The Algol 68 Jargon File

Jose E. Marchesi

Table of Contents

This file provides definitions for many terms used in the context of the Algol 68 programming language and associated technologies. You can find this file in other formats along with the sources at `https://jemarch.net`.

1 Introduction

As C. H. Lindsey puts it in his legendary Informal Introduction to Algol 68, a language in which fundamental concepts combine in an orthogonal way requires very precise terminology. Algol 68 is the orthogonal programming language for antonomasia, and it for sure introduces a rich set of very precise terminology.

Furthermore, when the language got introduced the IFIP WG-2.1 took great care of using new terms for concepts that had their rough similar equivalences in other programming languages, instead of using the most common terms. Such is the case of *assignation*, which is similar but not exactly the same than the *assignment* of other programming languages. Many of the new terms are neologisms created for the occasion, also for good reasons as discussed below.

This all means that the Algol 68 programmers, implementors and aficionados need to get familiar with a very precise and somewhat extensive terminology. That may be quite confusing to the uninitiated.

As with most things related to Algol 68, mastering the terminology requires a little bit of effort and time, but believe me, it pays back in spades. Watching two Algol 68 programmers discussing about their programs is like watching two well greased machines: the terms they use are precise, and they can use terms referring to domain-specific concepts that would require the usage of a (probably not very well constructed on the fly) metaphor or analogy in other programming languages, and very little if anything is lost in translation. The communication is fast, rich and precise. It is also fun.

This jargon file is an attempt to gather and summarize this terminology for the benefit of anyone introducing herself in the enthralling world of algorithmic languages.

How to use this file

Each entry in the file describes the meaning of one particular term, including a more or less extensive description of the entity or concept described by the term. This usually involves programming examples, but note that the purpose of this file is *not* to be an Algol 68 manual. Usage examples *of the term* are shown in the form of hypothetical lines of dialogues. When applicable, the syntax of the concept associated with the term will be also explained as simplified syntactic rules from the Report. Finally, references to other entries or to the bibliography are included in the entries.

So how to look for a term in this file?

If you are reading this document in an *info* reader, then you can press `m` and introduce the term you are looking for. Your info reader shall be nice enough to provide auto-complete. References can then be followed the same way.

If you are reading this document as a man-page, then you will find references to all the entries of the jargon file in the `SEE ALSO` section below.

If you are reading this document as a PDF, then you can use either the table of contents or the concepts index you can found in the appendices. Depending on how nice your PDF reader is, and assuming you are not reading a printed document, you can probably follow the references by clicking on them.

If you are reading this document as an HTML in some website, then you can follow the hyperlinks in table of contents and indexes.

Bibliography

- The Revised Report on the Algorithmic Language Algol 68 By A. van Wijngaarden, B.J. Mailloux, J.E.L Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker.

  Referenced by marks like [RR *section*].

- The Report on the Standard Hardware Representation for ALGOL 68 By Wilfred J. Hansen and Hendrik Boom.

  Referenced by marks like [SHR *section*].
- The Informal Introduction to Algol68 By C.H. Lindsey and V.D. Meulen.

  Referenced by marks like [II *section*]

2 Metalanguage

2.1 Aleph

Meaning

The transput section of the Revised Report consists in a description of the transput facilities in the form of near-Algol 68 code, intended to serve as a precise reference of the intended programming interface and also of the semantics of the several operations like `print` or `read`. The same technique is used to describe the standard prelude.

Consider for example the mode **channel**, which part of the standard transput:

```
mode channel =
   struct (proc(ref book)bool reset, set,
                                get, put, bi, compress, reidf,
           proc bool estab,
           proc pos max pos,
           ...)
```

The fields of values of mode **channel** are not supposed to be accessed by users. In fact, one may imagine a transput implementation that uses different names for the fields, or a completely different set of fields. Algol 68, however, doesn't have secret fields.

In order to denote structure fields that-should-not-be-named, the authors of the Report resorted to a clever syntactic trick: to precede the fields names with a metanotion that produces an infinite number of `f`'s, which are obviously impossible to write. Something like:

```
A) F :: f, F ; F.

a) unmentionable field : F tag.
```

In order to represent the productions of the metanotion `F` in the representation language, they chose the symbol Aleph, since it represents an alepth-sub-zero number of `fs`. Thus the mode above would be written, using `%` for Aleph:

```
mode channel =
   struct (proc(ref book)bool % reset, % set,
                                % get, % put, % bi, % compress, % reidf,
           proc bool % estab,
           proc pos % max pos,
           ...)
```

The WG 2.1, however, didn't appreciate the joke, and ended using some strange glyph in both Report and Revised Report to represent Aleph.

2.2 Pseudo Comment

Meaning

The chapter 10 of the Algol 68 Revised Report describes the standard environment in which programs run. This chapter includes many code snippets with declarations and other entities that describe the interface provided by the standard preludes. However, code for the preludes is not given in full, suitable to be compiled form: many details are abstracted. Furthermore, the code that is actually provided in more detail is intended to serve as reference algorithms and is not necessarily the most efficient or even convenient way to encode the expressed logic.

In addition to regular comments, the code snippets in this part of the Report use *pseudo-comments*, which are delimited by a pair of bold tags **c**, and represent either a declarer or a closed-clause, as suggested by the contents of the pseudo-comment.

An example of the usage of pseudo-comments from the report:

```
op round = (real a) int:
     c An integral value, if one exists, which is
       widenable to a real value differing by not more
       than one-half from the value of 'a'
     c;
```

Many other texts, articles and books in the Algol 68 sphere make use of pseudo-comments.

It may be even possible to add support to compilers so they recognize them and compile them into some appropriate run-time diagnostic, which could be helpful in top-level programming.

See Also

- See Section 3.3 [Comment], page 9,
- [RR 10.1.3.step7]

2.3 Reference Language

Meaning

The *reference language* is a particular representation language for Algol 68, used and suggested by the Revised Report. All the code examples in the report are written using the reference language.

Implementations of the language are encouraged (but not strictly required) to use representations that are reasonably close to the reference language whenever possible.

The reference language prescribes a representation (typographical marks) for many of the symbols in the strict language, including the infinite set of TAX-symbols, but there is still some room for implementations to diverge in the following aspects:

– Some symbols are not given a representation in the reference language, but could be given one by some other representation language. An example is `brief-pragmat-symbol`. Note that this does not apply to the symbols `letter-aleph-symbol` and `primal-symbols` for which no representation should exist out of the representation of the preludes.

– Implementations can add any number of representations for `style-TALLY-letter-ABC-symbol` and `style-TALLY-monad-symbols`, and any terminal production of `STYLE other PRAGMENT item` and `other string item`.

– The representation language provides several alternative representations for some symbols, typically operator symbols. In that case, implementations of the reference language must implement at least one of these representations.

– If an implementation uses an alphabet generated by the meta-notion `ABC` that differs from the reference language, then the resulting language is a *variant of Algol 68*. This happens in translations of the reference language to different natural languages.

See Also

- Section 2.5 [Strict Language], page 6,
- Section 2.4 [Representation Language], page 5,
- [RR 9.4]

2.4 Representation Language

Meaning

A construct in the strict language, which consists in a production tree leading to a terminal production consisting in a sequence of symbols such as `'bold begin symbol' 'skip symbol' 'bold end symbol'`, must be represented somehow so it can be read by either a human interpreter or some mechanical interpreter such as a computer program. A *representation language* assigns some particular representation to each symbol.

