set the frame stack size to number bytes.
```

```
    -HANDLES number
            Set the handle space size to number bytes.

    -HEAP number
            Set the heap size to number bytes.

    -LISTing
            Generate a concise listing.

    -MOIDS  Generate an overview of modes in the listing file.

    -OVERHEAD number
            Set overhead for stack checking.

    -PORTcheck | -NOPORTcheck
            Enable or disable portability warning messages.

    -PRagmats | -NOPRagmats
            Control elaboration of pragmats.

    -PRECision number
            Set the precision for LONG LONG modes to number significant dig-
            its.

    -PRELUDElisting
            Generate a listing of preludes.

    -Print unit
            Print the value yielded by the Algol 68 unit. In this way one-
            liners can be executed from the command line.

    -QUOTEstropping
            Use quote stropping.

    -REDuctions
            Print reductions made by the parser.

    -RUN    Override the norun option.

    -SOURCE | -NOSOURCE
            Control the listing of source lines in the listing file.

    -STACK number
            Set the stack size to number bytes.

    -STatistics
            Generate statistics in the listing file.

    -TImelimit number
            Interrupt the interpreter after number seconds, generating a time
            limit exceeded error.

    -TRace | -NOTRace
            Control tracing of the running program.
```

```

     -TREE | -NOTREE
             Control listing of the syntax tree in the listing file.

     -UNUSED
             Generate an overview of unused tags in the listing file.

     -UPPERstropping
             Use upper stropping, which is the default stropping regime.

     -VERBose
             Use verbose mode.

     -Version
             Print the version of the running image of a68g.

     -Warnings | -NOWarnings
             Enable warning messages or suppress suppressible warning mes-
             sages.

     -Xref | -NOXref
             Control generation of a cross-reference in the listing file.

AUTHOR
     Author of Algol68G is Marcel van der Veer <algol68g@xs4all.nl>.


