

The GNU Algol 68 Compiler

For GCC version 16.0.1 (pre-release)

(GCC)

Jose E. Marchesi

Copyright © 2025-2026 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Short Contents

1	Invoking ga68	1
2	Composing programs	4
3	Comments and pragmat	14
4	Hardware representation	16
5	Standard prelude	25
6	Extended prelude	39
7	POSIX prelude	41
8	Language extensions	46
	GNU General Public License	48
	GNU Free Documentation License	59
	Option Index	67
	Index	68

Table of Contents

1	Invoking ga68	1
1.1	Dialect options	1
1.2	Options for Directory Search	1
1.3	Module search options	1
1.4	Warnings options	2
1.5	Runtime options	2
1.6	Linking options	3
1.7	Developer options	3
2	Composing programs	4
2.1	Packets	4
2.2	Modules	4
2.2.1	Writing modules	5
2.2.2	Accessing modules	5
2.2.2.1	Accessing several modules	6
2.2.2.2	The controlled clause	7
2.2.2.3	The value yielded by an access clause	7
2.2.2.4	Modules accessing other modules	7
2.2.3	Module activation	8
2.2.4	Modules and exports	10
2.2.5	Modules and libraries	12
2.2.6	Modules and protection	12
2.3	Particular programs	12
2.3.1	Exit status	12
2.3.2	The <code>stop</code> label	12
2.4	The standard environment	13
3	Comments and pragmat	14
3.1	Comments	14
3.2	Pragmats	14
3.2.1	pragmat include	15
4	Hardware representation	16
4.1	Representation languages	16
4.2	Worthy characters	17
4.3	Base characters	17
4.4	Stropping regimes	18
4.4.1	POINT stropping	19
4.4.2	RES stropping	20
4.4.3	UPPER stropping	21
4.4.4	SUPPER stropping	22
4.5	Monads and Nomads	23
4.6	String breaks	24

5	Standard prelude	25
5.1	Environment enquiries	25
5.2	Standard modes	26
5.3	Standard priorities	27
5.4	Rows operators	28
5.5	Boolean operators	29
5.6	Integral operators	29
5.6.1	Arithmetic	29
5.6.2	Arithmetic combined with assignation	30
5.6.3	Relational	30
5.6.4	Conversion	31
5.7	Real operators	31
5.7.1	Arithmetic	31
5.7.2	Arithmetic combined with assignation	32
5.7.3	Relational	32
5.7.4	Conversions	33
5.8	Character operators	33
5.8.1	Relational	33
5.8.2	Conversions	34
5.9	String operators	34
5.9.1	Relational	34
5.9.2	Composition	35
5.9.3	Composition combined with assignation	35
5.10	Complex operators	35
5.11	Bits operators	35
5.11.1	Logical	35
5.11.2	Shifting	36
5.11.3	Relational	36
5.11.4	Conversions	36
5.12	Bytes operators	37
5.13	Semaphore operators	37
5.14	Math procedures	37
5.14.1	Arithmetic	37
5.14.2	Logarithms	37
5.14.3	Trigonometric	37
6	Extended prelude	39
6.1	Extended priorities	39
6.2	Extended environment enquiries	39
6.3	Extended rows operators	39
6.4	Extended boolean operators	40
6.5	Extended bits operators	40
6.6	Extended math procedures	40
6.6.1	Logarithms	40

7	POSIX prelude	41
7.1	POSIX process	41
7.2	POSIX command line	41
7.3	POSIX environment	41
7.4	POSIX errors	41
7.5	POSIX files	42
7.5.1	Standard file descriptors	42
7.5.2	Opening and closing files	42
7.5.3	Creating files	42
7.5.4	Flags for <code>fopen</code>	43
7.5.5	Getting file properties	43
7.6	POSIX sockets	44
7.7	POSIX string transport	44
7.7.1	Output of strings and chars	44
7.7.2	Input of strings and chars	44
8	Language extensions	46
8.1	<code>bin</code> and <code>abs</code> of negative integral values	46
8.2	Bold taggles	46
	GNU General Public License	48
	GNU Free Documentation License	59
	ADDENDUM: How to use this License for your documents	66
	Option Index	67
	Index	68

1 Invoking ga68

The `ga68` command is the GNU compiler for the Algol 68 language and supports many of the same options as `gcc`. See Section “Option Summary” in *Using the GNU Compiler Collection (GCC)*. This manual only documents the options specific to `ga68`.

1.1 Dialect options

The following options control how the compiler handles certain dialect variations of the language.

-std=std Specify the standard to which the program is expected to conform, which may be one of ‘`algol68`’ or ‘`gnu68`’. The default value for *std* is ‘`gnu68`’, which specifies a strict super language of Algol 68 allowing GNU extensions. The ‘`algol68`’ value specifies that the program strictly conform to the Revised Report.