It is thus possible to represent programs in the Algol 68 strict language in different ways, tailored to different purposes. A *publication language* will likely use rich text, fonts and/or graphical features in order to represent symbols such as `'bold begin symbol'`, `'bold letter a symbol'` or `'brief case symbol'`. A *programming language* to be used by programmers and text processing programs would typically use some stropping regime, resulting for example in `BEGIN SKIP END`. Finally, a *hardware language* could use a compact binary representation to ease the storage, transmission and automatic processing of the programs.

The Revised Report suggests and uses a particular representation language, which is the *reference language*. Implementations are encouraged to use representations that are reasonably closed to the reference language whenever possible.

See Also

- Section 2.5 [Strict Language], page 6,
- Section 2.3 [Reference Language], page 5,
- [RR 9.3]

2.5 Strict Language

Meaning

A construct in the *strict language* is a production tree produced by the Algol 68 two-level grammar's hyper-rules and meta-production rules. The production tree leads to a terminal production whose constituents are symbols.

For example, the following particular program in the reference language:

begin skip end

corresponds to a program in the strict language whose terminal production is:

```
'bold begin symbol' 'skip symbol' 'bold end symbol'
```

See Also

- Section 2.4 [Representation Language], page 5,

2.6 Taggle

Meaning

The Standard Hardware Representation defines a *taggle* as a nonempty sequence of letters and digits. Taggles are the constituents of tags. For example, in:

int `age of retirement = 65;`

The tag `age of retirement` is composed by three taggles: `age`, `of` and `retirement`. Note how typographical display features (space characters in this case) can appear between taggles in a tag.

See Also

- [RR 9.4.2.2.a]
- [SHR 1]

3 Language

3.1 Actual Parameter

Meaning

An *actual parameter* is the right hand side of an identity declaration, and consists of an unit whose context is strong. The value yielded by this unit, after strong coercion if necessary, shall be of the same mode than the one specified by the formal declarer. The value is ascribed to the defining identifier in the identity declaration.

For example, in the following identity declaration the actual parameter is `0`, which is in a strong context, and therefore gets widening to match the mode specified by the formal declarer **real**:

real **ratio** `= 0;`

Actual parameters also appear in routine calls, where they define the values passed to a procedure or an operator. This highlights that in Algol 68 the mechanism of associating formal parameters with actual parameters is the identity declaration: during a function call the internal values provided in the call get ascribed to the formal parameters. For example, in the following routine call:

```
multiply vectors ((10, 20), (1, 2));
```

The actual parameters are `(10, 20)` and `(1, 2)`, which are row displays of some **vector** mode.

Syntax

Simplified [RR 4.4.1.A,d]:

```
A) MODINE :: MODE ; routine.

d) MODE source for MODINE:
     where (MODINE) is (MODE), MODE source;
     where (MODINE) is (routine), MODE routine text.
```

Simplified [RR 5.2.1.1.c]:

```
c) MODE source:
     strong MODE unit.
```

We are not including here the rules for `routine text` but these can be found in [RR 5.4.1.a,b].

See Also

- Section 3.14 [Formal Parameter], page 16,
- Section 3.13 [Formal Declarer], page 15,
- [II 2.2.1]
- [RR 4.4.1.d]

3.2 Affirmation

Meaning

The *affirmation* operation is defined for integral and real values in the standard prelude as:

op `+ = (`**l int** `a)` **l int**`: a;`
op `+ = (`**l real** `a)` **l real**`: a;`

In both cases the given value is returned as such, resulting in that `a = +a`.

See Also

- [RR 10.2.3.3]
- [RR 10.2.3.4]

3.3 Comment

Meaning

Like in other programming languages, comments in Algol 68 programs are intended to document the program and their contents are ignored by the compiler: they are stripped out by the lexer. There are three styles of comments, that differ only by the delimiters used to begin and end the comment.

The first style uses **comment** to delimit the comment contents:

```
comment
  This program does foo and bar.
  Written by John Doe.
comment
```

The second style uses **co** to delimit the comment contents:

```
if not ok
then co This happens rarely co
     abort
fi
```

The third style uses # to delimit the comment contents:

```
print (whatever) # XXX remove trace #
```

Comments of different styles can be nested. Therefore up to three nesting levels is supported, which must be more than enough.

3.4 Completer

Meaning

Serial clauses contain zero or more declarations and at least one unit. When more than an unit is present then the value yielded by the last one is the value yielded by the serial clause. For example, in the serial clause:

```
(int tmp := a; a := b; a / tmp)
```

The value yielded by the unit `a / tmp` is the value yielded by the serial clause. Sometimes, however, it is useful to have more than one "exit point" in a serial clause. For example:

```
begin
  int tmp := a;
  a := b;
  if tmp = 0 then divbyzero fi;
  a / temp exit
divbyzero:
  0
end
```

When the unit `a / temp` in the serial clause above gets elaborated, the fact it is separated from the next phrase by an **exit** rather than a go-on symbol (semicolon) marks it as an exit point and therefore as a **mode**`-unit` or expression rather than a statement to be voided. The syntax mandates that an **exit** shall always be followed by a label.

A *completer* is the combination of an exit followed by a label.

```
    exit
label:
```

The units preceding a completer in a serial clause are **mode-units**, i.e. *expressions*. In contrast, other units in the serial clause but the last one are **void-units**, also known as *statements*.

Note that enquiry clauses are not allowed to contain labels, and therefore they can't contain completers. This is to prevent code in if-parts, else-parts and do-parts to jump back to the enquiry clause of their enclosed clause.

See Also

- Section 3.16 [Go-On Symbol], page 18,
- [II 3.1.4]

3.5 Contraction

Meaning

Certain language constructions which can be cumbersome for the programmer to write can be "contracted" into equivalent forms. The resulting shorter form is called a *contraction*. The constructions that can be contracted are:

- Collateral variable declarations.
- Collateral identity declarations (constant declarations).
- Identity declarations of routine modes.
- Priority declarations.

See for example the following collateral declaration of several variables of the same name, followed by it's corresponding contraction:

```
int size, int offset, int value := 1024;
int size, offset, value := 1024;
```

In the contracted form above, the same actual declarer (**int**) is shared among all the declared variables. The elaboration is still collateral, as implied by the comma separator.

The same can be applied to identity declarations. If we turn the variables above into constants, we have:

```
int size = 0, int offset = 0, int value = 1024;
int size = 0, offset = 0, value = 1024;
```

Note that you cannot mix variable declarations and constant declarations in the same contraction. If you tried to do:

```
int alignment = 1, int value := 1024;
int alignment = 1, value := 1024; # BAD #
```

The first collateral declaration is perfectly valid, but the resulting contraction is not. The reason is that in the variable declaration for `value` the mode at the left is an actual declarer that generates a new name to hold the value, whereas the mode at the left in the identity declaration for `alignment` is a formal declarer. This becomes more clear if we explicit the generator in the variable declaration:

```
int alignment = 1, loc int value := 1024;
int alignment = 1, value := 1024; # BAD #
```

Identity declarations of routines can become clunky:

```
proc([]real,real)real waverage = ([]real numbers, real weight) real:
begin
```

```
  ...
end
```

The corresponding contracted form, where the actual declarer is shortened to **proc**, would be:

```
proc waverage = ([]real numbers, real weight) real:
begin
  ...
end
```

Note however that the contraction form of a routine declaration is less expressive than the uncontracted form. In the contracted form it is required for the right hand side to be a routine text. That is not the case in the uncontracted form, in which the right hand side can be any unit yielding a routine of the expected mode, like in:

```
proc(int,int) transformer = (op = add
                             | int(int a, int b)int: a + b
                             | int(int a, int b)int: a * b);
```

Finally, collateral declarations of the priority of operators can also be contracted in the expected way:

```
prio isoneof = 6, prio ismanyof = 6;
prio isoneof = 6, ismanyof = 6;
```

Syntax

Simplified [RR 4.1.1.b:c]:

```
b) COMMON joined definition of PROPS:
     COMMON joined definition of PROPS, and also token, joined definition of PROP.
c) COMMON joined definition of PROP:
     COMMON definition of PROP.
```

Note that `and also token` is the comma symbol in most representations.

