The GNU Algol 68 Compiler

For GCC version 17.0.0 (pre-release)

(GCC)

Jose E. Marchesi

Short Contents

Table of Contents

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 5.4 [Stropping regimes], page 22.

`-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:

`-I`*`dir`*
: Add the directory *dir* to the list of directories to be searched for files when processing the Section 4.2.1 [pragmat include], page 19. 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.

`-L`*`dir`*
: 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, lib*basename*.so, lib*basename*.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 2.4 [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 later 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.

`trilean.so`
: Is a DSO, or shared object, expected to contain a `.a68_exports` section with exports data for one or more modules.

`trilean.a`
: Is an archive, whose constituent object files may contain `.a68_exports` sections with exports data for one or more modules.

`trilean.o`
: Is an object file expected to contain export data for one or more modules in a `.a68_exports` section.

The files are tried in order, and if export data for the requested module **Trilean** is found, it is read in, decoded, and used to compile `main.a68` into `main.o`.

The last step in obtaining an executable for our program would be to use GCC to do a link like `ga68 trilean.o main.o -o main`.

Module indicants such as **Trilean** are bold words in the language. This means that, independently of the stropping regime used, they are constituted by a bold letter followed by a sequence of zero or more bold letters and digits. Using the modern stropping supported by the GNU Algol 68 compiler, this means that all of **Trilean**, **TRILEAN** and **Tri_lean** denote exactly the same module indicant, **trilean**.

The mapping from module indicant to the "base name" used to locate the module exports is quite straightforward: the bold letters are transformed to lower-case letters, and the bold digits are just normal digits. Therefore, the exports for the module **GRAMP_Grammar** would looked in files `grampgrammar.m68`, `libgrampgrammar.so`, etc.

But often this default, straightforward mapping, is not what we need.

Suppose for example that a shared object installed in the system, `liba68goodies.so`, provides many facilities in the form of several modules, including a **Trilean** module. We want to use the trilean implementation of liba68goodies in our program `main.a68`. If we just **access Trilean** GCC will look for `trilean.m68` etc, but wont even consider looking in `liba68goodies.so`. Accessing **A68Goodies** is obviously not a solution, as the module we want is **Trilean** and there may not even be a module called **A68Goodies** in `liba68goodies.so`.

The solution for this is to use the *modules map* of the compiler. This map is an association or map between module indicants and base-names. When it comes to access some module, the compiler looks in the map. If there is an entry for the module's indicant, then it fetches the base-name to use for looking for the module's export data. If there is not an entry for the module's indicant then the default, straightforward mapping described above is attempted.

By default the map is empty, but we can add entries by using the `-fmodules-map=` and `-fmodules-map-file=` command-line options. The first option expects entries to be added to the map in a string in the command-line, whereas the second option expects the name of a file containing the entries to add to the map. In both cases the format describing the entries is exactly the same (see Section 1.3 [Module search options], page 1).

In our case, we could compile our main program specifying an entry in the map telling the compiler where to find the trilean and logging modules:

```
$ ga68 -fmodules-map="a68goodies=Trilean,Logger" -c main.a68
```

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 Holes

Holes are part of the modules system and are a mechanism used for two main purposes:

- Top-down programming.
- Communication with other programming languages.

At the moment only the second kind of holes are supported by this compiler. See Chapter 3 [Foreign Function Interface], page 15.

2.4 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
```

2.4.1 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 united 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
```

2.4.2 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
```

2.5 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 Foreign Function Interface

It is possible to call functions written in other programming languages, and also to access variables and constants, using the following form of *formal hole*:

nest language_indicant `"row char denotation"`

Where **language_indicant** is a bold word indicating the programming language we are communicating with and the row char denotation specifies the entity we are accessing. The interpretation of the later depends on the specific language.

The formal hole construction is an *unit*, and can only appear in a strong context. It's mode is the mode expected by the strong context. For example, in the following declaration:

```
int counter = nest C "_counter";
```

The mode of the formal hole is **int**, since it is in a strong context (the actual parameter of an identity declaration) in which an integral value is expected. Likewise, in:

```
proc long int func = nest C "random";
```

The expected mode is a procedure that gets no arguments and returns a **long int**.

The set of modes that are accepted in the formal holes depend on the specific language we are communicating to. These are usually restricted because the compiler may not know how to translate certain Algol 68 concepts into the foreign language ones. A compile time error is issued if an invalid mode is required. Specifics are described in the sections below.

It is important to note that the language-indicants like **C** or **Fortran** are still available to be used as mode indicants, operators or module indicants: they only qualify as language indicants when they appear in a formal hole (**nest**) construct.

The following subsections document the specific supported foreign languages.

3.1 Communicating with C

The language indicant **C** is used to access C variables and calling C functions:

nest C `"[&]symbol"`

The row char denotation should contain a symbol corresponding to a C variable or function, optionally preceded by an ampersand `&`. It is the responsibility of the programmer to make sure the specified symbol actually corresponds to an entity with the right type. In case a malformed symbol is specified it will very likely result in an assembler error.

For example, this is how we can access a C variable as an Algol 68 constant:

```
int counter = nest C "counter";

if counter = 0
then ... fi;
```

If we wanted to be able to change the value of the C variable `counter` then we need to add a leading ampersand to the row denotation, and of course change the mode on the Algol 68 side to a **ref int**:

```
ref int counter = nest C "&counter";

counter +:= 1;
```

If we wanted to access a C pointer variable `int *ptr` as an Algol 68 name we could do:

```
ref int ptr = nest C "ptr";

ptr := 100; { Changes the value pointed by the C pointer ptr,
              not of ptr itself }
```

If we wanted to call the standard POSIX functions `random` and `srandom` we could do:

```
proc long int random = nest C "random";
proc(bits)void srandom = nest C "srandom";

if random < 100 then ... fi
```

The modes accepted in a formal hole for C are:

void As C `void`.

bool As C `bool`.

char As C 32-bit integer.

int As C `int`.

short int As C `short`.

short short int
As C `char`.

long int As C `long` or as C `int`.

long long int
As C `long long` or as C `long` or as C `int`.

bits As C `unsigned int`.

short bits As C `unsigned short`.

short short bits
As C `unsigned char`.

long bits As C `unsigned long` or as C `unsigned int`.

long long bits
As C `unsigned long long` or as C `unsigned long` or as C `unsigned int`.

real As C `float`

long real As C `double`

`string` but only as formal parameters of procedures
Each Algol 68 string formal parameter turns into three parameters in C:

`uint32_t *s`
A pointer to the UCS-4 characters composing the string.

`size_t len`
The length of `s` in number of characters.

`size_t stride`
The distance in bytes between each character in `s`.

proc with accepted formal parameter modes and yielded mode
: As the corresponding C functions.

Structs with fields of accepted modes
: As the corresponding C structs.

4 Comments and pragmats

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

4.1 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 **note** ... **eton**. 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.

4.2 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.

4.2.1 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`.

5 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.

5.1 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` respectively[1]. The different stropping 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 5.4 [Stropping regimes], page 22.

5.2 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.

5.3 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
```

5.4 Stropping 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*, known as a *bold word*. In contrast, other constructs like identifiers, field selectors and labels, collectively known as *tags*, are composed of regular, non-bold letters and digits.

What is precisely a bold letter or digit, and how they differ from non-bold letters and digits, is not specified by the Report. This is no negligence, but a conscious effort at abstracting the definition of the so-called *strict language* from its representation. This allows different representations of the same language.

Some representations of Algol 68 are intended to be published in books, be it paper or electronic devices, and be consumed by persons. These are called *publication languages*. In publication languages bold letters and digits are typically represented by actual bold alphanumeric typographic marks. An Algol 68 program hand written on a napkin or a sheet of paper would typically represent bold letters and digits underlined, or stroked using a different color ink.

Other representations of Algol 68 are intended to be automatically processed by a computer. These representations are called *hardware languages*. Sometimes the hardware languages are also intended to be written and read by programmers; these are called *programming languages*.