-fstropping=stropping_regime
Specify the stropping regime to expect in the input programs. The default value for *stropping_regime* is ‘`supper`’, which specifies the modern SUPPER stropping which is a GNU extension. The ‘`upper`’ value specifies the classic UPPER stropping of Algol 68 programs. See Section 4.4 [Stropping regimes], page 18.

-fbrackets
This option controls whether `[. .]` is considered equivalent to `(. .)`. This syntactic variation is blessed by the Revised Report and is still strict Algol 68. This option is disabled by default.

1.2 Options for Directory Search

These options specify directories to search for files, libraries, and other parts of the compiler:

-Idir Add the directory *dir* to the list of directories to be searched for files when processing the Section 3.2.1 [pragmat include], page 15. Multiple `-I` options can be used, and the directories specified are scanned in left-to-right order, as with `gcc`. The directory will also be added to the list of directories to be searched for module interface-definitions Section 2.2.3 [Module activation], page 8.

-Ldir Add the directory *dir* to the list of directories to be searched for module interface-definitions Section 2.2.3 [Module activation], page 8. Multiple `-L` options can be used, and the directories specified are scanned in left-to-right order, as with `gcc`. The directory will also be added to the list of library search directories, as with `gcc`.

1.3 Module search options

The following options can be used to tell the compiler where to look for certain modules.

-fmodules-map=string
Use the mapping between module indicants and module base filenames specified in *string*, which must contain a sequence of entries with form

basename=moduleindicant[,moduleindicant]... separated by colon (:) characters.

When a module *moduleindicant* is accessed, the compiler will look for exports information for it in files *basename.m68*, *libbasename.so*, *libbasename.a*, *basename.o*, in that order.

This option is used to avoid the default behavior, in which the *basename* used to search for an accessed module is implicitly derived from its *indicant*, by transforming it to lower case.

The effect of this option is accumulative.

-fmodules-map-file=<filename>

Like **-fmodules-map**, but read the mapping information from the file *<filename>*.

1.4 Warnings options

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there is likely to be a bug in the program. Unless **-Werror** is specified, they do not prevent compilation of the program.

-Wvoiding

Warn on non-void units being voided due to a strong context.

-Wscope

Warn when a potential name scope violation is found.

-Whidden-declarations=level

Warn when a declaration hides another declaration in a larger reach. This includes operators that hide firmly related operators defined in larger reach.

-Whidden-declarations=none

At this level no warning is issued for any hidden declaration on an outer scope.

-Whidden-declarations=prelude

At this level, warnings are issued for hidden declarations defined in the standard prelude. This is the default warning level of **-Whidden-declarations**.

-Whidden-declarations=all

At this level, warnings are issued for any and all hidden declarations.

-Wextensions

Warn when a non-portable Algol 68 construct is used, like GNU extensions to Algol 68.

1.5 Runtime options

These options affect the runtime behavior of programs compiled with **ga68**.

-fno-assert

Turn off code generation for **ASSERT** contracts.

-fcheck=<keyword>

Enable the generation of run-time checks; the argument shall be a comma-delimited list of the following keywords. Prefixing a check with **no-** disables it if it was activated by a previous specification.

'all' Enable all run-time test of **-fcheck**.

'none' Disable all run-time test of **-fcheck**.

'nil' Check for nil while dereferencing.

'bounds' Enable generation of run-time checks when indexing and trimming multiple values.

1.6 Linking options

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

-shared-libga68

On systems that provide **libga68** as a shared and a static library, this option forces the use of the shared version. If no shared version was built when the compiler was configured, this option has no effect.

-static-libga68

On systems that provide **libga68** as a shared and a static library, this option forces the use of the static version. If no static version was built when the compiler was configured, this option has no effect. This is the default.

1.7 Developer options

This section describes command-line options that are primarily of interest to developers.

-fa68-dump-modes

Output a list of all the modes parsed by the front-end.

-fa68-dump-ast

Dump a textual representation of the parse tree.

-fa68-dump-module-interfaces

Dump the interfaces of module definitions in the compiled packet.

2 Composing programs

This chapter documents how to compose full Algol 68 programs using the modules and separated compilation support provided by this compiler.

2.1 Packets

Each Algol 68 source file, which are files using the file extension `.a68`, contains the definition of a so-called *packet*. Packets therefore play the role of *compilation units*, and each packet can be compiled separately to an object file. A set of compiled object files can then be linked in the usual fashion into an executable, archive or shared object by the system linker, without the need of any language-specific link editor or build system.

This compiler supports two different kind of packets:

- *Particular programs* constitute the entry point of a program. They roughly speaking correspond to the `main` function of other languages like C.