BSD                            November 6, 2008                            BSD
```

Appendix E

ASCII table

Binary	Octal	Decimal	Hexadecimal	Character	Description
000 0000	000	0	00	^@	Null character `\0`
000 0001	001	1	01	^A	Start of Header
000 0010	002	2	02	^B	Start of Text
000 0011	003	3	03	^C	End of Text
000 0100	004	4	04	^D	End of Transmission
000 0101	005	5	05	^E	Enquiry
000 0110	006	6	06	^F	Acknowledgement
000 0111	007	7	07	^G	Bell `\a`
000 1000	010	8	08	^H	Backspace `\b`
000 1001	011	9	09	^I	Horizontal Tab `\t`
000 1010	012	10	0A	^J	Line feed `\n`
000 1011	013	11	0B	^K	Vertical Tab `\v`
000 1100	014	12	0C	^L	Form feed `\f`
000 1101	015	13	0D	^M	Carriage return `\r`
000 1110	016	14	0E	^N	Shift Out
000 1111	017	15	0F	^O	Shift In
001 0000	020	16	10	^P	Data Link Escape
001 0001	021	17	11	^Q	Device Control 1 (XON)
001 0010	022	18	12	^R	Device Control 2
001 0011	023	19	13	^S	Device Control 3 (XOFF)
001 0100	024	20	14	^T	Device Control 4
001 0101	025	21	15	^U	Negative Acknowledgement
001 0110	026	22	16	^V	Synchronous Idle
001 0111	027	23	17	^W	End of Transmission Block
001 1000	030	24	18	^X	Cancel
001 1001	031	25	19	^Y	End of Medium
001 1010	032	26	1A	^Z	Substitute
001 1011	033	27	1B	^[	Escape `\e`
001 1100	034	28	1C	^	File Separator
001 1101	035	29	1D	^]	Group Separator
001 1110	036	30	1E	^^	Record Separator
001 1111	037	31	1F	^_	Unit Separator
111 1111	177	127	7F	^?	Delete

Binary	Octal	Decimal	Hexadecimal	Character	Description
010 0000	040	32	20		Space
010 0001	041	33	21	!	
010 0010	042	34	22	”	
010 0011	043	35	23	#	
010 0100	044	36	24	$	
010 0101	045	37	25	%	
010 0110	046	38	26	&	
010 0111	047	39	27	’	
010 1000	050	40	28	(	
010 1001	051	41	29	)	
010 1010	052	42	2A	*	
010 1011	053	43	2B	+	
010 1100	054	44	2C	,	
010 1101	055	45	2D	-	
010 1110	056	46	2E	.	
010 1111	057	47	2F	/	
011 0000	060	48	30	0	
011 0001	061	49	31	1	
011 0010	062	50	32	2	
011 0011	063	51	33	3	
011 0100	064	52	34	4	
011 0101	065	53	35	5	
011 0110	066	54	36	6	
011 0111	067	55	37	7	
011 1000	070	56	38	8	
011 1001	071	57	39	9	
011 1010	072	58	3A	:	
011 1011	073	59	3B	;	
011 1100	074	60	3C	¡	
011 1101	075	61	3D	=	
011 1110	076	62	3E	¿	
011 1111	077	63	3F	?	

Binary	Octal	Decimal	Hexadecimal	Character	Description
100 0000	100	64	40	@	
100 0001	101	65	41	A	
100 0010	102	66	42	B	
100 0011	103	67	43	C	
100 0100	104	68	44	D	
100 0101	105	69	45	E	
100 0110	106	70	46	F	
100 0111	107	71	47	G	
100 1000	110	72	48	H	
100 1001	111	73	49	I	
100 1010	112	74	4A	J	
100 1011	113	75	4B	K	
100 1100	114	76	4C	L	
100 1101	115	77	4D	M	
100 1110	116	78	4E	N	
100 1111	117	79	4F	O	
101 0000	120	80	50	P	
101 0001	121	81	51	Q	
101 0010	122	82	52	R	
101 0011	123	83	53	S	
101 0100	124	84	54	T	
101 0101	125	85	55	U	
101 0110	126	86	56	V	
101 0111	127	87	57	W	
101 1000	130	88	58	X	
101 1001	131	89	59	Y	
101 1010	132	90	5A	Z	
101 1011	133	91	5B	[	
101 1100	134	92	5C	\	
101 1101	135	93	5D	]	
101 1110	136	94	5E	ˆ	
101 1111	137	95	5F	_	

Binary	Octal	Decimal	Hexadecimal	Character	Description
110 0000	140	96	60	‘	
110 0001	141	97	61	a	
110 0010	142	98	62	b	
110 0011	143	99	63	c	
110 0100	144	100	64	d	
110 0101	145	101	65	e	
110 0110	146	102	66	f	
110 0111	147	103	67	g	
110 1000	150	104	68	h	
110 1001	151	105	69	i	
110 1010	152	106	6A	j	
110 1011	153	107	6B	k	
110 1100	154	108	6C	l	
110 1101	155	109	6D	m	
110 1110	156	110	6E	n	
110 1111	157	111	6F	o	
111 0000	160	112	70	p	
111 0001	161	113	71	q	
111 0010	162	114	72	r	
111 0011	163	115	73	s	
111 0100	164	116	74	t	
111 0101	165	117	75	u	
111 0110	166	118	76	v	
111 0111	167	119	77	w	
111 1000	170	120	78	x	
111 1001	171	121	79	y	
111 1010	172	122	7A	z	
111 1011	173	123	7B	{	
111 1100	174	124	7C	—	
111 1101	175	125	7D	}	
111 1110	176	126	7E	˜	

Appendix F

GNU General Public License

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange,

for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

   (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU

Appendix G

GNU Free Documentation License

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-

4. MODIFICATIONS

arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the com-

Appendix H

Bibliography

For other informal texts on the language, see for instance:

1. D. F. Brailsford, A. N. Walker. Introductory Algol 68 programming. Ellis Horwood Ltd, Chichester [1979].

2. C. H. Lindsey and S. G. van der Meulen, Informal Introduction to Algol 68, North-Holland [1977].

3. A. D. McGettrick. Algol 68: a first and second course. Cambridge University Press [1978].

4. C. H. A. Koster, H. Meijer. Systematisch programmeren in Algol 68. Deel 1, 2. Kluwer, Deventer [1978][1981].