The rules above are used in the syntax of all the constructs mentioned in this article. For example the following simplified rule [RR 4.3.1.a] implements priority declarations:

```
a) priority declaration of DECS:
     priority token, priority joined definition of DECS.
```

Where `priority` plays the role of `COMMON` and `DECS` of `PROPS`. The rules for the other constructions are built the same way, so we are not including them here.

See Also

- [II 1.1.3,2.1.2,4.2.2.1]
- [RR 4.1.1.b:c]

3.6 Declarer

Meaning

A *declarer* is a source construct that specifies some particular mode. The simplest form of a declarer is the name of a mode, which can be one of the predefined primitive modes such as **int**, **real** or **compl**, or a mode indication previously defined by the programmer, such as **tree_node**. Declarers can get arbitrarily complicated depending the mode they specify. For example, the declarer corresponding to a ref to a row of structs is **ref[]struct (int** `age`**, string** `name`**)**.

Declarers specify modes, but they are not the same than modes. Different declarers can specify the same mode. This is the case for example with **union(int,real)** and **union(real,int)**, which specify the same united mode (unions are commutative and associative in Algol 68). Also, declarers can convey information that is not properly part of the mode it specifies. An example is `[10:20]`**int**, which denotes the mode row of integers but that also specify bounds which are not part of the mode. This is an example of actual declarer, that provides bounds to be used by a sample generator.

There are three kind of declarers, depending on the context where they appear and whether they convey bounds information or not: formal declarers, actual declarers and virtual declarers.

See Also

- Section 3.13 [Formal Declarer], page 15,
- Actual Declarer
- Virtual Declarer
- Mode
- [RR 4.6]

3.7 Development

Meaning

Development is the process of replacing a mode indication by its actual declarer. For example, given the following mode declaration:

```
mode tree_node = struct (int payload, ref tree_node left, right);
```

In the example below, which denotes a variable declaration, the occurrence of the mode indication **tree_node** is developed into the full structure mode definition:

```
tree_node top = (0, nil);
```

Which is then equivalent to:

```
struct (int payload, ref tree_node left, right) top = (0, nil);
```

The term "development" is not to be confused with "elaboration". The first applies to modes, the second to phrases and clauses. The first happens at compile-time, the second at run-time. It doesn't make sense to elaborate a mode, nor to develop a formula for example.

Usage

- *"The mode node develops into a structure"*

See Also

- [II 1.3.3.1]

3.8 Enquiry Clause

Meaning

An *enquiry clause* is a serial clause in a meek context that doesn't immediately contain labels, and therefore nor completers. Serial clauses that appear in the enquiry clause can feature labels and completers on their own.

Enquiry clauses (or just "enquiries") can be found in the following constructions:

- In conditional clauses, the if-part is an enquiry clause that must yield an **int**.
- In case clauses, the in-part is an enquiry clause that must yield an **int**.
- In loop clauses, if present, the while-part is an enquiry clause that must yield a **bool**.
- In conformity clauses, the case-part' is an enquiry clause that must yield an union.

Early drafts of the language used regular serial clauses in these contexts, which led to an unexpected problem. Consider the following conditional clause:

```
if int i := x + 10; xxx: i = 0
then ...
else ... i := 0; go to xxx ...
fi
```

In conditional clauses the if-part introduces a range that is visible in the rest of the clause. In the example above, if `x` is not zero when the clause is elaborated the `else` part gets elaborated and jumps back to the if-part. Similar situations happen in case, loop and conformity clauses. To avoid these difficulties, enquiry clauses got introduced with the restrictions explained above.

See Also

- Conditional Clause
- Loop Clause
- Enquiry Clause
- Conformity Clause
- [II 3.2.4.2]
- [II 3.2.4.3]
- [II 3.5.2]
- [II 3.6]

3.9 Environment Enquiry

Meaning

An *environment enquiry* is a kind of procedure defined in the standard prelude whose purpose is to provide information about the properties of the particular implementation used to compile the program.

Procedures implementing environment enquiries do not take any argument and yield a value of some appropriate mode. For example, the `max int` environment enquiry yields a value of mode **int**, whereas `null character` yields a value of mode **char**.

The section 10.2.1 of the Revised Report defines the environment enquiries that a conforming implementation must provide. These are:

int `int lenghts`
: 1 plus the number of extra lenghts of integers. This determines how many **long** entries in a longsety preceding **int** are meaningful in the implementation.

int `int shorts`
: 1 plus the number of extra shorts of integers. This determines how many **short** entries in a shorsety preceding **int** are meaningful in the implementation.

sizety int *`sizety`* `max int`
: The largest *sizety* integral value.

int `real lengths`
: 1 plus the number of extra lenghts of real numbers. This determines how many **long** entries in a longsety preceding **real** are meaningful in the implementation.

int `real shorts`
: 1 plus the number of extra shorts of real numbers. This determines how many **short** entries preceding **real** in a shortsety are meaningful in the implementation.

sizety real *sizety* `max real`
: The largest *sizety* real value.

sizety real *sizety* `small real`
: The smallest *sizety* real value such that both **sizety** `1 +` *sizety* `small real >` **sizety** `1` and **sizety** `1 -` *sizety* `small real <` **sizety** `1`.

int `bit lengths`
: 1 plus the number of extra longs of bits. This determines how many **long** entries in a longsety preceding **bits** are meaningful in the implementation.

bin `bit shorts`
: 1 plus the number of extra shorts of bits. This determines how many **short** entries in a shortsety preceding **bits** are meaningful in the implementation.

int *sizety* `bits width`
: The number of elements in a value of mode **sizety bits**.

int `bytes lenghts`
: 1 plus the number of extra longs of bytes. This determines how many **long** entries in a longsety preceding **bytes** are meaningful in the implementation.

int `bytes shorts`
: 1 plus the number of extra shorts of bytes. This determines how many **short** entries in a shortsety preceding **bytes** are meaningful in the implementation.

int *sizety* `bytes width`
: The number of elements in a value of mode **sizety bytes**.

op abs `= (`**char** `a)` **int**
: The integral equivalent of the character `a`.

op repr `= (`**int** `a)` **char**
: That character `x`, if it exists, for hich **abs** `x = a`.

int `max abs char`
: The largest integral equivalent of a character.

char `null character`
: Some character.

char `flip` The character used to represent **true** during transput.

char `flop` The character used to represent **false** during transput.

char `errorchar`
: The character used to represent unconvertible arithmetic values.

char `blank`
: The blank character.