See Section 2.3 [Particular programs], page 12.

- *Prelude packets* contain the definition of *modules*, which *publicize* definitions of modes, procedures, variables, operators and even the publicized definitions of other modules. Each prelude packet defines a single packet, defined at the top-level, which can be accessed by other packets (be them particular programs or other prelude packets) via an **access** construct. Prelude packets are so-called because their contents get conceptually stuffed in the *user-prelude* in the case of user-defined modules, or the *library-prelude* in the case of module packets provided by the compiler. They are usually used to compose libraries that can be used in a bottom-up fashion.

See Section 2.2 [Modules], page 4.

Future versions of this compiler will eventually support a third kind of packet, oriented to top-down development:

- *Stuffing packets* contain the definition of an *actual hole*, an **egg** construct, that can be stuffed in a matching *formal hole* in another package via a **nest** construct. Formal holes are used in order to achieve separated compilation in a top-bottom fashion, and also to invoke procedures written in other languages, such as C functions or Fortran subroutines.

A *collection of packets*, all of which must be compatible with each other, constitutes either a *program* or a *library*. Exactly one of the packets constituting a program shall be a particular program. In libraries at least one packet must be a prelude packet.

2.2 Modules

Definition modules, often referred as just *modules* in the sequel, fulfill two different but related purposes. On one side, they provide some degree of *protection* by preventing accessing indicators defined within the module but not explicitly publicized. On the other, they allow to define *interfaces*, allow separated compilation based on these interfaces, and conform libraries.

Modules are usually associated with bottom-up development strategies, where several already written components are grouped and combined to conform bigger components.

2.2.1 Writing modules

A *definition module* is a construct that provides access to a set of publicized definitions. They appear in the outer reach of a prelude packet and constitute its only contents (see Section 2.1 [Packets], page 4). They are composed of a prelude and a postlude. The publicized definitions appear in the module's prelude.

Consider for example the following definition module, which implements a very simple logging facility:

```

module Logger =
def int fd = stderr;
    pub string originator;
    pub proc log = (string msg) void:
        fputs (fd, (originator /= "" | ": ") + msg + "'n");

    log ("beginning of log'n")
postlude
    log ("end of log'n")
fed

```

The *module text* delimited by **def** and **fed** gets ascribed to the module indicator **Logger**. Module indicators are bold tags.

The *prelude* of the module spans from **def** to either **postlude**, or to **fed** in case of modules not featuring a postlude. It consists of a restricted serial clause in a void strong context, which can contain units and declarations, but no labels or completers. The declarations in the prelude may be either publicized or no publicized. As we shall see, publicized indicators are accessible within the reach of the defining module publicizing them. Publicized declarations are marked by preceding them with **pub**.

In our example the module prelude consists of three declarations and one unit. The tag **fd** is not publicized and is to be used internally by the module. The indicators **originator** and **log**, on the other hand, are publicized and conform the interface of the module. Note how the range of the prelude also covers the postlude: the **log** procedure is reachable there, as it would be **fd** as well.

The *postlude* of the module is optional and spans from **postlude** to **fed**. It consists of a serial clause in a **void** strong context, where definitions, labels and module accesses are not allowed, just units.

2.2.2 Accessing modules

Once a module is defined (see Section 2.2.1 [Writing modules], page 5) it can be accessed from other packets using an *access clause*. The access clause identifies the modules to access and then makes the publicized definitions of these modules visible within a *control clause*.

For example, this is how we could use the logger definition module defined in a previous section to log the progress of some particular program that reads an input file and writes some output file:

```

access Logger
begin { Identify ourselves with the program name }
    originator := argv (1);

```

```

    { Read input file.  }
    if NOT parse_input (argv (2))
    then log ("error parsing input file"); stop fi;

    { Write output file.  }
    if NOT write_output (argv (3))
    then log ("error writing output file"); stop fi;

    log ("success")
end

```

In this case the controlled clause is the closed clause conforming the particular program, and the definitions publicized by the logger module, in this case `originator` and `log`, can be used within it.

2.2.2.1 Accessing several modules

An access clause is not restricted to just provide access to a single module: any number of module indicators can be specified in an access clause. Suppose that our example processing program has to read and write the data in JSON format, and that a suitable JSON library is available in the form of a reachable module. We could then make both logger and json modules accessible like this:

```

access Logger, JSON
begin { Identify ourselves with the program name }
    originator := argv (1);

    JSONVal data;

    { Read input file.  }
    if data := json_from_file (argv (2));
        data = json_no_val
    then log ("error parsing input file"); stop fi;

    { Write output file.  }
    if not json_to_file (argv (3), data)
    then log ("error writing output file"); stop fi;

    log ("success")
end

```

In this version of the program the access clause includes the module indicator **JSON**, and that makes the mode indicator `jsonval` and the tags `json_no_val`, `json_from_file` and `json_to_file` visible within the program's closed clause.