Unfortunately, computer systems today usually do not yet provide readily usable and ergonomic bold or underline alphanumeric marks, despite the existence of Unicode and modern and sophisticated editing environments. The lack of appropriate input methods surely plays a role to explain this. Thus, the programming representation languages of Algol 68 should resort to a technique known as *stropping* in order to differentiate bold letters and digits from non-bold letters and digits. The set of rules specifying the representation of these characters is called a *stropping regime*.

There are three classical stropping regimes for Algol 68, which are standardized and specified in the Standard Hardware Representation normative document. These are *POINT stropping*, *RES stropping* and *UPPER stropping*. The following sections do a cursory tour over them; for more details the reader is referred to the Standard Hardware Representation.

This compiler implements UPPER stropping and SUPPER stropping.

5.4.1 POINT stropping

POINT stropping is in a way the most fundamental of the three standard regimes. It was designed to work in installations with limited character sets that provide only one alphabet, one set of digits, and a very restricted set of other symbols.

In POINT stropping a bold word is represented by its constituent letters and digits preceded by a point character. For example, the symbol `bold begin symbol` in the strict language, which is represented as **begin** in bold face in the reference language, would be represented as `.BEGIN` in POINT stropping.

More examples are summarized in the following table.

Strict language	Reference language	POINT stropping
`true` symbol	**true**	`.TRUE`
`false` symbol	**false**	`.FALSE`
`integral` symbol	**int**	`.INT`
`completion` symbol	**exit**	`.EXIT`
`bold-letter-c-...`	**crc32**	`.CRC32`

In POINT stropping a tag is represented by writing its constituent non-bold letters and digits in order. But they are organized in several *taggles*.

Each taggle is a sequence of one or more letters and digits, optionally followed by an underscore character. For example, the tag `PRINT` is composed of a single taggle, but the tag `PRINT_TABLE` is composed of a first taggle `PRINT_` followed by a second taggle `TABLE`.

To improve readability it is possible to insert zero or more white space characters between the taggles in a tag. Therefore, the tag `PRINT_TABLE` could have been written `PRINT TABLE`, or even `PRINT_ TABLE`. This is the reason why Algol 68 identifiers, labels and field selectors can and do usually feature white spaces in them.

It is important to note that both the trailing underscore characters in taggles and the white spaces in a tag do not contribute anything to the denoted tag: these are just stropping artifacts aimed to improve readability. Therefore `FOOBAR` `FOO BAR`, `FOO_BAR` and `FOO_BAR_` are all representations of the same tag, that represents the `letter-f-letter-o-letter-o-letter-b-letter-a-letter-r` language construct.

Below is the text of an example Algol 68 procedure encoded in POINT 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
```

5.4.2 RES stropping

The early installations where Algol 68 ran not only featured a very restricted character set, but also suffered from limited storage and complex to use and time consuming input methods such as card punchers and readers. It was important for the representation of programs to be as compact as possible.

It is likely that is what motivated the introduction of the RES stropping regime. As its name implies, it removes the need of stropping many bold words by introducing *reserved words*.

A *reserved word* is one of the bold words specified in the section 9.4.1 of the Report as a representation of some symbol. Examples are **at**, **begin**, **if**, **int** and **real**.

RES stropping encodes bold words and tags like POINT stropping, but if a bold word is a reserved word then it can then be written without a preceding point, achieving this way a more compact, and easier to read, representation for programs.

Introducing reserved words has the obvious disadvantage that some tags cannot be written the obvious way due to the possibility of conflicts. For example, to represent a tag `if` it is not possible to just write `IF`, because it conflicts with a reserved word, but this can be overcome easily (if not elegantly) by writing `IF_` instead.

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 an operator indications still require explicit stropping.

5.4.3 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.

5.4.4 SUPPER stropping

In the SUPPER stropping regime bold words are written by writing a sequence of one or more *taggles*. Each taggle is written by writing a letter followed by zero or more other letters and digits and is optionally followed by a trailing underscore character. The first letter in a bold word shall be an upper-case letter. The rest of the letters in the bold word may be either upper- or lower-case.

For example, `RecRset`, `Rec_Rset` and `RECRset` are all different ways to represent the same mode indication. This allows to recreate popular naming conventions such as `CamelCase`.

As in the other stropping regimes, the casing of the letters and the underscore characters are not really part of the mode or operator indication.

Operator indications are also bold words and are written in exactly the same way than mode indications, but it is usually better to always use upper-case letters in operator indications. On one side, it looks better, especially in the case of dyadic operators where the asymmetry of, for example `Equal` would look odd, consider `m1 Equal m2` as opposed to `m1 EQUAL m2`. On the other side, tools like editors can make use of this convention in order to highlight operator indications differently than mode indications.

In the SUPPER stropping regime tags are written by writing a sequence of one or more *taggles*. Each taggle is written by writing a letter followed by zero or more other letters and digits and is optionally followed by a trailing underscore character. All letters in a tag shall be lower-case letters.

For example, the identifier `list` is represented by a single taggle, and it is composed by the letters `l`, `i`, `s` and `t`, in order. In the jargon of the strict language we would spell the tag as `letter-l-letter-i-letter-s-letter-t`.

The label `found_zero` is represented by two taggles, `found_` and `zero`, and it is composed by the letters `f`, `o`, `u`, `n`, `d`, `z`, `e`, `r` and `o`, in order. In the jargon of the strict language we would spell the tag as `letter-f-letter-o-letter-u-letter-n -letter-d-letter-z-letter-e-letter-r-letter-o`.

The identifier `crc_32` is likewise represented by two taggles, `crc_` and `32`. Note how the second taggle contains only digits. In the jargon of the strict language we would spell the tag as `letter-c-letter-r-letter-c-digit-three-digit-two`.

The underscore characters are not really part of the tag, but part of the stropping. For example, both `goto found_zero` and `goto foundzero` jump to the same label.

In general, typographical display features are allowed between any symbol in the written program. In SUPPER stropping, however, it is not allowed to place spaces or tab characters between the constituent digits of bits denotations when the radix is 16. This is to avoid confusing situations like the following invalid program:

while `bitmask /= 16r0` **do** ~ **od**

Where the bits denotation would be interpreted as `16r0d` rather than `16r0`, leading to a syntax error. Note however that typographical display features are still allowed between the radix part and the digits, so `16r aabb` is valid also in SUPPER stropping.

The `recsel output records` procedure, encoded in SUPPER stropping, looks like below.

```
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_uick, 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
```

5.5 Monads and Nomads

Algol68 operators, be them predefined or defined by the programmer, can be referred via either bold tags or sequences of certain non-alphabetic symbols. For example, the dyadic operator `+` is defined for many modes to perform addition, the monadic operator **entier** gets a real value and rounds it to an integral value, and the operator `:=:` is the identity relation. Many operators provide both bold tag names and symbols names, like in the case of `:/=:` that can also be written as **isnt**.

Bold tags are lexically well delimited, and if the same tag is used to refer to a monadic operator and to a dyadic operator, no ambiguity can arise. For example, in the following program it is clear that the second instance of **plus** refers to the monadic operator, and the first instance refers to the dyadic operator[2].