See Also

- [RR 10.2.1]
- Section 3.12 [Flip and Flop], page 15,

3.10 Expression

See Also

- Section 3.22 [Mode-Unit], page 22,

3.11 Field Selector

Meaning

Structure modes consist on one or more fields, each of which have a mode on their own and a name. For example, this is how we would declare a mode for a node in a linked list:

```
node = struct (int id, real weight, ref node next);
```

The names of the fields in the structure, `id`, `weight` and `next`, are known as *field selectors* of the structure mode. Field selectors look like identifiers and are formed using the same rules, but they are not identifiers: they cannot be used on their own, and can only appear in a program text as part of a *selection*, like in `next` **of** `node`.

Note that the field selectors are integral part of the structure mode. The two structure modes **struct** `(`**int** `a,` **int** `b)` and **struct** `(`**int** `x,` **int** `y)` are different modes, since the field selectors of their fields are different. All the fields in a structure mode must feature a field selector: there is no provision in the language for "anonymous" fields.

Syntax

Simplified [RR 4.6.1.d]:

```
d) structured with FIELDS mode declarator:
     structure token, FIELDS portrayer of FIELDS brief pack.
```

Simplified [RR 1.2.1.I:J]:

```
I) FIELDS :: FIELD ; FIELDS FIELD.
J) FIELD :: MODE field TAG.
```

Note that `TAG` is the metanotion that produces identifier tokens.

See Also

- [II 2.4.1]
- [RR 4.6.1.d,1.2.1.I:J]

3.12 Flip and Flop

Meaning

During transput, the boolean values **true** and **false** are represented by two characters known as *flip* and *flop* respectively. The particular characters used for flip and flop are provided by two enviroment enquiries. Most implementation have used the character `T` for flip and `F` for flop.

See Also

- [RR 10.2.1]
- Section 3.9 [Environment Enquiry], page 13,

3.13 Formal Declarer

Meaning

A *formal declarer* specifies the mode of the value being ascribed in an identity declaration. It appears on the left hand side of an identity declaration, before the defining identifier. For example, in the following identity declaration the formal declarer is the mode indication **real**:

```
real pi = 3.141592;
```

Formal declarers also appear in routine texts as the modes of formal parameters, which shouldn't be surprising, since the mechanism of associating formal parameters with actual parameters in a routine call is the identity declaration: during a function call the internal values provided in the call get ascribed to the formal parameters. For example, in the following routine the mode indications **ref tree** and []**int** are formal declarers:

```
proc set tree weights = (ref tree node, []int weights) void:
begin
  ...
end
```

The mode specified in a cast is also a formal declarer. In the following example, where a cast is used in the firm context of an operator, the formal declarer is **real**:

```
c := real (2) * pi * r;
```

Note that (unlike actual declarers) formal declarers of row modes do not include bounds. If bounds are provided they are ignored, although some implementation may offer checking the bounds at run-time as a security measure. This could be particularly useful in formal parameters, where the run-time check would make sure multiples of the expected bounds get passed to the routine.

Syntax

Simplified [RR 4.4.1.c] (formal declarer in identity declaration):

```
A) MODINE :: MODE ; routine.

c) identity definition of MODE TAG:
     MODE defining identifier with TAG, is defined as token,
     source for MODINE.
```

Simplified [RR 4.6.1.r] (formal declarer in formal parameter):

```
r) MODE parameter joined declarer:
     formal MODE declarer.
```

Simplified [RR 5.5.1.a] (formal declarer in cast):

```
a) MOID cast:
     formal MOID declarer, strong MOID ENCLOSED clause.
```

In `4.4.1.c` the formal declarer is the `MODE` before the `defining identifier`.

Note that `is defined as token` is the equal sign character in the standard representation.

See Also

- [II 2.2.1]
- [RR 4.4.1.c,4.6.1.r,5.5.1.a]

3.14 Formal Parameter

Meaning

A *formal parameter* is the left hand side of an identity declaration, and consists of a formal declarer, which indicates the mode of the internal object being ascribed in the identity declaration, followed by a defining identifier to which the value will be ascribed. In the identity declaration:

```
real ratio = 2.71828;
```

The formal parameter is **real** `ratio`, the formal declarer is the mode indication **real** and the defining identifier is `ratio`.

Formal parameters also appear in routine texts, where they define which values are accepted as parameters by the routine when it is called. This highlights that in Algol 68 the mechanism of associating formal parameters with actual parameters is the identity declaration: during a function call the internal values provided in the call get ascribed to the formal parameters. For example, in the following routine:

```
proc multiply vectors = (vector a, vector b) vector:
begin
  ...
end
```

The formal parameters are **vector** `a` and **vector** `b`.

Note that formal parameters may appear "distributed" in the case of contracted definitions. In the following example:

```
real x, y, z;
```

There are three formal parameters, which are **real** `x`, **real** `y` and **real** `z`.

Syntax

Simplified [RR 4.4.1.a:c]

```
A) MODINE :: MODE ; routine.

a) MODINE identity declaration of DECS:
     formal MODINE declarer, identity joined definition of DECS.

b) routine declarer: procedure token.

c) identity definition of MODE TAG:
     MODE defining identifier with TAG, is defined as token, MODE source for MODINE.
```

Note that `is defined as token` is the equal sign character in the standard representation.

See Also

- See Section 3.13 [Formal Declarer], page 15,
- [II 2.2.1]
- [RR 4.4.1.a:c]

3.15 Frobyt

Meaning

A *Frobyt* or *FROBYT* is a **for-**, **from-**, **by-** or **to**-part of a loop clause. Loops featuring frobyts are endowed with an iterator, which may be explicit or explicit, and they will never run indefinitely.

The following loop clause has frobyts **for** and **to**, and has an explicit iterator `i`. It iterates 100 times:

```
for i to 100
do ... od
```

The following loop clause has frobyts `for` and `while`, and has an explicit iterator `i` used to determine whether we are in the first iteration. Since the loop is endowed with an iterator and it doesn't feature a **to**-part, it will iterate at most `max_int` times, at which point the iterator would overflow:

```
for i while node :/=: no node
do print ((name of node));
```

```
        if i > 0 then print ((",")) fi;
        node := next of node
    od
```

The following loop is endowed by an interator, this time implicit, due to the presence of the frobyt **to**:

```
    to 1000 while node :/=: no node
    do c process node c od
```

If the **by**-part of a loop clause is negative, then the **to**-part defaults to `min_int`.

3.16 Go-On Symbol

Meaning

The *go-on symbol* separates the phrases (declarations and units) in serial clauses. The concrete syntax for the go-on symbol is almost always the semicolon character `;`.

Consider for example the following closed clause, that consists on a serial clause with a declaration, a statement (voided assignation) and a final expression that determines the value of the serial clause:

```
    (int t := x; x := y; t)
```

In Algol 68 the go-on symbol always implies serial elaboration. In the example above, the declaration is elaborated first, then the assignation and finally the final expression.

Strictly speaking, it is not legal to put extra go-on symbols after the sequence of phrases: unlike in ALGOL 60, Algol 68 doesn't support the notion of "empty statement" (**skip** is used for that purpose instead) so the following code is invalid:

```
    begin foo;
          bar;
          baz;
    end
```

However, some implementations are lenient and just emit a warning about the superfluous go-on symbol. That is the case of both GNU Algol 68 and Algol68 Genie.

Syntax

Simplified [RR 3.2.1.b]:

```
    b) SOID series:
         strong void unit, go on token, SOID series;
         declaration of DECS, go on token, SOID series LABSETY;
         label definition of LAB, series with LABSETY;
         completion token, label definition of LAB, series with LABSETY;
         SOID unit.
```

See Also

- [RR 3.2.1.b]

3.17 Incestuous Union

Meaning

An *incestuous union* is an union that contains two or more alternatives whose modes are firmly related. Two modes M1 and M2 are firmly related if it would be possible to coerce a value of mode M1 to a value of mode M2 in a firm context, or to vice versa.