Note that the following two access clauses are not equivalent:

```

access Logger, JSON C ... C;
access Logger access JSON C ... C;

```

In the first case, a compilation time error is emitted if there is a conflict among the publicized definitions of both modules; for example, if both modules were to publicize a procedure

called `log`. In the second case, the declaration of `log` publicized by **Logger** would hide the declaration of `log` publicized by **JSON**. This also has implications related to activation, that we will be discussing in a later section.

2.2.2.2 The controlled clause

The controlled clause in an access clause doesn't have to be a serial clause, like in the examples above. It can be any enclosed clause, like for example a loop clause:

```
proc frobnicate frobs = ([Frob frobs] void:
  access Logger to UPB frobs
    do log ("frobnicating " + name of frob);
    frobnicate (frob)
  od
```

2.2.2.3 The value yielded by an access clause

The elaboration of an access clause yields a value, which is the value yielded by the elaboration of the controlled clause. Since the latter is an enclosed clause, coercions get passed into them whenever required, the usual fashion.

We can see an example of this in the following procedure, whose body is a controlled closed clause that yields a **real** value:

```
proc incr factor = (ref[real] factors, int idx) real:
  access logger (log ("factor increased"); factors[idx] += 1.0)
```

Note how the access clause above is in a strong context requiring a value of mode **real**. The value yielded by the access clause is the value yielded by the controlled enclosed clause, which in this case is a closed clause. The strong context and required mode gets passed to the last unit of the closed clause (the assignation) which in this case yields a value of mode **ref real**. The unit is coerced to **real** by dereferencing, and the resulting value becomes the value yielded by the access clause.

2.2.2.4 Modules accessing other modules

Up to this point we have seen particular programs accessing modules, but a definition module may itself access other modules. This is done by placing the module text as a controlled clause of an access clause. Suppose we rewrite our logger module so it uses JSON internally to log JSON objects rather than raw strings. We could do it this way:

```
module Logger =
  access JSON
  def int fd = stderr;
  pub string originator;
  pub proc log = (string msg) void:
    fputs (fd, json_array (json_string (originator),
                                json_string (msg)));

    log (json_string ("beginning of log'n"))
  postlude
    log (json_string ("end of log'n"))
  fed
```

Note how this version of **Logger** uses a few definitions publicized by the **JSON** module.

A program accessing **Logger** will not see the definitions publicized by the **JSON** module. If that is what we intended, for example to allow the users of the logger to tweak their own JSON, we would need to specify it this way:

```
module Logger =
  access pub JSON
  def c ... as before... c fed
```

In this version the definitions publicized by **JSON** become visible to programs accessing **Logger**.

2.2.3 Module activation

In all the examples seen so far the modules were accessed just once. In these cases, accessing the module via an access-clause causes the *activation* of the module.

Activating a module involves elaborating all the declarations and units that conform its prelude. Depending on the particular module definition that gets activated, this may involve all sort of side effects, such as allocating space for values and initializing them, opening files, *etc.* Once the modules specified in the access clause are successfully activated, the controlled clause gets elaborated itself, within the reach of all the publicized definitions by the activated modules as we saw in the last section. Finally, once the controlled clause has been elaborated, the module gets *revoked* by elaborating the serial clause in its postlude.

However, nothing prevents some given definition module to be accessed more than once in the same program. The following program, that makes use of the **logger** module, exemplifies this:

```
access Logger
begin originator := argv (1);
  log ("executing program");
  c ... c
  access Logger (originator := argv (1) + ":subtask";
    log ("doing subtask")
    c ... c)
end
```

In this program the module **Logger** is accessed twice. The code is obviously written assuming that the inner access clause triggers a new activation of the **Logger** module, allocating new storage and executing its prelude. This would result in the following log contents:

```
a.out: beginning of log
a.out: executing program
a.out:subtask: beginning of log
a.out:subtask: doing subtask
a.out:subtask: end of log
a.out: end of log
```

However, this is not what happens. The module gets only activated once, as the result of the outer access clause. The inner access clause just makes the publicized indicators visible in its controlled clause. The actual resulting log output is:

```
a.out: beginning of log
```

```

a.out: executing program
a.out:subtask: doing subtask
a.out:subtask: end of log