```
op PLUS = (int a, b) int: a + b,
   PLUS = (int a): a;
int val = 2 PLUS PLUS 3;
```

On the other hand, symbols are not lexically delimited as words, and one symbol can appear immediately following another symbol. This can lead to ambiguities. For example, if we were to define a C-like monadic operator `++` like:

```
op ++ = (ref int a) int: (int t = a; a +:=1; t);
```

Then the expression `++a` would be ambiguous: is it `++a` or `+(+a)`?. In a similar way, if we would use `++` as the name of a dyadic operator, an expression like `a++b` could be also interpreted as both `a++b` and `a+(+b)`.

To avoid these problems Algol 68 divides the symbols which are suitable to appear in the name of an operator into two classes: monads and nomads. *Monads* are symbols that can be used as monadic operators. *Nomads* are symbols which can be used as both monadic or dyadic operators. Given these two sets, the rules to conform a valid operator are:

- A bold tag.
- Any monad.
- A monad followed by a nomad.
- A monad optionally followed by a nomad followed by either `:=` or `=:`, but not by both.

In the GNU Algol 68 compiler:

- The set of monads is `%^&+-~!?`.

[2] If one would write **plusplus**, it would be a third different bold tag.

– The set of nomads is `></=*`.

5.6 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:

`''` Apostrophe character `'`.

`'n` Newline character.

`'f` Form feed character.

`'r` Carriage return (no line feed).

`'t` Tab.

`'(list of character codes separated by commas)`
: The indicated characters, where each code has the form `uhhhh` or `Uhhhhhhhh`, where `hhhh` and `hhhhhhhh` 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.

6 Standard prelude

The Algol 68 Revised Report defines an extensive set of standard modes, operators, procedures and values, collectively known as the *standard prelude*.

The standard prelude is available to Algol 68 programs without needing to import any module.

For brevity, in this section the pseudo-mode **L** represents a *shortsety*, i.e. a sequence of either zero or more **LONG** or zero or more **SHORT**.

6.1 Environment enquiries

An *environment enquiry* is a constant or a procedure, whose elaboration yields a value that may be useful to the programmer, that reflects some characteristic of the particular implementation. The values of these enquiries are also determined by the architecture and operating system targeted by the compiler.

int `int lengths` [Constant]
: 1 plus the number of extra lengths of integers which are meaningful.

int `int shorths` [Constant]
: 1 plus the number of extra shorths of integers which are meaningful.

l int `L max int` [Constant]
: The largest integral value.

int `real lengths` [Constant]
: 1 plus the number of extra lengths of real numbers which are meaningful.

int `real shorths` [Constant]
: 1 plus the number of extra shorths of real numbers which are meaningful.

l real `L max real` [Constant]
: The largest real value.

l real `L small real` [Constant]
: The smallest real value such that both `1 + small real > 1` and `1 - small real < 1`.

int `bits lengths` [Constant]
: 1 plus the number of extra widths of bits which are meaningful.

int `bits shorths` [Constant]
: 1 plus the number of extra shorths of bits which are meaningful.

int `bits width` [Constant]
int `long bits width` [Constant]
int `long long bits width` [Constant]
: The number of bits in a **bits** value.

int `bytes lengths` [Constant]
: 1 plus the number of extra widths of bytes which are meaningful.

int `bytes shorths` [Constant]
1 plus the number of extra shorths of bytes which are meaningful.

int `bytes width` [Constant]
int `long bytes width` [Constant]
int `long long bytes width` [Constant]
The number of chars in a **bytes** value.

int `max abs char` [Constant]
The largest value which **abs** of a **char** can yield.

char `null character` [Constant]
Some character.

char `flip` [Constant]
char `flop` [Constant]
Characters used to represent **true** and **false** boolean values in textual transput.

char `error char` [Constant]
Character used to represent the digit of a value resulting from a conversion error in textual transput.

char `blank` [Constant]
The space character.

l real `L pi` [Constant]
The number pi.

6.2 Standard modes

void [Mode]
The only value of this mode is **empty**.

bool [Mode]
Mode for the boolean truth values **true** and **false**.

l int [Mode]
Modes for signed integral values. Each **long** or **short** may increase or decrease the range of the domain, depending on the characteristics of the current target. Further **long**s and **short**s may be specified with no effect.

l real [Mode]
Modes for signed real values. Each **long** may increase the upper range of the domain, depending on the characteristics of the current target. Further **long**s may be specified but with no effect.

char [Mode]
Mode for character values. The character values are mapped one-to-one to code points in the 21-bit space of Unicode.

string = **flex**[*1:0*]**char** [Mode]

Mode for sequences of characters. This is implemented as a flexible row of **char** values.

l compl = **struct** (**real** *re,im*) [Mode]

Modes for complex values. Each **long** may increase the precision of both the real and imaginary parts of the numbers, depending on the characteristics of the current target. Further **long**s may be specified with no effect.

l bits [Mode]

Compact and efficient representation of a row of boolean values. Each **long** may increase the number of booleans that can be packed in a bits, depending on the characteristics of the current target.

l bytes [Mode]

Compact and efficient representation of a row of character values. Each **long** may increase the number of characters that can be packed in a bytes, depending on the characteristics of the current target.

6.3 Standard priorities

1

- `plusab`, `+:=`
- `minusab`, `-:=`
- `timesab`, `*:=`
- `divab`, `/:=`
- `overab`, `%:=`
- `modab`, `%*:=`
- `plusto`, `+=:`

2

- `or`

3

- `and`
- `xor`

4

- **eq**, `=`
- **ne**, `/=`

5

- **lt**, `<`,
- **le**, `<=`
- **gt**, `>`
- **ge**, `>=`

6

- `+`

- -

7

- *
- /
- **over**, %
- **mod**, %*
- **elem**

8

- **
- **shl**, **up**
- **shr**, **down**
- **up**, **down**
- ^
- **lwb**
- **upb**

9

- **i**
- +*

6.4 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.

6.5 Boolean operators

not = (**bool** a) **bool** [Operator]
˜ = (**bool** a) **bool** [Operator]

Monadic operator that yields the logical negation of its operand.

or = (**bool** a, b) **bool** [Operator]

Dyadic operator that yields the logical "or" of its operands.

and = (**bool** a, b) **bool** [Operator]
& = (**bool** a, b) **bool** [Operator]

Dyadic operator that yields the logical "and" of its operands.

eq = (**bool** a, b) **bool** [Operator]
= = (**bool** a, b) **bool** [Operator]

Dyadic operator that yields **true** if its operands are the same truth value, **false** otherwise.

ne = (**bool** a, b) **bool** [Operator]
/= = (**bool** a, b) **bool** [Operator]

Dyadic operator that yields **false** if its operands are the same truth value, **true** otherwise.

abs = (**bool** a) **int** [Operator]

Monadic operator that yields 1 if its operand is **true**, and 0 if its operand is **false**.

6.6 Integral operators

6.6.1 Arithmetic

+ = (**l int** a) **l int** [Operator]

Monadic operator that yields the affirmation of its operand.

- = (**l int** a) **l int** [Operator]

Monadic operator that yields the negative of its operand.

abs = (**l int** a) **l int** [Operator]

Monadic operator that yields the absolute value of its operand.

sign = (**l int** a) **int** [Operator]

Monadic operator that yields -1 if a if negative, 0 if a is zero and 1 if a is positive.

odd = (**l int** a) **bool** [Operator]

Monadic operator that yields **true** if its operand is odd, **false** otherwise.

+ = (**l int** a, b) **l int** [Operator]

Dyadic operator that yields the addition of its operands.

- = (**l int** a, b) **l int** [Operator]

Dyadic operator that yields b subtracted from a.

`*` = (**l int** `a, b`) **l int** [Operator]

Dyadic operator that yields the multiplication of its operands.

over = (**l int** `a, b`) **l int** [Operator]
`%` = (**l int** `a, b`) **l int** [Operator]

Dyadic operator that yields the integer division of `a` by `b`, rounding the quotient toward zero.

mod = (**l int** `a, b`) **l int** [Operator]
`%*` = (**l int** `a, b`) **l int** [Operator]

Dyadic operator that yields the remainder of the division of `a` by `b`.

`/` = (**l int** `a, b`) **l real** [Operator]

Dyadic operator that yields the integer division with real result of `a` by `b`.

`**` = (**l int** `a, b`) **l int** [Operator]
`^` = (**l int** `a, b`) **l int** [Operator]

Dyadic operator that yields `a` raised to the exponent `b`.

6.6.2 Arithmetic combined with assignation

plusab = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]
`+:=` = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]

Plus and become. Dyadic operator that calculates `a + b`, assigns the result of the operation to the name `a` and then yields `a`.

minusab = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]
`-:=` = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]

Dyadic operator that calculates `a - b`, assigns the result of the operation to the name `a` and then yields `a`.

timesab = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]
`*:=` = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]

Dyadic operator that calculates `a * b`, assigns the result of the operation to the name `a` and then yields `a`.

overab = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]
`%:=` = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]

Dyadic operator that calculates `a % b`, assigns the result of the operation to the name `a` and then yields `a`.

modab = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]
`%*:=` = (**ref l int** `a`, **l int** `b`) **ref l int** [Operator]

Dyadic operator that calculates `a %* b`, assigns the result of the operation to the name `a` and then yields `a`.

6.6.3 Relational

eq = (**l int** `a, b`) **bool** [Operator]
`=` = (**l int** `a, b`) **bool** [Operator]

Dyadic operator that yields whether its operands are equal.

ne = (**l int** a, b) **bool** [Operator]
/= = (**l int** a, b) **bool** [Operator]
Dyadic operator that yields whether its operands are not equal.

lt = (**l int** a, b) **bool** [Operator]
< = (**l int** a, b) **bool** [Operator]
Dyadic operator that yields whether a is less than b.

le = (**l int** a, b) **bool** [Operator]
<= = (**l int** a, b) **bool** [Operator]
Dyadic operator that yields whether a is less than, or equal to b.

gt = (**l int** a, b) **bool** [Operator]
> = (**l int** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than b.

ge = (**l int** a, b) **bool** [Operator]
>= = (**l int** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than, or equal to b.

6.6.4 Conversion

shorten = (**short int** a) **short short int** [Operator]
shorten = (**int** a) **short int** [Operator]
shorten = (**long int** a) **int** [Operator]
shorten = (**long long int** a) **long int** [Operator]
Monadic operator that yields, if it exists, the integral value that can be lengthened to the value of a. If the value doesn't exist then the operator yields either the most positive integral value in the destination mode, if a is bigger than that value, or the most negative integral value in the destination mode, if a is smaller than that value.

leng = (**short short int** a) **short int** [Operator]
leng = (**short int** a) **int** [Operator]
leng = (**int** a) **long int** [Operator]
leng = (**long int** a) **long long int** [Operator]
Monadic operator that yields the integral value lengthened from the value of a.

6.7 Real operators

6.7.1 Arithmetic

+ = (**l real** a) **l real** [Operator]
Monadic operator that yields the affirmation of its operand.

- = (**l real** a) **l real** [Operator]
Monadic operator that yields the negative of its operand.

abs = (**l real** a) **l real** [Operator]
Monadic operator that yields the absolute value of its operand.

sign = (**l real** a) **int** [Operator]
Monadic operator that yields -1 if a if negative, 0 if a is zero and 1 if a is positive.

+ = (**l real** a, b) **l real** [Operator]
Dyadic operator that yields the addition of its operands.

- = (**l real** a, b) **l real** [Operator]
Dyadic operator that yields b subtracted from a.

* = (**l real** a, b) **l real** [Operator]