Consider the following union mode definition:

```
mode datum = union (int,ref int,proc int);
```

This union is incestuous, as both **ref int** and **proc int** values can be coerced to **int** in a firm context, by dereferencing and deproceduring respectively. If allowed in the language, this would lead to an ambiguity. After the assignation in the following example, the value stored in the union variable `mydatum` may either an **int** or a **ref int**:

```
int var;
datum mydatum := var;
```

To avoid these ambiguities incestuous unions are not allowed by the language and should be reported in compile-time errors by Algol 68 compilers.

Syntax

[RR 4.7.1.a]:

```
f) WHETHER MOODSETY1 with MOODSETY2 incestuous:
     where (MOODSETY2) is (MOOD MOODSETY3),
       WHETHER MOODSETY1 MOOD with MOODSETY3 incestuous
         or MOOD is firm union of MOODSETY1 MOODSETY3 mode;
```

See Also

- [RR 4.7.1.f]
- Firmly Related

3.18 Indicator

Meaning

An *indicator* is either an identifier, a mode indication or an operator. In all cases it specifies or denotes some other entity: identifiers specify the internal objects ascribed to them in identity declarations, mode indications specify modes associated to them in mode declarations, and operators specify routines ascribed to them in operation declarations.

The indicator in the following identity declaration is `pi`:

```
real pi = 3.14;
```

The indicator in the following mode declaration is **tree_node**:

```
mode tree_node = struct (int payload, ref tree_node next);
```

The indicator in the following operation declaration is `+`:

```
op + = (tree_node n1, tree_node n2) tree_node: ...;
```

Syntax

[RR 4.8.1.A,G]:

```
A) INDICATOR :: identifier ; mode indication ; operator.
G) TAX :: TAG ; TAB ; TAD ; TAM.
```

See Also

- Section 3.21 [Mode Indication], page 21,
- [II 1.1.1]
- [RR 4.8.1.A]

3.19 Longsety

Meaning

A *longsety* is a sequence of zero or more **long** bold tags. The term follows the fashion of the Revised Report, where the suffix `-ety` means "or empty".

The Algol 68 modes **int**, **real**, **compl**, **bits** and **bytes** can be prefixed with any number of **long** tag words. The effect of each **long** is to double the precision of the mode.

At some point, however, a "saturation" point is reached where the addition of extra **long** has no further effect on the mode. Where that point resides is up to the particular implementation.

For example, if the precision of **int** is four bytes or 32-bit, the precision of **long int** is 64-bit, and the precision of **long long int** is 128-bit.

A longsety can also be used in an integral denotation in order to specify the mode of the denotation. For example in the formula:

```
long 20 + long 30
```

The denotations **long** `20` and **long** `30` are of mode **long int**, which determines its precision. The reason why it is important to specify the mode in the denotations is that in Algol 68 it is not legal to widen to a mode having a different precision, so the following identity declaration is not legal:

```
long long int number = 100; # BAD #
```

This is because the mode of the denotation `100` is `int` whereas the expected mode is **long long int**. This can be achieved by a longsety in the denotation:

```
long long int number = long long 100;
```

Note that some Algol 68 implementations allow to widen to modes having a different precision.

Syntax

Simplified [RR 1.2.1.E]:

```
E) LONGSETY :: long LONGSETY ; EMPTY.
```

See Also

- Section 3.24 [Shortsety], page 23,
- Section 3.25 [Sizety], page 23,
- [II 2.7.2]
- [RR 1.2.1E]

3.20 Monads and Nomads

Meaning

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 code:

```
op plus = (int a, b) int: a + b,
   plus = (int a): a;
```

```
int val = 2 plus plus 3;
```

It is clear that the second instance of **plus** refers to the monadic operator and the first instance refers to the dyadic operator. If one would write **plusplus**, it would be a third different bold tag.

However, 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.

The Revised Report defines the sets of monads and nomads as metanotions, referring to symbols in an abstract way using symbolical names like "is at most" or "plus i times". These symbols do not always have a clear correspondence in click-able and printable symbols in all computers, so different implementations provide slightly different sets of monads and nomads. For example, in both GNU Algol 68 and Algol 68 Genie the set of monads is `%^&+-~!?` and the set of nomads is `><\/=*`.

Now that we know about monads and nomads, we can give the precise rules to conform valid operator names in Algol 68:

- 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.

Syntax

Simplified [RR 9.4.2.I,H] defines monads and nomads as metanotions:

```
H) MONAD ::  or; and; ampersand; differs from; is at most; is at least;
             over; percent; indow; floor; ceiling;
             plus i times; not; tilde; down; up;
             plus; minus; style TALLY monad.

I) NOMAD :: is less than; is greater than; divided by;
            equals; times; asterisk.
```

See Also

- [RR 9.4.2.I,H]

3.21 Mode Indication

Meaning

A *mode indication* is a bold word, an external object, that specifies a mode. Examples are **int** and **complex**. It is possible to introduce new mode indications via mode declarations. A mode indication is interchangeable with the mode it denotes within its range, which spans until the end of the current block. For example, the following mode declaration declares a mode indication **tuple**, which is visible until the end of the closed clause:

```
begin
```

```
    mode tuple = [2];
    ...
  end
```

Mode indications are very often abbreviated and referred to as “MOIDs” or “moids”.

Syntax

Simplified [RR 4.8.1.A,a]:

```
A) INDICATOR :: identifier ; mode indication ; operator.
G) TAX :: TAG ; TAB ; TAD ; TAM.

a) QUALITY new defining INDICATOR with TAX:
     TAX token.
```

Note that `TAB` is the meta-notion for a bold tag.

See Also

- [II 2.3]
- [RR 4.8.1.A,a]

3.22 Mode-Unit

See Also

- Section 3.36 [Void-Unit], page 30,

3.23 Ravelling

Meaning

Algol 68 unions are both commutative and associative. The associativity implies that, for example, given the declaration:

```
mode u1 = union (int,real);
```

Then writing **union (u1,string)** results in the mode **union (int,real,string)**. This associativity, which is conceptually clear, is syntactically implemented by an operation known as *ravelling* and consists in that, given a set of modes, some of them united, the united modes in the set are replaced by their components.

Syntax

Simplified [RR 4.7.1] is a predicate that determines whether a given set of moids ravels to a set of moods:

```
g) WHETHER MOIDS ravels to MOODS:
     where (MOIDS) is (MOODS), WHETHER true ;
     where (MOIDS) is
          (MOODSETY union of MOODS1 mode MOIDSETY),
       WHERE MOODSETY MOODS1 MOIDSETY ravels to MOODS.
```

See Also

- [RR 4.7.1]

3.24 Shortsety

Meaning

A *shortsety* is a sequence of one or more **short** bold tags. The term follows the fashion of the Revised Report, where the suffix `-ety` means "or empty".

The Algol 68 modes **int**, **real**, **compl**, **bits** and **bytes** can be prefixed with any number of **short** tag words. The effect of each **short** is to half the precision of the mode.

At some point, however, a "saturation" point is reached where the addition of extra **short** has no further effect on the mode. Where that point resides is up to the particular implementation.

For example, if the precision of **int** is four bytes or 32-bit, the precision of **short int** is 16-bit, and the precision of **short short int** is 8-bit.