```

Which is not what we intended. Modules are not classes. If we wanted the logger to support several originators that can be nested, we would need to add support for it in the definition module. Something like:

```

module Logger =
  def int fd = stderr, max_originators = 10;
  int orig := 0;
  [max_originators]string originators;

  pub proc push_originator = (string str) void:
    (assert (orig < max_originators);
     orig += 1;
     originators[orig] := str);
  pub proc pop_originator = void:
    (assert (max_originators > 0);
     orig -= 1);
  pub proc log = (string msg) void:
    fputs (fd, (originator[orig] /= "" | ": ") + msg + '\n');

  log ("beginning of log\n")
postlude
  log ("end of log\n")
fed

```

Note how in this version of **Logger** `originators` acts as a stack of originator strings, and it is not publicized. The management of the stack is done via a pair of publicized procedures `push_originator` and `pop_originator`. Our program will now look like:

```

access Logger
begin push_originator (argv (1));
  log ("executing program");
  c ... c
  access logger (push_originator ("subtask");
    log ("doing subtask")
    c ... c;
    pop_originator)
end

```

And the log output is:

```

a.out: beginning of log
a.out: executing program
a.out:subtask: doing subtask
a.out: end of log

```

2.2.4 Modules and exports

As we have seen, each Algol 68 source file contains either a particular program or a prelude packet. Prelude packets consist on the definition of a single top-level module, that is itself identified by a module indicant.

Consider for example a source file called `trilean.a68` that implements strong Kleene three-valued (“trilean”) logic. It does so by the mean of a definition module called **Trilean**. A sketch of such a file may look like this:

```

module Trilean =
def
    pub mode Tril = int;

    pub Tril dontknow = 0, yes = 1, no = 2;

    pub prio AND3 = 3, OR3 = 3, XOR3 = 3;

    pub op NOT3 = (Tril a) Tril:
        (a + 1 | dontknow, no, yes);

    C ... other definitions ... C

skip
fed

```

The module indicant **Trilean** identifies the module. If we now compile this file to an object file using GCC:

```
$ ga68 -c trilean.a68
```

The result of the compilation is an object file `trilean.o`, plus some *exports information* which is placed in a section in the object file, named `.a68_exports`. The exports information describes the interface provided by the **Trilean** module defined in the compilation unit. This includes all the modes, identifiers, priorities, operators, etc, that are publicized by the module. The particular encoding used to hold these exports is highly compact and not easy readable by persons; instead, it is designed to be read back by GCC when it builds another compilation unit that, in turn, needs to access the **Trilean** module.

Consider the following sketched particular program that resides in a source file `main.a68`, and that uses trilean logic:

```

access Trilean
begin
    C ... C
end

```

When this program gets compiled by GCC using `ga68 -c program.a68`, the compiler finds the access clause and needs to locate some exports for the module **Trilean**. To do so, it searches in the modules search path, composed by the current working directory, some system directories and all directories specified in `-I` and `-L` options, looking for files called `trilean.m68`, `trilean.so`, `trilean.a` and `trilean.o`, in that order, where:

`trilean.m68`

Is a stand-alone file expected to contain export data for one or more modules.

2.2.5 Modules and libraries

XXX

As we have seen modules are accessed by referring to them in access-clauses, using the same sort of bold-word indicants that identify user-defined modes and operators, such as `JSON`, `Transput` or `LEB128_Arithmetic`.

2.2.6 Modules and protection

XXX

2.3 Particular programs

An Algol 68 *particular program* consists of an enclosed clause in a strong context with target mode **void**, possibly preceded by a set of zero or more labels. For example:

```
hello:
begin puts ("Hello, world!\n")
end
```

Note that the enclosed clause conforming the particular program doesn't have to be a closed clause. Consider for example the following program, that prints out its command line arguments:

```
for i to argc
do puts (argv (i) + "\n") od
```

2.3.1 Exit status

Some operating systems have the notion of *exit status* of a process. In such systems, by default the execution of the particular program results in an exit status of success. It is possible for the program to specify an explicit exit status by using the standard procedure `posix exit`, like:

```
begin { ... program code ... }
if error found;
then posix exit (1) fi
end
```

In POSIX systems the status is an integer, and the system interprets a value other than zero as a run-time error. In other systems the status may be of some other type. To support this, the `posix exit` procedure accepts as an argument an untyped value that accommodates all the supported systems.

The following example shows a very simple program that prints “Hello world” on the standard output and then returns to the operating system with a success status:

```
begin puts ("Hello world\n")
end
```

2.3.2 The stop label

A predefined label named **stop** is defined in the standard postlude. This label can be jumped to at any time by a program and it will cause it to terminate and exit. For example:

```
begin if argc /= 2
```

```

        then puts ("Program requires exactly two arguments.");
    goto stop
    fi
    C ... C
end

```

2.4 The standard environment

The environment in which particular programs run is expressed here in the form of pseudo code:

```

(c standard-prelude c;
 c library-prelude c;
 c system-prelude c;
 par begin c system-task-1 c,
           c system-task-2 c,
           c system-task-n c,
           c user-task-1 c,
           c user-task-2 c,
           c user-task-m c
        end)

```

Where each user task consists of:

```

(c particular-prelude c;
 c user-prelude c;
 c particular-program c;
 c particular-postlude c)

```

The only standard system task described in the report is expressed in pseudo-code as:

```
do down gremlins; undefined; up bfileprotect od
```

Which denotes that, once a book (file) is closed, anything may happen. Other system tasks may exist, depending on the operating system. In general these tasks in the parallel clause denote the fact that the operating system is running in parallel (intercalated) with the user's particular programs.