Dyadic operator that yields the multiplication of its operands.

/ = (**l real** a, b) **l real** [Operator]
Dyadic operator that yields the realeger division with real result of a by b.

** = (**l real** a, b) **l real** [Operator]
^ = (**l real** a, b) **l real** [Operator]
Dyadic operator that yields a raised to the real exponent b.

** = (**l real** a, **int** b) **l real** [Operator]
^ = (**l real** a, **int** b) **l real** [Operator]
Dyadic operator that yields a raised to the integral exponent b.

6.7.2 Arithmetic combined with assignation

plusab = (**ref l real** a, **l real** b) **ref l real** [Operator]
+:= = (**ref l real** a, **l real** b) **ref l real** [Operator]
Plus and become. Dyadic operator that calculates a + b, assigns the result of the operation to the name a and then yields a.

minusab = (**ref l real** a, **l real** b) **ref l real** [Operator]
-:= = (**ref l real** a, **l real** b) **ref l real** [Operator]
Dyadic operator that calculates a - b, assigns the result of the operation to the name a and then yields a.

timesab = (**ref l real** a, **l real** b) **ref l real** [Operator]
*:= = (**ref l real** a, **l real** b) **ref l real** [Operator]
Dyadic operator that calculates a * b, assigns the result of the operation to the name a and then yields a.

divab = (**ref l real** a, **l real** b) **ref l real** [Operator]
/:= = (**ref l real** a, **l real** b) **ref l real** [Operator]
Dyadic operator that calculates a / b, assigns the result of the operation to the name a and then yields a.

6.7.3 Relational

eq = (**l real** a, b) **bool** [Operator]
= = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether its operands are equal.

ne = (**l real** a, b) **bool** [Operator]
/= = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether its operands are not equal.

lt = (**l real** a, b) **bool** [Operator]
< = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is less than b.

le = (**l real** a, b) **bool** [Operator]
<= = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is less than, or equal to b.

gt = (**l real** a, b) **bool** [Operator]
> = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than b.

ge = (**l real** a, b) **bool** [Operator]
>= = (**l real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than, or equal to b.

6.7.4 Conversions

round = (**l real** a) **int** [Operator]
Monadic operator that yields the nearest integer to its operand.

entier = (**l real** a) **l int** [Operator]
Monadic operator that yields the integer equal to a, or the next integer below (more negative than) a.

Selecting the gnu68 dialect provides the **floor** operator as a synonym for **entier**. See Section 7.8 [Extended real operators], page 48.

shorten = (**long real** a) **real** [Operator]
shorten = (**long long real** a) **long real** [Operator]
Monadic operator that yields, if it exists, the real value that can be lengthened to the value of a. If the value doesn't exist then the operator yields either the most positive real value in the destination mode, if a is bigger than that value, or the most negative real value in the destination mode, if a is smaller than that value.

leng = (**real** a) **long real** [Operator]
leng = (**long real** a) **long long real** [Operator]
Monadic operator that yields the real value lengthened from the value of a.

6.8 Character operators

6.8.1 Relational

eq = (**char** a, b) **bool** [Operator]
= = (**char** a, b) **bool** [Operator]
Dyadic operator that yields whether its operands are equal.

ne = (**char** `a, b`) **bool** [Operator]
/= = (**char** `a, b`) **bool** [Operator]
Dyadic operator that yields whether its operands are not equal.

lt = (**char** `a, b`) **bool** [Operator]
< = (**char** `a, b`) **bool** [Operator]
Dyadic operator that yields whether `a` is less than `b`.

le = (**char** `a, b`) **bool** [Operator]
<= = (**char** `a, b`) **bool** [Operator]
Dyadic operator that yields whether `a` is less than, or equal to `b`.

gt = (**char** `a, b`) **bool** [Operator]
> = (**char** `a, b`) **bool** [Operator]
Dyadic operator that yields whether `a` is greater than `b`.

ge = (**char** `a, b`) **bool** [Operator]
>= = (**char** `a, b`) **bool** [Operator]
Dyadic operator that yields whether `a` is greater than, or equal to `b`.

6.8.2 Conversions

ABS = (**char** `a`) **int** [Operator]
Monadic operator that yields an unique integer for each permissible value of **char**.

REPR = (**int** `a`) **char** [Operator]
The opposite of **abs** of a character.

6.9 String operators

6.9.1 Relational

eq = (**string** `a, b`) **bool** [Operator]
= = (**string** `a, b`) **bool** [Operator]
Dyadic operator that yields whether its operands are equal. Two strings are equal if they contain the same sequence of characters.

ne = (**string** `a, b`) **bool** [Operator]
/= = (**string** `a, b`) **bool** [Operator]
Dyadic operator that yields whether its operands are not equal.

lt = (**string** `a, b`) **bool** [Operator]
< = (**string** `a, b`) **bool** [Operator]
Dyadic operator that yields whether the string `a` is less than the string `b`.

le = (**string** `a, b`) **bool** [Operator]
<= = (**string** `a, b`) **bool** [Operator]
Dyadic operator that yields whether the string `a` is less than, or equal to string `b`.

gt = (**string** `a`, `b`) **bool** [Operator]
`>` = (**string** `a`, `b`) **bool** [Operator]
Dyadic operator that yields whether the string `a` is greater than the string `b`.

ge = (**string** `a`, `b`) **bool** [Operator]
`>=` = (**string** `a`, `b`) **bool** [Operator]
Dyadic operator that yields whether the string `a` is greater than, or equal to the string `b`.

6.9.2 Composition

`+` = (**string** `a`, `b`) **string** [Operator]
Dyadic operator that yields the concatenation of the two given strings as a new string.

`+` = (**string** `a`, **char** `b`) **string** [Operator]
Dyadic operator that yields the concatenation of the given string `a` and a string whose contents are the character `b`.

`*` (= (**int** `a`, **string** `b`) **string**) [Operator]
`*` (= (**string** `b`, **int** `a`) **string**) [Operator]
Dyadic operator that yields the string composed of `a` concatenated copies of `b`. If `a` is less than zero then it is interpreted to be zero.

6.9.3 Composition combined with assignation

plusab = (**ref string** `a`, **string** `b`) **ref string** [Operator]
`+:=` = (**ref string** `a`, **string** `b`) **ref string** [Operator]
Plus and become. Dyadic operator that calculates `a + b`, assigns the result of the operation to the name `a` and then yields `a`.

plusto = (**string** `b`, **ref string** `a`) **ref string** [Operator]
`+=:` = (**string** `b`, **ref string** `b`) **ref string** [Operator]
Dyadic operator that calculates `a + b`, assigns the result of the operation to the name `a` and then yields `a`.

timesab = (**ref string** `a`, **string** `b`) **ref string** [Operator]
`*:=` = (**ref string** `a`, **string** `b`) **ref stringl** [Operator]
Plus and become. Dyadic operator that calculates `a * b`, assigns the result of the operation to the name `a` and then yields `a`.

6.10 Complex operators

6.11 Bits operators

6.11.1 Logical

NOT = (**l bits** `a`, `b`) **l bits** [Operator]
`~` = (**l bits** `a`, `b`) **l bits** [Operator]
Monadic operator that yields the element-wise not logical operation in the elements of the given bits operand.

AND = (**l bits** `a, b`) **l bits** [Operator]
& = (**l bits** `a, b`) **l bits** [Operator]

Dyadic operator that yields the element-wise and logical operation in the elements of the given bits operands.

OR = (**l bits** `a, b`) **l bits** [Operator]

Dyadic operator that yields the element-wise "or" logical operation in the elements of the given bits operands.

6.11.2 Shifting

SHL = (**l bits** `a`, **int** `n`) **l bits** [Operator]
UP = (**l bits** `a`, **int** `n`) **l bits** [Operator]

Dyadic operator that yields the given bits operand shifted `ABS n` positions to the left if `n >= 0` or `ABS n` positions to the right if `n < 0`. Extra elements introduced on the right or left are initialized to **false**. If `ABS n > L_bits_width` then the resulting bits value has all bits set to `false`.

SHR = (**l bits** `a`, **int** `n`) **l bits** [Operator]
DOWN = (**l bits** `a`, **int** `n`) **l bits** [Operator]

Dyadic operator that yields the given bits operand shifted `ABS n` positions to the right if `n >= 0` or `ABS n` positions to the left if `n < 0`. Extra elements introduced on the right or left are initialized to **false**. If `ABS n > L_bits_width` then the resulting bits value has all bits set to `false`.

6.11.3 Relational

eq = (**l bits** `a, b`) **bool** [Operator]
= = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether its operands are equal. Two bits are equal if they contain the same sequence of booleans.

ne = (**l bits** `a, b`) **bool** [Operator]
/= = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether its operands are not equal.

lt = (**l bits** `a, b`) **bool** [Operator]
< = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is less than the bits `b`.

le = (**l bits** `a, b`) **bool** [Operator]
<= = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is less than, or equal to bits `b`.

gt = (**l bits** `a, b`) **bool** [Operator]
> = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is greater than the bits `b`.

ge = (**l bits** `a, b`) **bool** [Operator]
>= = (**l bits** `a, b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is greater than, or equal to the bits `b`.

6.11.4 Conversions

abs = (**l bits** a) **l int** [Operator]

Monadic operator that yields the integral value whose constituent bits correspond to the booleans stored in a. See Section 9.1 [**bin** and **abs** of negative integral values], page 56.

bin = (**l int** a) **l bits** [Operator]

Monadic operator that yields the bits value whose boolean elements map the bits in the given integral operand. See Section 9.1 [**bin** and **abs** of negative integral values], page 56.

shorten = (**long bits** a) **bits** [Operator]

shorten = (**long long bits** a) **long bits** [Operator]

Monadic operator that yields the bits value that can be lengthened to the value of a.

leng = (**bits** a) **long bits** [Operator]

leng = (**long bits** a) **long long bits** [Operator]

Monadic operator that yields the bits value lengthened from the value of a. The lengthened value features **false** in the extra left positions added to match the lengthened size.

6.12 Bytes operators

6.13 Semaphore operators

6.14 Math procedures

6.14.1 Arithmetic

sqrt = (**l real** a) **l real** [Procedure]

Procedure that yields the square root of the given real argument.

6.14.2 Logarithms

ln = (**l real** a) **l real** [Procedure]

Procedure that yields the base e logarithm of the given real argument.

exp = (**l real** a) **l real** [Procedure]

Procedure that yields the exponential function of the given real argument. This is the inverse of **ln**.

6.14.3 Trigonometric

sin = (**l real** a) **l real** [Procedure]

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

arcsin = (**l real** a) **l real** [Procedure]

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

cos = (**l real** a) **l real** [Procedure]

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

arccos = (**l real** a) **l real** [Procedure]

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

tan = (**l real** a) **l real** [Procedure]

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

arctan = (**l real** a) **l real** [Procedure]

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

7 Extended prelude

This chapter documents the GNU extensions to the standard prelude. The facilities documented below are available to Algol 68 programs only if the `gnu68` language dialect is selected, which is the default.

The extended prelude is available to Algol 68 programs without needing to import any module, provided they are compiled as `gnu68` code, which is the default.

7.1 Extended priorities

`3`

- **xor**

`8`

- **elems**

7.2 Extended environment enquiries

An *environment enquiry* is a constant, whose value may be useful to the programmer, that reflects some characteristic of the particular implementation. The values of these enquiries are also determined by the architecture and operating system targeted by the compiler.

l int `L min int` [Constant]
: The most negative integral value.

int `word bits width` [Constant]
: The number of bits in a **word bits** value.