A shortsety can also be used in an integral denotation in order to specify the mode of the denotation. For example in the formula:

```
short 20 + short 30
```

The denotations **short** `20` and **short** `30` are of mode **short int**, which determines its precision. The reason why it is important to specify the mode in the denotations is that in Algol 68 it is not legal to widen to a mode having a different precision, so the following identity declaration is not legal:

```
short short int number = 10; # BAD #
```

This is because the mode of the denotation `100` is `int` whereas the expected mode is **short short int**. This can be achieved by a shortsety in the denotation:

```
short short int number = short short 10;
```

Note that some Algol 68 implementations allow to widen to modes having a different precision.

Syntax

Simplified [RR 1.2.1.F]:

```
F) SHORTSETY :: short SHORTSETY ; EMPTY.
```

See Also

- Section 3.19 [Longsety], page 20,
- Section 3.25 [Sizety], page 23,
- [II 2.7.2] [RR 1.2.1.F]

3.25 Sizety

Meaning

A *sizety* is either a *longsety* or a *shortsety*. The term follows the fashion of the Revised Report, where the suffix `-ety` means "or empty".

For example, the sizety of a mode declared as **long long bits** is **long long**.

Usage

This term is useful in order to inquiry the number of size modifiers some particular mode has, like in:

- *"What is the sizety of* **file_size***?"*
- *"The sizety was wrong, I changed it to long long."*

Note that this is not exactly the same than asking for the precision of **FILE_SIZE**. The sizety implies some particular precision, but only indirectly.

Syntax

Simplified [RR 1.2.1.F]:

```
SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.
```

See Also

- Section 3.19 [Longsety], page 20,
- Section 3.24 [Shortsety], page 23,
- [II 2.7.2] [RR 1.2.1.F]

3.26 Statement

See Also

- Section 3.36 [Void-Unit], page 30,

3.27 Specification Part

Meaning

Each alternative in a conformity clause is composed by a *specification part*, which determines whether the alternative is chosen, followed by a unit that yields the value to which the clause elaborates in case the alternative is chosen. Each specification part contains a formal declarer followed by an optional defining identifier.

Consider for example the following conformity clause:

```
case datum
in (int i): i + 10,
   (real r): entier r + 10,
   (void): 0
esac
```

The first alternative has a specification part `(`**int** `i):`. It specifies that the alternative is chosen in case the enquiry clause `datum` is an **int**, and ascribes that value to the identifier `r` (which becomes a defining identifier) in the following unit. The second alternative has a similar specification part `(`**real** `r)`. The specification part of the third and last alternative, `(`**void**`)`, doesn't have an identifier.

Syntax

Simplified [RR 3.4.1.j,k]:

```
j) MODE specification defining new MODE TAG:
     declarative defining new MODE TAG brief pack, colon token.
k) MOID specification defining new EMPTY:
     formal MOID declarer brief pack, colon token.
```

See Also

- Conformity Clause
- Section 3.13 [Formal Declarer], page 15,
- [II 3.6]
- [RR 3.4.1.j,k]

3.28 String Break

Meaning

The intrinsic value of each worthy character that appears inside a string denotation is itself. The string `"/abc"`, for example, contains a slash character followed by the three letters `a`, `b` and `c`. A *string break* is a sequence of worthy characters that can occur inside a string or character denotation, that denotes some particular character.

String break sequences start with a *break character*. The Algol 68 Standard Hardware Representation allows implementations to define their own set of string breaks, but insists that the apostrophe should be the escape character. An example would be `'/` to denote a newline character, for example. The GNU Algol 68 compiler deviates from this and uses the backslash character to start string breaks emulating the familiar escape sequences used in C-like languages.

See Also

- [SHR 3.1]

3.29 Structure Display

Meaning

When a collateral clause is in a strong context where a primary yielding a structure value is expected, its constituent units are elaborated collaterally as usual, and the resulting values are used to conform the value of the fields of a new structure value of the expected mode. These collateral clauses are called *structure displays*, and play the role of structure denotations in Algol 68, even though they are not truly denotations.

The constituent units of a structure display are known as the *field positions* of the structure display. They are always elaborated in strong context with the mode of the corresponding structure mode field expected. The units are elaborated collaterally.

Consider the following structured mode with a couple of **real** fields and the declaration of a constant of that mode:

```
mode vector = (real x, y);
vector v1 = (3.14, 10)
```

The right hand side of an identity declaration is a strong context, and therefore the required mode is known at compile-time. In this case the mode expected is **vector**. The collateral clause `(3.14, 10)` can then recognized as a structure display of that particular mode, and its constituent units `3.14` and `10` become strong field positions with expected mode **real**. This allows the widening of `10` to `10.0` in this case.

When the context is not strong, however, structure displays cannot be recognized as such. Consider the following operator that adds two **vector**s:

```
op + = (vector a, b) vector:
   (x of a + x of b, y of a + y of b)
```

Again, the structure display in the body of the routine text ascribed to the operator `+` is in a strong context expecting a **vector**, so no problem there. But then consider the following formula that uses the just defined operator:

```
(1, 2) + (3, 4)
```

That is not valid code and a compiler will complain. The operands of a formula are in firm context, and the collateral clauses are recognized as such, which are *void units*. To remedy this we are forced to use casts in order to surround the collateral clauses with a strong context with required mode **vector**:

```
vector (1, 2) +  vector (3, 4)
```

Note that structure displays must have two or more field positions, or certain syntactic ambiguity known as *Yoneda's ambiguity* would arise: given **mode** = **m** (**ref m** m); **m** nobuo, yoneda; the assignation nobuo := (yoneda) is ambiguous. This difficulty can be easily circumvented by using the non-ambiguous m **of** nobuo := yoneda.

Syntax

Simplified [RR 3.3.1.e:h]:

```
FIELD :: MODE field TAG.

e) strong structured with FIELDS FIELD mode collateral clause:
     FIELDS FIELD portrait.
f) FIELDS FIELD portrait:
     FIELDS portrait, and also token, FIELD portrait.
g) MODE field TAG portrait:
     strong MODE unit;
h) *structure display:
     strong structured with FIELDS FIELD mode collateral clause.
```

Note that and also token is the comma symbol in most representations.

Note how the structure mode in e has at least two fields.

See Also

- [II 3.4]
- [RR 3.3.1.e:h]

3.30 Subname

Meaning

Selecting a name of a structure value results in another name, which is known as a *sub-name*. For example, given the following structure mode:

```
mode node = struct (int data, ref int next);
```

And a name of a value of mode **node**:

```
node anode := (0, nil);
```

Then selecting the field data of the structure name yields a sub-name with mode **ref int**:

```
data of anode := 100;
print ((data of anode))
```

It is said that the mode **ref node** is *"endowed with sub-names"*.

See Also

- [II 1.4.1.2]

3.31 Subscript

Meaning

A *subscript* is used to refer to some particular entry in a multiple's dimension while slicing. For example, in the slice:

```
foo[1,2,3]
```

The subscript 1 refers to the entry in the first dimension of the multiple with index 1. This doesn't necessarily means the first element: it depends on the bounds of the dimension. Likewise, the subscripts `2` and `3` refer to the values with indexes `2` and `3` in the second and third dimensions of the multiple. The action of applying a subscript is known as *subscripting*.

When subscripts are provided for all the dimensions of a multiple the result of the slice is an element from the multiple.