- The library-prelude contains, among other things, the prelude parts of the defining modules provided by library.
- The particular-prelude and particular-postlude are common to all the particular programs.
- The user-prelude is where the prelude parts of the defining modules involved in the compilation get stuffed. If no defining module is involved then the user-prelude is empty.

Subsequent sections in this manual include a detailed description of the contents of these preludes.

3 Comments and pragmat

Comments and pragmat, also known collectively as *pragments*, can appear almost anywhere in an Algol 68 program. Comments are usually used for documentation purposes, and pragmat contain annotations for the compiler. Both are handled at the lexical level.

3.1 Comments

In the default modern stropping regime supported by GCC comments are written between { and } delimiters, and can be nested to arbitrary depth. For example:

```
foo += 1; { Increment foo. }
```

If UPPER stropping is selected, this compiler additionally supports three classical Algol 68 comment styles, in which the symbols marking the beginning of comments are the same than the symbols marking the end of comments and therefore can't be nested: **comment** ... **comment**, **co** ... **co** and **#** .. **#**. For example:

```
comment
    This is a comment.
comment

foo := 10; co this is also a comment co
foo += 1; # and so is this. #
```

Unless `-std=algol68` is specified in the command line, two styles of nestable comments can be also used with UPPER stropping: the already explained { ... } and a “bold” style that uses **code** ... **edoc**. For example:

```
foo := 10; { this is a nestable comment in brief style. }
foo += 1; note this is a nestable comment in bold style. eton.

note
    "Bold" nestable comments.
eton

{ "Brief" nestable comments. }
```

In UPPER stropping all comment styles are available, both classic and nestable. In modern SUPPER stropping, which is based on reserved words, only { ... } is available.

3.2 Pragmats

Pragmats (also known as *pragmas* in other programming languages) are directives and annotations for the compiler, and their usage impacts the compilation process in several ways. A pragmat starts with either **pragmat** or **pr** and finished with either **pragmat** or **pr** respectively. Pragmats cannot be nested. For example:

```
pr include "foo.a68" pr
```

The interpretation of pragmats is compiler-specific. This chapter documents the pragmats supported by GCC.

3.2.1 pragmat include

An *include pragmat* has the form:

```
pr include "PATH" pr
```

Where **PATH** is the path of the file whose contents are to be included at the location of the pragmat. If the provided path is relative then it is interpreted as relative to the directory containing the source file that contains the pragmat.

The **-I** command line option can be used in order to add additional search paths for **include**.

4 Hardware representation

The *reference language* specified by the Revised Report describes Algol 68 particular programs as composed by *symbols*. However, the Report leaves the matter of the concrete representation of these symbols, the *representation language*, open to the several implementations. This was motivated by the very heterogeneous computer systems in existence at the time the Report was written, which made flexibility in terms of representation a crucial matter.

This flexibility was indeed exploited by the early implementations, and there was a price to pay for it. A few years after the publication of the Revised Report the different implementations had already given rise to a plethora of many related languages that, albeit being strict Algol 68, differed considerably in appearance. This, and the fact that people were already engrossed in writing programs other than compilers that needed to process Algol 68 programs, such as code formatters and macro processors, prompted the WG 2.1 to develop and publish a *Report on the Standard Hardware Representation for ALGOL 68*, which came out in 1975.

This compiler generally follows the Standard Hardware Representation, but deviates from it in a few aspects. This chapter provides an overview of the hardware representation and documents any deviation.

4.1 Representation languages

A program in the strict Algol 68 language is composed by a series of symbols. These symbols have names such as `letter-a-symbol` and `assigns-to-symbol` which are, well, purely symbolic. In fact, these are notions in the two-level grammar that defines the strict language.

A *representation language* provides a mapping between symbols in the strict language and the representation of these symbols. Each representation is a sequence of syntactic marks. For example, the `completion symbol` may be represented by **exit**, where the marks are the bold letters. The `tilde symbol` may be represented by `~`, which is a single mark. The representation of `assigns to symbol` is `:=`, which is composed by the two marks `:` and `=`. The representation of `letter-a` is, not surprising, the single mark `a`.