int `word bytes width` [Constant]
: The number of chars in a **word bytes** value.

l real `L min real` [Constant]
: The most negative real value.

word real `word max real` [Constant]
: The largest word real value.

word real `word min real` [Constant]
: The most negative word real value.

l real `L infinity` [Constant]
: Positive infinity expressed in a real value.

l real `L minus infinity` [Constant]
: Negative infinity expressed in a real value.

word real `word pi` [Constant]
: The number pi as a word real.

char `replacement char` [Constant]
: A character that is unknown, unrecognizable or unrepresentable in Unicode.

char `eof char` [Constant]
: **char** value that doesn't denote an actual char, but an end-of-file situation.

7.3 Extended standard modes

word int [Mode]

Modes for signed integral values of the same width than a word (address) in the target machine.

word bits [Mode]

Compact and efficient representation of a row of boolean values, with as many bits as they fit in a word (address) in the target machine.

word bytes [Mode]

Compact and efficient representation of a row of character values, with as many chars as they fit in a word (address) in the target machine.

word real [Mode]

Modes for signed real values of the same with than a word (address) in the target machine.

7.4 Extended rows operators

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

elems = (**rows** `a`) **int** [Operator]

Monadic operator that yields the number of elements implied by the first bound pair of the descriptor of the value of `a`.

elems = (**int** `n`, **rows** `a`) **int** [Operator]

Dyadic operator that yields the number of elements implied by the n-th bound pair of the descriptor of the value of `a`.

7.5 Extended boolean operators

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

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

7.6 Extended integral operators

7.6.1 Arithmetic

`+` = (**word int** `a`) **word int** [Operator]

Monadic operator that yields the affirmation of its operand.

`-` = (**word int** `a`) **word int** [Operator]

Monadic operator that yields the negative of its operand.

abs = (**word int** `a`) **word int** [Operator]

Monadic operator that yields the absolute value of its operand.

sign = (**word int** `a`) **int** [Operator]

Monadic operator that yields -1 if `a` if negative, 0 if `a` is zero and 1 if `a` is positive.

odd = (**word int** a) **bool** [Operator]

Monadic operator that yields **true** if its operand is odd, **false** otherwise.

+ = (**word int** a, b) **word int** [Operator]

Dyadic operator that yields the addition of its operands.

- = (**word int** a, b) **word int** [Operator]

Dyadic operator that yields b subtracted from a.

* = (**word int** a, b) **word int** [Operator]

Dyadic operator that yields the multiplication of its operands.

over = (**word int** a, b) **word int** [Operator]
% = (**word int** a, b) **word int** [Operator]

Dyadic operator that yields the integer division of a by b, rounding the quotient toward zero.

mod = (**word int** a, b) **word int** [Operator]
%* = (**l int** a, b) **l int** [Operator]

Dyadic operator that yields the remainder of the division of a by b.

/ = (**word int** a, b) **word real** [Operator]

Dyadic operator that yields the integer division with real result of a by b.

** = (**word int** a, b) **word int** [Operator]
^ = (**word int** a, b) **word int** [Operator]

Dyadic operator that yields a raised to the exponent b.

7.6.2 Arithmetic combined with assignation

plusab = (**ref word int** a, **word int** b) **ref word int** [Operator]
+:= = (**ref word int** a, **word int** b) **ref word int** [Operator]

Plus and become. Dyadic operator that calculates a + b, assigns the result of the operation to the name a and then yields a.

minusab = (**ref word int** a, **word int** b) **ref word int** [Operator]
-:= = (**ref word int** a, **word int** b) **ref word int** [Operator]

Dyadic operator that calculates a - b, assigns the result of the operation to the name a and then yields a.

timesab = (**ref word int** a, **word int** b) **ref word int** [Operator]
*:= = (**ref word int** a, **word int** b) **ref word int** [Operator]

Dyadic operator that calculates a * b, assigns the result of the operation to the name a and then yields a.

overab = (**ref word int** a, **word int** b) **ref word int** [Operator]
%:= = (**ref word int** a, **word int** b) **ref word int** [Operator]

Dyadic operator that calculates a % b, assigns the result of the operation to the name a and then yields a.

modab = (**ref word int** a, **word int** b) **ref word int** [Operator]
%*:= = (**ref word int** a, **word int** b) **ref word int** [Operator]

Dyadic operator that calculates a %* b, assigns the result of the operation to the name a and then yields a.

7.6.3 Relational

eq = (**word int** a, b) **bool** [Operator]
= = (**word int** a, b) **bool** [Operator]

Dyadic operator that yields whether its operands are equal.

ne = (**word int** a, b) **bool** [Operator]
/= = (**word int** a, b) **bool** [Operator]

Dyadic operator that yields whether its operands are not equal.

lt = (**word int** a, b) **bool** [Operator]
< = (**word int** a, b) **bool** [Operator]

Dyadic operator that yields whether a is less than b.

le = (**word int** a, b) **bool** [Operator]
<= = (**word int** a, b) **bool** [Operator]

Dyadic operator that yields whether a is less than, or equal to b.

gt = (**word int** a, b) **bool** [Operator]
> = (**word int** a, b) **bool** [Operator]

Dyadic operator that yields whether a is greater than b.

ge = (**word int** a, b) **bool** [Operator]
>= = (**l int** a, b) **bool** [Operator]

Dyadic operator that yields whether a is greater than, or equal to b.

7.7 Extended bits operators

7.7.1 Logical

NOT = (**word bits** a, b) **word bits** [Operator]
~ = (**word bits** a, b) **word bits** [Operator]

Monadic operator that yields the element-wise not logical operation in the elements of the given bits operand.

AND = (**word bits** a, b) **word bits** [Operator]
& = (**word bits** a, b) **word bits** [Operator]

Dyadic operator that yields the element-wise and logical operation in the elements of the given bits operands.

OR = (**word bits** a, b) **word bits** [Operator]

Dyadic operator that yields the element-wise “or” logical operation in the elements of the given bits operands.

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

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

7.7.2 Shifting

SHL = (**word bits** `a`, **int** `n`) **word bits** [Operator]
UP = (**word bits** `a`, **int** `n`) **word bits** [Operator]

Dyadic operator that yields the given bits operand shifted `ABS n` positions to the left if `n >= 0` or `ABS n` positions to the right if `n < 0`. Extra elements introduced on the right or left are initialized to **false**. If `ABS n > L_bits_width` then the resulting bits value has all bits set to `false`.

SHR = (**word bits** `a`, **int** `n`) **word bits** [Operator]
DOWN = (**word bits** `a`, **int** `n`) **word bits** [Operator]

Dyadic operator that yields the given bits operand shifted `ABS n` positions to the right if `n >= 0` or `ABS n` positions to the left if `n < 0`. Extra elements introduced on the right or left are initialized to **false**. If `ABS n > L_bits_width` then the resulting bits value has all bits set to `false`.

7.7.3 Relational

eq = (**word bits** `a`, `b`) **bool** [Operator]
= = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether its operands are equal. Two bits are equal if they contain the same sequence of booleans.

ne = (**word bits** `a`, `b`) **bool** [Operator]
/= = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether its operands are not equal.

lt = (**word bits** `a`, `b`) **bool** [Operator]
< = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is less than the bits `b`.

le = (**word bits** `a`, `b`) **bool** [Operator]
<= = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is less than, or equal to bits `b`.

gt = (**word bits** `a`, `b`) **bool** [Operator]
> = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is greater than the bits `b`.

ge = (**word bits** `a`, `b`) **bool** [Operator]
>= = (**word bits** `a`, `b`) **bool** [Operator]

Dyadic operator that yields whether the bits `a` is greater than, or equal to the bits `b`.

7.7.4 Conversions

abs = (**word bits** `a`) **word int** [Operator]

Monadic operator that yields the integral value whose constituent bits correspond to the booleans stored in `a`. See Section 9.1 [**bin** and **abs** of negative integral values], page 56.

bin = (**word int** a) **word bits** [Operator]

Monadic operator that yields the bits value whose boolean elements map the bits in the given integral operand. See Section 9.1 [**bin** and **abs** of negative integral values], page 56.

shorten = (**long bits** a) **bits** [Operator]
shorten = (**long long bits** a) **long bits** [Operator]

Monadic operator that yields the bits value that can be lengthened to the value of a.

leng = (**bits** a) **long bits** [Operator]
leng = (**long bits** a) **long long bits** [Operator]

Monadic operator that yields the bits value lengthened from the value of a. The lengthened value features **false** in the extra left positions added to match the lengthened size.

7.7.5 Manipulating elements

set = (**l bits** b, **int** n) **l bits** [Operator]
set = (**l bits** b, **int** n) **l bits** [Operator]

Dyadic operator that sets the nth least significant bit in **b**, where n is zero based. If n is not in the range [0,L_bits_width) then the operator yields **b**.

clear = (**l bits** b, **int** n) **l bits** [Operator]
clear = (**word bits** b, **int** n) **word bits** [Operator]

Dyadic operator that clears the nth least significant bit in **b**, where n is zero based. If n is not in the range [0,L_bits_width) then the operator yields **b**.

test = (**l bits** b, **int** n) **bool** [Operator]
test = (**word bits** b, **int** n) **bool** [Operator]

Dyadic operator that tests whether the nth least significant bit in **b** is set, where n is zero based. If n is not in the range 0,L_bits_width) then the operator yields **false**.

7.8 Extended real operators

7.8.1 Arithmetic

\+ = (**word real** a) **word real** [Operator]

Monadic operator that yields the affirmation of its operand.

\- = (**word real** a) **word real** [Operator]

Monadic operator that yields the negative of its operand.

abs = (**word real** a) **word real** [Operator]

Monadic operator that yields the absolute value of its operand.

sign = (**word real** a) **int** [Operator]

Monadic operator that yields -1 if a if negative, 0 if a is zero and 1 if a is positive.

\+ = (**word real** a, b) **word real** [Operator]

Dyadic operator that yields the addition of its operands.

- = (**word real** a, b) **word real** [Operator]

Dyadic operator that yields b subtracted from a.

***** = (**word real** a, b) **word real** [Operator]

Dyadic operator that yields the multiplication of its operands.

/ = (**word real** a, b) **word real** [Operator]

Dyadic operator that yields the realeger division with real result of a by b.

****** = (**word real** a, b) **word real** [Operator]
ˆ = (**word real** a, b) **word real** [Operator]

Dyadic operator that yields a raised to the real exponent b.

****** = (**word real** a, **int** b) **word real** [Operator]
ˆ = (**word real** a, **int** b) **word real** [Operator]

Dyadic operator that yields a raised to the integral exponent b.

7.8.2 Arithmetic combined with assignation

plusab = (**ref word real** a, **word real** b) **ref word real** [Operator]
+:= = (**ref word real** a, **word real** b) **ref word real** [Operator]

Plus and become. Dyadic operator that calculates a + b, assigns the result of the operation to the name a and then yields a.

minusab = (**ref word real** a, **word real** b) **ref word real** [Operator]
-:= = (**ref word real** a, **word real** b) **ref word real** [Operator]

Dyadic operator that calculates a - b, assigns the result of the operation to the name a and then yields a.

timesab = (**ref word real** a, **word real** b) **ref word real** [Operator]
***:=** = (**ref word real** a, **word real** b) **ref word real** [Operator]

Dyadic operator that calculates a * b, assigns the result of the operation to the name a and then yields a.

divab = (**ref word real** a, **word real** b) **ref word real** [Operator]