The Revised Report is the final arbiter of what constitutes Algol 68. This publication ranks among the difficult publications of computer science.

1. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker (editors), Revised Report on the Algorithmic Language Algol 68, Springer-Verlag [1976].

a68g implements extensions that are documented in following material:

1. C. H. Lindsey. Specification of partial parametrization proposal. Algol Bulletin 39.3.1 pages 6-9. Available from the internet.

2. S. G. van der Meulen, M. Veldhorst. TORRIX - A programming language for operations on vectors and matrices over arbitrary fields and of variable size. Rijksuniversiteit Utrecht [1977].

Some interesting material on the history of Algol 68:

1. C. H. A. Koster, The Making of Algol 68 [1996].
   http://www.cs.ru.nl/ kees/home/biblio.html

2. C. H. Lindsey, A History of Algol 68. ACM SIGPLAN Notices **28**(3) 97 [1993].

Online information on Algol 68

1. The Algol 68 Genie web pages provide a copy of the Revised Report in HTML that is based on the version prepared by W.B. Kloke.
   http://www.xs4all.nl/~jmvdveer/algol.html

2. Greg Nunan maintains a page on Algol 68 web resources.
   http://www.algol68.org/

3. HOPL - History of programming languages. On-line encyclopaedia of programming languages has an entry for Algol 68.
   http://hopl.murdoch.edu.au/

4. Karl Kleine collects and posts Algol 68 historic documents.
   http://www.fh-jena.de/~kleine/history/history.html

5. Brian Wichmann maintains The Algol Bulletin on-line archive.
   http://archive.computerhistory.org/resources/text/algol/algol_bulletin/

6. The Algol 68 directory from the Open Directory Project.
   http://dmoz.org/Computers/Programming/Languages/Algol_68/

7. The Wikipedia Project has entries for Algol 68, Algol 68 Genie, Adriaan van Wijngaarden, and two-level grammars.
   http://en.wikipedia.org/

8. Dick Grune has posted an Algol 68 program collection, including the MC Test Set.
   http://www.cs.vu.nl/pub/dick/Algol68/

Next to Algol 68 genie, other implementations are still available:

1. *ALGOL68S* compiler by Charles Lindsey.
   http://www.cs.man.ac.uk/~chl/

2. Algol 68 to C translator *a68toc*, a ported compiler by Sian Leitch. The front-end essentially is the *ALGOL68RS* portable compiler originally written by the Defence Research Agency when it was known as the RSRE (Royal Signals and Radar Establishment) but formats are unavailable. *ALGOL68RS* did not implement the parallel-clause. The *a68toc* distribution includes the TEX source of *Programming Algol 68 Made Easy* which is available under the GNU General Public License. The *a68toc* distribution is available from Greg Nunan's site
   http://www.algol68.org/

This manual refers to legacy implementations of Algol 68:

1. *CONTROL DATA ALGOL 68*. Zoethout et al., CDC Netherlands 1974. Full implementation (apparently it only lacked the format-pattern), used mainly in teaching in Germany and the Netherlands on CDC mainframes.
2. *ALGOL68C*. Bourne et al., Cambridge 1980. Subset implementation for IBM mainframes, DECsystems 10 and 20, VAX/UNIX, CYBER 205 and other systems. Compiler front-end that generated code for a ZCODE back-end.
3. *ALGOL68R*. Currie et al., RSRE (Royal Signals and Radar Establishment, Malvern) 1970. Single-pass compiler, used extensively in military and scientific circles in the UK on ICL machines.
4. *ALGOL68S*. Hibbard et al., UK and Carnegie Mellon 1976. Subset, ported to small machines as the PDP-11, (apparently) mostly used in teaching.
5. *ALGOL68RS*. For ICL 2900, VAX/VMS and other systems.
6. *FLACC*. Full Language Algol 68 Checkout Compiler. Mailloux et al., Edmonton, Canada 1978. An interpreter for the full language, available on IBM mainframes and used in teaching, both in North America and in Europe.

This manual contains smaller or larger parts from different open sources:

1. *Programming Algol 68 Made Easy* by Sian Leitch (GNU GPL).
2. GNU Compiler Collection (GNU GPL).
3. GNU Debugger (GNU GPL).
4. GNU C Library (GNU GPL).
5. GNU Scientific Library documentation (GNU GPL).
6. GNU Plotting Utilities documentation (GNU GPL).
7. PostgreSQL documentation (BSD license).
8. Wikipedia (GNU FDL).
9. Biografisch Woordenboek van Nederlandse Wiskundigen (Public domain).

This manual refers to various articles not related to Algol 68:

1. J.A. Barker and D. Henderson. *What is "liquid"? Understanding the states of matter*. Reviews of Modern Physics **48**(4) 587 [1976].
2. R.J. Nemiroff. *Visual Distortions Near a Black Hole and Neutron Star*. American Journal of Physics **61** 619 [1993].

Index