The section 9.4 of the Report describes the recommended representation for all the symbols of the language. The set of all recommendations constitutes the so-called *reference language*. Algol 68 implementations are strongly encouraged to use representation languages which are similar enough to the reference language, recognizable “without further elucidation”, but this is not strictly required.

A representation language may specify more than one representation for a given symbol. For example, in the reference language the `is not symbol` is represented by `isnt`, `:/=` and a variant of the later where the slash sign is superimposed on the equal sign. In this case, an implementation can choose to implement any number of the representations.

Spaces, tabs and newlines are *typographical display features* that, when they appear between symbols, are of no significance and do not alter the meaning of the program. However, when a space or a tab appear in string or character denotations, they represent the `space symbol` and the `tab symbol` respectively¹. The different stopping regimes, however,

¹ The `tab symbol` is a GNU extension

may impose specific restrictions on where typographical display features may or may not appear. See Section 4.4 [Stropping regimes], page 18.

4.2 Worthy characters

The syntactic marks of a representation language, both symbols and typographical display features, are realized as a set of *worthy characters* and the newline. Effectively, an Algol 68 program is a sequence of *worthy characters* and newlines. The worthy characters are:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
space tab " # $ % & ' ( ) * + , - . / : ; < = > @ [ \ ]
^ _ | ! ? ~ { }
```

Some of the characters above were considered unworthy by the original Standard Hardware Representation:

- ! It was considered unworthy because many installations didn't have a vertical bar base character, and ! was used as a base character for |. Today every computer system features a vertical bar character, so ! can qualify as a worthy character.
- & The Revised Report specifies that & is a monad, used as a symbol for the dyadic **and** operator. The Standard Hardware representation decided to turn it into an unworthy character, motivated by the fact that no nomads existed for the other logical operators **not** and **or**, and also with the goal of maintaining the set of worthy characters as small as possible to improve portability. Recognizing that the first motivation still holds, but not the second, this compiler re-instates & as a monad but doesn't use it as an alternative representation of the **and** operator.
- ~ The Standard Hardware Representation vaguely cites some "severe difficulties" with the hardware representation of the tilde character. Whatever these difficulties were at the time, they surely don't exist anymore. This compiler therefore recognizes ~ as a worthy character, and is used as a monad.
- ? The question mark character was omitted as a worthy character to limit the size of the worthy set. This compiler recognizes ? as a worthy character, and is used as a monad.
- \ Back-slash wasn't included as a worthy character because back in 1975 it wasn't supported in EBCDIC (it is now). This compiler recognizes \ as a worthy character.
- tab This compiler recognizes the tabulator character as a worthy character, and it is used as a typographical display feature.

4.3 Base characters

The worthy characters described in the previous section are to be interpreted symbolically rather than visually. The worthy character |, for example, is the vertical line character and generally looks the same in every system. The worthy character **space** is obviously referred by a symbolic name.

The actual visually distinguishable characters available in an installation are known as *base characters*. The Standard Hardware Representation allows implementations the possibility of using two or more base characters to represent a single worthy character. This was the case of the | character, which was represented in many implementations by either | or !.

This compiler uses the set of base characters corresponding to the subset of the Unicode character set that maps one to one to the set of worthy characters described in the previous section:

A-Z	65-90
a-z	97-122
space	32
tab	9
!	33
"	34
#	35
\$	36
%	37
&	38
'	39
(40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
:	58
;	59
<	60
=	61
>	62
?	63
@	64
[91
\	92
]	93
^	94
_	95
	124
~	126

4.4 Stopping regimes

The Algol 68 reference language establishes that certain source constructs, namely mode indications and operator indications, consist in a sequence of *bold letters* and *bold digits*,

Below is the `recsel output records` procedure again, this time encoded in RES stropping.

```
PROC RECSEL OUTPUT RECORDS = VOID:
BEGIN BITS FLAGS
    := (INCLUDE DESCRIPTORS | REC F DESCRIPTOR | REC F NONE);
    .RECRSET RES = REC DB QUERY (DB, RECUTL TYPE,
                                RECUTL QUICK, FLAGS);
    .RECWRITER WRITER := REC WRITER FILE NEW (STDOUT);

    SKIP COMMENTS OF WRITER := TRUE;
    IF RECUTL PRINT SEXPS
    THEN MODE .OF WRITER := REC WRITER SEXP FI;
    REC WRITE (WRITER, RES)
END
```

Note how user-defined mode and operator indications still require explicit stropping.

4.4.3 UPPER stropping

In time computers added support for more than one alphabet by introducing character sets with both upper and lower case letters, along with convenient ways to both input and display these.

In UPPER stropping the bold letters in bold word are represented by upper-case letters, whereas the letters in tags are represented by lower-case letters.

The notions of upper- and lower-case are not applicable to digits, but since the language syntax assures that it is not possible to have a bold word that starts with a digit, digits are considered to be bold if they follow a bold letter or another bold digit.

Below is the `recsel output records` procedure again, this time encoded in UPPER stropping.