/:= = (**ref word real** a, **word real** b) **ref word real** [Operator]

Dyadic operator that calculates a / b, assigns the result of the operation to the name a and then yields a.

7.8.3 Relational

eq = (**word real** a, b) **bool** [Operator]
= = (**word real** a, b) **bool** [Operator]

Dyadic operator that yields whether its operands are equal.

ne = (**word real** a, b) **bool** [Operator]
/= = (**word real** a, b) **bool** [Operator]

Dyadic operator that yields whether its operands are not equal.

lt = (**word real** a, b) **bool** [Operator]
< = (**word real** a, b) **bool** [Operator]

Dyadic operator that yields whether a is less than b.

le = (**word real** a, b) **bool** [Operator]
<= = (**word real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is less than, or equal to b.

gt = (**word real** a, b) **bool** [Operator]
> = (**word real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than b.

ge = (**word real** a, b) **bool** [Operator]
>= = (**word real** a, b) **bool** [Operator]
Dyadic operator that yields whether a is greater than, or equal to b.

7.8.4 Conversions

round = (**word real** a) **int** [Operator]
Monadic operator that yields the nearest integer to its operand.

entier = (**word real** a) **word int** [Operator]
Monadic operator that yields the integer equal to a, or the next integer below (more negative than) a.

Selecting the gnu68 dialect provides the **floor** operator as a synonym for **entier**. See Section 7.8 [Extended real operators], page 48.

floor = (**word real** a) **word int** [Operator]
Monadic operator that yields the integer equal to a rounded towards negative infinity.

An alias for operator **entier**. See Section 6.7 [Real operators], page 35.

7.9 Extended math procedures

7.9.1 Logarithms

log = (**l real** a, b) **l real** [Procedure]
log = (**word real** a, b) **word real** [Procedure]
Procedure that calculates the base ten logarithm of the given arguments.

8 POSIX prelude

The POSIX prelude provides facilities to perform simple transput (I/O) based on POSIX file descriptors, accessing the file system, command-line arguments, environment variables, etc.

This prelude is available to Algol 68 programs without needing to import any module, provided they are compiled as `gnu68` code, which is the default.

8.1 POSIX process

The Algol 68 program reports an exit status to the operating system once it stops running. The exit status reported by default is zero, which corresponds to success.

`posix exit` = (**int** `status`) [Procedure]

Procedure that sets the exit status to report to the operating system and immediately stops executing the program. The default exit status is 0 which, by convention, is interpreted by POSIX systems as success. A value different to zero is interpreted as an error status.

8.2 POSIX command line

Algol 68 programs can access the command-line arguments passed to them by using the following procedures.

`argc` = **int** [Procedure]

Procedure that yields the number of arguments passed in the command line, including the name of the program.

`argv` = (**int** `n`) **string** [Procedure]

Procedure that yields the `n`th argument passed in the command line. The first argument is always the name used to invoke the program. If `n` is out of range then this procedure returns the empty string.

8.3 POSIX environment

`getenv` = (**string** `varname`) **string** [Procedure]

Procedure that yields the value of the environment variable `varname` as a string. If the specified environmental variable is not defined the this procedure returns an empty string.

8.4 POSIX errors

When a call to a procedure in this prelude results in an error, the called procedure signals the error in some particular way and also sets a global `errno` to a code describing the error. For example, trying to opening a file that doesn't exist will result in `fopen` returning `-1`, which signals an error. The caller can then inspect the global `errno` to see what particular error prevented the operation to be completed: in this case, `errno` will contain the error code corresponding to "file doesn't exist".

`errno` = **int** [Procedure]

This procedure yields the current value of the global `errno`. The yielded value reflects the error status of the last executed POSIX prelude operation.

`strerror` = (**int** `ecode`) **string** [Procedure]

This procedure gets an error code and yields a string containing an explanatory short description of the error. It is typical to pass the output of `errno` to this procedure.

`perror` = (**string** `msg`) **void** [Procedure]

This procedure prints the given string `msg` in the standard error output, followed by a colon character, a space character and finally the string error of the current value of `errno`.

8.5 POSIX files

File descriptors are **int** values that identify open files that can be accessed by the program. The `fopen` procedure allocates file descriptors as it opens files, and the descriptor is used in subsequent transput calls to perform operations on the files.

8.5.1 Standard file descriptors

There are three descriptors, however, which are automatically opened when the program starts executing and automatically closed when the program finishes. These are:

int `stdin` [Constant]

File descriptor associated with the standard input. Its value is `0`.

int `stdout` [Constant]

File descriptor associated with the standard output. Its value is `1`.

int `stderr` [Constant]

File descriptor associated with the standard error. Its value is `2`.

8.5.2 Opening and closing files

`fopen` = (**string** `pathname`, **bits** `flags`) **int** [Procedure]

Open the file specified by `pathname`. The argument `flags` is a combination of `file o` flags as defined below. If the specified file is successfully opened while satisfying the constraints implied by `flags` then this procedure yields a file descriptor that is used in subsequent I/O calls to refer to the open file. Otherwise, this procedure yields -1. The particular error can be inspected by calling the `errno` procedure.

`fclose` = (**int** `fd`) **int** [Procedure]

Close the given file descriptor, which no longer refers to any file. This procedure yields zero on success, and -1 on error. In the later case, the program can look at the particular error by calling the `errno` procedure.

8.5.3 Creating files

`fcreate` = (**string** `pathname`, **bits** `mode`) **int** [Procedure]

Create a file with name `pathname`. The argument `mode` is a **bits** value containing a bit pattern that determines the permissions on the created file. The bit pattern has the form `8rUGO`, where `U` reflects the permissions of the user who owns the file, `G` reflects the permissions of the users pertaining to the file's group, and `O` reflects the permissions of all other users. The permission bits are 1 for execute, 2 for write and 4 for read. If the file is successfully created then this procedure yields a file descriptor that is used in subsequent I/O calls to refer to the newly created file. Otherwise, this procedure yields -1. The particular error can be inspected by calling the `errno` procedure.

8.5.4 Flags for `fopen`

The following flags can be combined using bit-wise operators. Note that in POSIX systems the effective mode of the created file is the mode specified by the programmer masked with the process's *umask*.

bits `file o default` [Constant]
Flag for `fopen` indicating that the file shall be opened with whatever capabilities allowed by its permissions.

bits `file o rdwr` [Constant]
Flag for `fopen` indicating that the file shall be opened for both reading and writing.

bits `file o rdonly` [Constant]
Flag for `fopen` indicating that the file shall be opened for reading only. This flag is not compatible with `file o rdwr` nor with `file o wronly`.

bits `file o wronly` [Constant]
Flag for `fopen` indicating that the file shall be opened for write only. This flag is not compatible with `file o rdwr` nor with `file o rdonly`.

bits `file o trunc` [Constant]
Flag for `fopen` indicating that the opened file shall be truncated upon opening it. The file must allow writing for this flag to take effect. The effect of combining `file o trunc` and `file o rdonly` is undefined and varies among implementations.

8.5.5 Getting file properties

`fsize = (`**int** `fd)` **long long int** [Procedure]
Return the size in bytes of the file characterized by the file descriptor `fd`. If the system entity characterized by the given file descriptor doesn't have a size, if the size of the file cannot be stored in a **long long int**, or if there is any other error condition, this procedure yields -1 and `errno` is set appropriately.

`lseek = (`**int** `fd,` **long long int** `offset,` **int** `whence)` **long long int** [Procedure]
Set the file offset of the file characterized by the file descriptor `fd` depending on the values of `offset` and `whence`. On success, the resulting offset, as measured in bytes from the beginning of the file, is returned. Otherwise, -1 is returned, `errno` is set to indicate the error, and the file offset remains unchanged. The effects of `offset` and `whence` are:

- If `whence` is `seek set`, the file offset is set to `offset` bytes.
- If `whence` is `seek cur`, the file offset is set to its current location plus `offset`.
- If `whence` is `seek end`, the file offset is set to the size of the file plus `offset`.

8.6 POSIX sockets

A program can communicate with other computers, or with other processes running in the same computer, via sockets. The sockets are identified by file descriptors.

`fconnect` = (**string** `host`, **int** `port`) **int** [Procedure]

This procedure creates a stream socket and connects it to the given `host` using port `port`. The established communication is full-duplex, and allows sending and receiving data using transput until it gets closed. On success this procedure yields a file descriptor. On error this procedure yields -1 and `errno` is set appropriately.

8.7 POSIX string transput

The following procedures read or write characters and strings from and to open files. The external encoding of the files is assumed to be UTF-8. Since Algol 68 **char**s are UCS-4, this means that reading or writing a character may involve reading or writing more than one byte, depending on the particular Unicode code points involved.

8.7.1 Output of strings and chars

`putchar` = (**char** `c`) **char** [Procedure]

Write the given character to the standard output. This procedure yields `c` in case the character got successfully written, or `eof char` otherwise.

`puts` = (**string** `str`) **void** [Procedure]

Write the given string to the standard output.

`fputc` = (**int** `fd`, **char** `c`) **int** [Procedure]

Write given character `c` to the file with descriptor `fd`. This procedure yields `c` on success, or `eof char` on error.

`fputs` = (**int** `fd`, **string** `str`) **int** [Procedure]

Write the given string `str` to the file with descriptor `fd`. This procedure yields the number of bytes written on success, or 0 on error.

8.7.2 Input of strings and chars

`getchar` = **char** [Procedure]

Read a character from the standard input. This procedure yields the read character in case the character got successfully read, or `eof char` otherwise.

`gets` = (**int** `n`) **ref string** [Procedure]

Read a string composed of `n` characters from the standard input 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.

`fgetc` = (**int** `fd`) **int** [Procedure]

Read a character from the file with descriptor `fd`. This procedure yields the read character in case a valid Unicode character got successfully read. If an unrecognizable or unknown character is found then this procedure yields `replacement char`. In case of end of file this procedure yields `eof char`.

`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.

9 Language extensions

This chapter documents the GNU extensions implemented by this compiler on top of the Algol 68 programming language. These extensions collectively conform a strict *superlanguage* of Algol 68, and are enabled by default. To disable them the user can select the strict Algol 68 standard by passing the option `-std=algol68` when invoking the compiler.

9.1 bin and abs of negative integral values

The **bin** operator gets an integral value and yields a **bits** value that reflects the internal bits of the integral value. The semantics of this operator, as defined in the Algol 68 standard prelude, are:

```
op bin = (L int a) L bits:
  if a >= L 0
  then L int b := a; L bits;
       for i from L bits width by -1 to 1
       do (L F of c)[i] := odd b; b := b % L 2 od;
       c
  fi;
```

The **abs** operator performs the inverse operation of **bits**. Given a `L` **bits** value, it yields the `L` **int** value whose bits representation is the bits value. The semantics of this operator, as defined in the Algol 68 prelude, are:

```
op abs = (L bits a) L int:
begin L int c := L 0;
      for i to L bits width
      do c := L 2 * c + K abs (L F of a)[i] od;
      c
end
```

Note how the **bin** of a negative integral value is not defined: the implicit else-part of the conditional yields **skip**, which is defined as any bits value in that context. Note also how **abs** doesn't make any provision to check whether the resulting value is positive: it assumes it is so.

The GNU Algol 68 compiler, when working in strict Algol 68 mode (`-std=algol68`), makes **bin** to always yield `L` **bits** `(`**skip**`)` when given a negative value, as mandated by the report. But the skip value is always the bits representation of zero, *i.e.* 2r0. Strict Algol 68 programs, however, must not rely on this.

When GNU extensions are enabled (`-std=gnu68`) the **bin** of a negative value yields the two's complement bit pattern of the value rather than zero. Therefore, **bin** `-` **short short** `2` yields `2r11111110`. And **abs short short** `2r11111110` yields -2.

9.2 Bold taggles

This compiler supports the stropping regimes known as UPPER and SUPPER. In both regimes bold words are written by writing their constituent bold letters and digits, in order. In UPPER regime all the letters of a bold word are to be written using upper-case. In

SUPPER regime, only the first bold letter is required to be written using upper-case, and this only when the bold word is not a reserved word.

When a bold word comprises several natural words, it may be a little difficult to distinguish them at first sight. Consider for example the following code, written fist in UPPER stropping:

```
MODE TREENODE = STRUCT (TREENODEPAYLOAD data, REF TREENODE next),
     TREENODEPAYLOAD = STRUCT (INT code, REAL average, mean);
```

Then written in SUPPER stropping:

```
mode TreeNode = struct (TreeNodePayload data, REF TreeNode next),
     TreeNodePayload = struct (int code, real average, mean);
```

Particularly in UPPER stropping, it may be difficult to distinguish the constituent natural words at first sight.

In order to improve this, this compiler implements a GNU extension called *bold taggles* that allows to use underscore characters (`_`) within mode and operator indications as a visual aid to improve readability. When this extension is enabled, mode indications and operator indications consist in a sequence of the so-called *bold taggles*, which are themselves sequences of one or more bold letters or digits optionally terminated by an underscore character.

With bold taggles enabled the program above could have been written using UPPER stropping as:

```
MODE TREE_NODE = STRUCT (TREE_NODE_PAYLOAD data, REF TREE_NODE next),
     TREE_NODE_PAYLOAD = STRUCT (INT code, REAL average, mean);
```

And using SUPPER stropping as:

```
mode Tree_Node = struct (Tree_Node_Payload data, ref Tree_Node next),
     Tree_Node_Payload = struct (int code, real average, mean);
```

Which is perhaps more readable for most people. Note that the underscore characters are not really part of the mode or operator indication. Both `TREE_NODE` and `TREENODE` denote the same mode indication. Note also that, following the definition, constructs like `Foo__bar` and `_Baz` are not valid indications.

Bold taggles are available when the gnu68 dialect of the language is selected. See Section 1.1 [Dialect options], page 1.

9.3 Brief selection

It was early recognized that a shorter alternative representation the of-symbol was very much needed, considering the fact the bold version **of** is at least four characters long. This makes certain phrases long and also slightly laborious to read, like in:

```
pub op + = (Pos a,b) Pos: (c of a + c of b, r of a + r of b),
       - = (Pos a,b) Pos: (c of a - c of b, r of a - r of b);
```

This compiler allows using a quote character `'` instead of `of` in selections of structs and multiples. Using this brief style the example above now can be written as:

```
pub op + = (Pos a,b) Pos: (c'a + c'b, r'a + r'b),
       - = (Pos a,b) Pos: (c'a - c'b, r'a - r'b);
```

GNU General Public License

Version 3, 29 June 2007

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 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 FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

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-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through 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 compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   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, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See `https://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

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.

Option Index

ga68's command line options are indexed here without any initial '-' or '--'. Where an option has both positive and negative forms (such as -f*option* and -fno-*option*), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

F

I

L

S

W

Index

%

&

*

+

–

/

<

=

>

^

~

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X