Syntax

Simplified [RR 5.3.2.e]:

```
e) subscript : meek integral unit.
```

See Also

- [II 1.5.2]
- [RR 5.3.2.e]

3.32 Symbol

See Also

- Section 3.33 [Token], page 27,

3.33 Token

Meaning

In most programming languages the tokens are the fundamental entities manipulated by the syntactic parser, and constitute the leaves of parse trees. A compiler component known as the lexical analyzer, or *lexer*, scans the source file and provides a flow of tokens. Typical tokens in these languages are strings, numerical literals and syntactic delimiters. For example:

```
                  foo (100)
                      |
          +-----------+-----------+
          |                       |
      identifier          +-------+------+
                          |       |      |
                          (      100     )
```

Where the terminal production would be the tokens `identifier`, `(`, `100` and `)`.

In Algol 68, on the other hand, the grammar extends all the way down to the individual letters, digits and symbols of a particular program: there is not a "lexical" specification separated from the "syntactic" specification. Therefore the individual digits of integral denotations, comments and its contents, string denotations and their contents *etc*, are all included in the grammar and are part of the parse tree. What is known as a *token* in other programming languages translates into the concept of *symbol* in Algol 68. The example above would be parsed in Algol 68 to something similar to:

```
                         foo (100)
                             |
           +-----------------+------------------+
           |                                    |
          tag               +-----------------+----------------+
           |                |                 |                |
  +--------+-------+    open-symbol       denotation      close-symbol
  |        |       |                          |
f-symbol o-symbol o-symbol             +------------+------------+
```

```
                                              |            |            |
                                          1-symbol     0-symbol     0-symbol
```

Where the terminal production would be the symbols `f-symbol`, `o-symbol`, `o-symbol`, `open-symbol`, `1-symbol`, `0-symbol`, `0-symbol`, `close-symbol`.

In conventional languages comments are considered a purely lexical artifact, meaning they get not tokenized, but simply skipper over and ignored by the lexer. Comments therefore never appear as tokens in the syntax of these programming languages, and can appear virtually anywhere in the program source without impacting the resulting parse tree.

On the other hand, Algol 68 accommodates comments (and the very similar pragmats) by defining a *token* as a syntactic construct composed by an optional comment (or pragmat) followed by a symbol. Note that this doesn't mean any symbol can be preceded by a comment or a pragmat. Comments and pragments can therefore appear anywhere the grammar generates a sequence of symbols via a sequence of tokens, but not where the grammar generates a sequence of symbols directly, such as in string denotations or inside other comments and pragments.

Syntax

The tokens are realized in the syntax by the following meta-production rule in [RR 9.1.1]:

```
f) NOTION token :
     pragment sequence option, NOTION symbol.
g) *token : NOTION token.
h) *symbol : NOTION symbol.
```

See Also

- Section 3.32 [Symbol], page 27,
- Section 3.3 [Comment], page 9,
- Pragmat
- [RR 9.1.1]

3.34 Trimmer

Meaning

A *trimmer* is used to refer to a subset of values in a multiple's dimension while slicing. For example, in the slice:slice

```
foo[1:5]
```

The trimmer `1:5` refers to the values with index `1` to `5` in the only dimension of the multiple `foo`. The action of applying a trimmer is known as *trimming*, and it always yields a slice.

The multiple resulting from the slice above will have lower bound `1` an upper bound `5`, but it is possible to "rebound" the result of trimming by using *revised bounds*. Consider:

```
[]int foo = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
[]int trim1 = foo[6:10];
[]int trim2 = foo[6:10 at 1];
[]int trim3 = foo[6:10 @ 1];
```

Where `trim1` has lower bound `6` and upper bound `10`, and both `trim2` and `trim3` have lower bound `1` and upper bound `5`. All `trim1`, `trim2` and `trim3` contain values `6, 7, 8, 9, 10`. Note that **at** and **@** are alternative representations of the "at token".

All components of a trimmer are optional. If the lower bound of the trimmer is omitted (as in `[:5]`) then it defaults to the lower bound of the multiple's dimension. If the upper bound of the trimmer is omitted (as in `[1:]`) then it defaults to the upper bound of the multiple's dimension. Both bounds can be omitted, resulting in `:` or simply an empty string, such as in the slice `foo[2,]`. If the lower bound revision part is omitted, the bounds of the resulting multiple are the same than the bounds specified in the trimmer (or implied by the trimmer.)

Syntax

Simplified [RR 5.3.2.f,g]:

```
f) trimmer : lower bound option, up to token, upper bound option,
             revised lower bound option.
g) revised lower bound : at token, lower bound.
```

See Also

- [II 1.5.2]
- [RR 5.3.2.f,g]

3.35 Vacuum

Meaning

Row displays contain zero, two or more constituent units. A row display that contains no units is known as a *vacuum*. The vacuum yields an empty multiple when evaluated, with whatever number of dimensions required by the appropriate row mode. Each dimension has a lower bound of one and an upper bound of zero.

The following example shows an identity declaration that ascribes a multiple that contains no elements to the identifier `empty`:

```
[]int empty = ();
```

The empty collateral clause `()` is in a strong context where a multiple of mode `[]int` is required, and therefore constitutes a row display. The following holds for the created multiple:

```
assert (lwb empty = 1);
assert (upb empty = 0);
assert (elems empty = 0);
```

The following example shows a similar identity declaration, but this time the row mode has three dimensions:

```
[,,]int empty cube = ();
```

Note how the vacuum is sill a single empty row display, i.e. it is not written `((()))`. All dimensions of the multiple have the same bounds:

```
assert (1 lwb empty = 1);
assert (1 upb empty = 0);
assert (1 elems empty = 0);
assert (2 lwb empty = 1);
assert (2 upb empty = 0);
assert (2 elems empty = 0);
assert (3 lwb empty = 1);
assert (3 upb empty = 0);
assert (3 elems empty = 0);
```

Syntax

[RR 3.3.1.k]:

```
k) *vacuum : EMPTY PACK.
```

See Also

- Row Display
- [II 3.5.1]
- [RR 3.3.1.k]

3.36 Void-Unit

Meaning

A serial clause contains one or more units. Of these, the units preceding a completer and the unit appearing last in the clause yield a value which in turn will be the value yielded by the complete serial clause.

Consider the following example:

```
begin
  int tmp := a;
  a := b;
  if tmp = 0 then divbyzero fi;
  a / temp exit
divbyzero:
  0
end
```

This particular serial clause contains one declaration, one label and four units.

The units `a / tmp`, which appears right before a completer, and and `0`, which is the unit appearing last in the serial clause, yield an integral value which is the result of the serial clause. These are called **int**-`units` or, more generally, **mode**-`units`. These are also known as *expressions*.

On the other hand the units `a := b` and **if** `tmp = 0` **then** `divbyzero` **fi** also yield values. For example, the assignation yields `a` of mode **ref int**. However, the value yielded by these units gets *voided* and discarded. These are **void**-*units*, also known as *statements*.

Syntax

Simplified [RR 3.2.1.b]:

```
b) SOID series with PROPSETY:
     strong void unit, go on token, SOID series with PROPSETY ;
     where (PROPSETY) is (DECS DECSETY LABSETY),
       declaration of DECS, go on token,
       SOID series with DECSETY LABSETY ;
     where (PROPSETY) is (LAB LABSETY),
       label definition of LAB,
       SOID series with LABSETY ;
     where (PROPSETY) is (LAB LABSETY) and SOID balances SOID1 and SOID2,
       SOID1 unit, completion token, label definition of LAB,
       SOID2 series with LABSETY ;
     where (PROPSETY) is (EMPTY),
       SOID unit.
```

In the hyper-rule above the first alternative matches a **void**-`unit`. Note that the unit is in a strong context with goal mode **void**, and therefore is subject to voiding.