```
PROC recsel output records = VOID:
BEGIN BITS flags
    := (include descriptors | rec f descriptor | rec f none);
    RECRSET res = rec db query (db, recutl type,
                                recutl quick, flags);
    RECWRITER writer := rec writer file new (stdout);

    skip comments of writer := TRUE;
    IF recutl print sexps
    THEN mode OF writer := rec writer sexp FI;
    rec write (writer, res)
END
```

Note how in this regime it is almost never necessary to introduce bold tags with points. All in all, it looks much more natural to contemporary readers. UPPER stropping is in fact the stropping regime of choice today. It is difficult to think of any reason why anyone would resort to use POINT or RES stropping.

- The set of nomads is `></=*`.

4.6 String breaks

The intrinsic value of each worthy character that appears inside a string denotation is itself. The string `"/abc"`, therefore, contains a slash character followed by the three letters `a`, `b` and `c`.

Sometimes, however, it becomes necessary to represent some non-worthy character in a string denotation. In these cases, an escape convention has to be used to represent these extra string-items. It is up to the implementation to decide this convention, and the only requirement imposed by the Standard Hardware Representation on this regard is that the character used to introduce escapes, the *escape character*, shall be the apostrophe. This section documents the escape conventions implemented by the GNU compiler.

Two characters have special meaning inside string denotations: double quote (`"`) and apostrophe (`'`). The first finishes the string denotation, and the second starts a *string break*, which is the Algol 68 term for what is known as an “escape sequence” in other programming languages. Two consecutive double-quote characters specify a single double-quote character.

The following string breaks are recognized by this compiler:

<code>' '</code>	Apostrophe character <code>'</code> .
<code>'n'</code>	Newline character.
<code>'f'</code>	Form feed character.
<code>'r'</code>	Carriage return (no line feed).
<code>'t'</code>	Tab.
<code>'(list of character codes separated by commas)'</code>	The indicated characters, where each code has the form <code>uhhhh</code> or <code>Uhhhhhhh</code> , where <code>hhhh</code> and <code>hhhhhhh</code> are integers expressing the character code in hexadecimal. The list must contain at least one entry.

A string break can appear as the single string-item in a character denotation, subject to the following restrictions:

- List of characters string breaks `'(...)'` that contain more than one character code are not allowed in character denotations. If the specified code point is not a valid Unicode character then a compilation error shall be raised.

- -
- 7
- *
 - /
 - over, %
 - mod, %*
 - elem
- 8
- **
 - shl, up
 - shr, down
 - up, down
 - ^
 - lwb
 - upb
- 9
- i
 - +*

5.4 Rows operators

The following operators work on any row mode, denoted below using the pseudo-mode **rows**.

lwb = (rows a) int [Operator]
 Monadic operator that yields the lower bound of the first bound pair of the descriptor of the value of **a**.

upb = (rows a) int [Operator]
 Monadic operator that yields the upper bound of the first bound pair of the descriptor of the value of **a**.

lwb = (int n, rows a) int [Operator]
 Dyadic operator that yields the lower bound in the n-th bound pair of the descriptor of the value of **a**, if that bound pair exists. Attempting to access a non-existing bound pair results in a run-time error.

upb = (int n, rows a) int [Operator]
 Dyadic operator that yields the upper bound in the n-th bound pair of the descriptor of the value of **a**, if that bound pair exists. Attempting to access a non-existing bound pair results in a run-time error.

tan = (1 real a) 1 real [Procedure]

Procedure that yields the tan trigonometric function of the given real argument.

arctan = (1 real a) 1 real [Procedure]

Procedure that yields the arc-tan trigonometric function of the given real argument.

6.4 Extended boolean operators

xor = (bool a, b) bool [Operator]

Dyadic operator that yields the exclusive-or operation of the given boolean arguments.

6.5 Extended bits operators

xor = (l bits a, b) l bits [Operator]

Dyadic operator that yields the bit exclusive-or operation of the given bits arguments.

6.6 Extended math procedures

6.6.1 Logarithms

log = (l real a, b) l real [Procedure]

Procedure that calculates the base ten logarithm of the given arguments.

fgets = (int fd, int n) ref string

[Procedure]

Read a string from the file with descriptor **fd** and yield a reference to it. If **n** is bigger than zero then characters get read until either **n** characters have been read or the end of line is reached. If **n** is zero or negative then characters get read until either a new line character is read or the end of line is reached.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