See Also

- Section 3.26 [Statement], page 24,

- Section 3.4 [Completer], page 9,

3.37 Well-Formedness

Meaning

As is usual in modern programming languages Algol 68 supports an infinity of user defined modes, which are derived from the primitive modes[1]. There are two ways a programmer could shoot herself in the foot while defining modes:

- Values of the specified mode may require infinite memory.
- The mode may introduce ambiguities if values of that mode may be strongly coerced into themselves.

The first problem arises in modes that somehow include themselves. This can happen both directly, when a structure mode has a field of its own mode, or indirectly like in the following example:

```
mode thunk = struct (int content, thunk_extra extra);
mode thunk_extra = struct (char ext code, thunk extra thunk);
```

The second problem is more difficult to find in practice. The following rather artificial example is taken from II:

```
mode itself = ref itself;
ref itself who = loc itself;
```

If some particular mode is free of these problems, it is said that the mode is *well formed*.

Note how the root cause of non-well formed modes is in all cases some sort of recursion. Structural recursion can be avoided by what is known as *shielding*: a **ref** or a **proc** "shields" the referred or procedured mode that follows from causing recursion. For example, the following mode is well formed and actually quite useful:

```
mode tree node = struct (int data, ref tree node left, right);
```

The well-formedness of modes can always be detected at compile-time using a method known as the *ying-yang algorithm* that is specified in the Revised Report as a predicate grammar (see below).

Syntax

[RR 7.1.1.A]:

```
A) PREF :: procedure yielding ; REF to.
```

[RR 7.4.1.a:d]:

```
a) WHETHER (NOTION) shields SAFE to SAFE:
     where (NOTION) is (PLAIN)
        or (NOTION) is (FLEXETY ROWS of)
        or (NOTION) is (union of) or (NOTION) is (void),
     WHETHER true.

b) WHETHER (PREF) shields SAFE to yin SAFE:
     WHETHER true.

c) WHETHER (structured with) sheilds SAFE to yang SAFE:
     WHETHER true.
```

[1] In fact Algol 68 was the first language that seriously introduced the concept

```
d) WHETHER (procedure with) shields SAFE to ying yang SAFE:
     WHETHER true.
```

See Also

- [II 2.4.3]
- [RR 7.1.1.A,7.4,7.4.1.a:d]

3.38 Widening

Meaning

Widening is one of the six coercions. It is allowed in strong syntactic positions. This coercion transforms:

- Integers to real numbers of the same longsety.
- Real numbers to complex numbers of the same longsety.
- A **bits** value to an unpacked row of booleans.
- A **bytes** value to an unpacked row of characters.

Some implementations (like Algol 68 Genie) extend the meaning of widening by allowing transformations from, say, **int** to **long int** or from **long real** to **long long real**, but this is not allowed in the strict language, which requires using the **leng** and **shorten** operators instead.

Syntax

Simplified [RR 6.5.1.a:d]:

```
a) widened to SIZETY real FORM:
     MEEK SIZETY integral FORM.
b) widened to structured with SIZETY real field re
                                SIZETY real field im mode FORM:
     MEEK SIZETY real FORM;
     widened to SIZETY real FORM.
c) widened to row of boolean FORM:
     MEEK BITS FORM.
d) widened to row of character FORM:
     MEEK BYTES FORM.
```

See Also

- Coercion
- [RR 6.5]

3.39 Worthy Character

Meaning

In the 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.

Different Algol 68 implementations support different sets of worthy characters. The GNU Algol 68 compiler considers the following characters as worthy characters:

```
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 " # $ % & ' ( ) * + , - . / : ; < = >   [ \ ]
^ _ |   ! ? ~
```

See Also

- Section 2.4 [Representation Language], page 5,
- [SHR 1]

4 Implementation

5 Other

5.1 Orthogonal

Meaning

Adriaan van Wijngaarden championed the notion of *orthogonal programming language* and applied the notion in all its strength in the design of Algol 68. An orthogonal programming language is one such that it comprises a number of primitive independent concepts which are then applied in an orthogonal way. This makes the language very expressive, reduces the number of arbitrary rules (which the programmer has to remember) and avoids redundancy.

There are many examples of orthogonality in Algol 68. In fact, what is seldom found are arbitrary rules! One nice example is: take the notions of the comma separator `,` (in the Report that symbol is known as the *and also token*), collateral clauses, parallel clauses, declarations, multiple sub-scripting, actual argument passing, row display and structure display. These concepts are all independent. Now let's establish a rule: the comma separator implies collateral elaboration. Then let's apply this rule "orthogonally" by combining the concepts above.

Starting with the most obvious example, the units in the following collateral clause are elaborated collaterally, no surprise there:

```
(x * 2, y / 2)
```

If the following parallel clause there is still collateral elaboration, and it would be expected:

```
par begin generate data (), consume data () end
```

But then the indexes in the following multiple subscripting are also elaborated collaterally:

```
play voice ((monster at[get x (current map), get y (current map)]))
```

The actual parameters in the following procedure call are elaborated... collaterally!:

```
encrypt buffer (str, get random (seed))
```

The following contracted identity declarations are elaborated, surprise surprise, collaterally:

```
[]real randoms1 = get random (seed), randoms2 = get random (seed)
```

The field positions in the following structure display are also elaborated collaterally:

```
maintainer maint = (default name,
                    default url,
                    get last package (packages))
```

You get the point: there is no Algol 68 code where a comma separator doesn't imply collateral elaboration. Out of strings, comments and pragmats that's it. The programmer is only required to remember a number of N+M concepts (like the ones enumerated above) instead of the effect of combining them in N*M different combinations.

Algol 68 is not absolutely orthogonal, it has rules that introduce exceptions. An example is: "sizety modifiers can be applied to **int real**, **complex**, **bits**, **bytes** modes, but not to structured or rowed modes".

Usage

- "Algol 68 is an orthogonal programming language".
- "In this language concepts are applied orthogonally".
- "That rule you mention is not orthogonal".

See Also

- [RR 0.1.2]

5.2 Uninitiated Reader

Meaning

The original Report on the Algorithmic Language Algol 68, accepted in December 1968, was notoriously difficult to read, not only because of the usage of the two-level grammars and formal representation, but also because it lacked pragmatic descriptions.

The Revised Report, accepted at the end of 1973, incorporated many improvements in the described language, but also added many pragmatic descriptions to improve the readability. It also acknowledged the reported difficulties in the following famous paragraph in [RR 0.1.1]:

> "The Group wishes to contribute to the solution of the problems of describing a language clearly and completely. The method adopted in this Report is based upon a formalized two-level grammar, with the semantics expressed in natural language, but making use of some carefully and precisely defined terms and concepts. **It is recognized, however, that this method may be difficult for the uninitiated reader**."

It is to note that, although the readability problems were in their most part fixed by the Revised Report, which was a way more accessible document than the original report, the bad reputation of the later persisted and contributed to create FUD and the false impression that the described language (as opposed to the method of representation) was very difficult to learn.

Usage

The *uninitiated reader* or simply the *uninitiated* is sometimes used to refer to inexperienced programmers or users.

C. H. Lindsey dedicated his Informal Introduction to ALGOL 68 "To the Uninitiated Reader".

See Also

- [RR 0.1.1]

Concept Index

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 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 WRIT-

ING 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.