stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 168/81 JUNI

L.G.L.T. MEERTENS & J.C. VAN VLIET

ALGOL 68+, A SUPERLANGUAGE OF ALGOL 68 FOR PROCESSING THE STANDARD-PRELUDE

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B20

ACM-Computing Reviews-category: 4.22,4.12

ALGOL 68+, a superlanguage of ALGOL 68 for processing the standard-prelude

by

L.G.L.T. Meertens & J.C. van Vliet

ABSTRACT

In order to have an ALGOL 68 compiler process a version of the standard-prelude, it is necessary to define a superlanguage of ALGOL 68, ALGOL 68+, in which that version is written. The compiler must be able to accept programs (modules) written in ALGOL 68+. The present document gives a definition of ALGOL 68+, but only insofar as changes are needed to accommodate the standard-prelude itself (including the transput).

KEY WORDS & PHRASES: ALGOL 68, standard-prelude, portability

1. INTRODUCTION

In most programming languages, a program is allowed to use certain identifiers which have not been declared in the program, but are somehow known to the system, such as "sqrt", or "print". In most programming languages, they are known to the user because their names appear in a list in the manual; they are known to the compiler because the same list has been built into it, with references to run-time routines.

In the Revised Report on the Algorithmic Language ALGOL 68 [1], the situation is slightly different. The list and the references are both there, but are supplied as a sequence of declarations in a superlanguage of ALGOL 68. These declarations together constitute the "standard-prelude".

It would no doubt be possible to incorporate all this information in the compiler. This approach has some disadvantages, however. It would inevitably complicate both the compiler and the run-time system. It would also hinder the machine-independence of the compiler. Moreover, the "transput" part of the standard-prelude is very substantial in size, but can largely, and without undue loss of efficiency, be described in proper ALGOL 68 (see [4]). It would be nice to utilize this by actually implementing the superlanguage needed for the standard-prelude. It can then simply be compiled like any other "user-prelude", thus automatically creating a large part of the run-time system.

The present document gives a definition of this superlanguage, called ALGOL 68+, but only insofar as changes are needed to accomodate the standard-prelude itself (including the transput). The extensions needed for separate compilation are not treated. Rather, one should interpret ALGOL 68 as being the language defined by the Revised Report [1] <u>together</u> with the official separate compilation and modules proposal [2]. An earlier treatment of some of the problems can be found in GRUNE [6].

ALGOL 68+ is not a "safe" language, unlike ALGOL 68. Some new constructions are introduced which are only meaningful in a particular context. It is the task of the writer of the standard-prelude to take care that this does not lead to calamitous results. Of course, some checking will be performed by the compiler, but not so fully as for user programs. This need not give rise to many problems, since it is expected that very few people will actually write ALGOL 68+ texts. Phrases like "it is the responsibility of the standard-prelude author..." will highlight the "unsafe" parts.

It is undesirable if two compilers have to be maintained, one for ALGOL 68 and one for ALGOL 68+. Therefore, ALGOL 68+ should be defined in such a way that a very minor restriction gives a sublanguage which is identical again to ALGOL 68. The compiler may then run in each of two modes: restriction absent or enforced. This might be implemented by having a flag "standard-prelude mode" which is tested on many occasions.

A consequence of this would be a compile-time overhead. Therefore, the following approach is adopted: introduce a letter-aleph-symbol with some representation. The restriction is now that this representation is not recognized as being a LETTER-symbol, unless the compiler runs in standard-prelude mode. Since the test of being a LETTER-symbol may be implemented in such a way as to have a time cost independent of the size of the set of terminal metaproductions of ´LETTER´, the desired effect is obtained.

For the representation of the letter-aleph-symbol we need a worthy character (one available in the Standard Hardware Representation [3]) that can neither precede nor follow a "taggle" (see [3]) in ALGOL 68, and that is unlikely to be useful for other purposes in the standard-prelude. These considerations have led us to the choice of the escape character "´" (apostrophe). It may be used in ALGOL 68+ to build TAG- and bold-TAG-symbols, but also to build new symbols to be added to section 9.4.1 of the Revised Report. If, in the sequel, e.g., a "kludge-symbol" is introduced, it is understood to have the same representation as a bold-letter-aleph-letter-k-letter-l-letter-u-letter-d-letter-g-letter-e-symbol would have, viz., "<u>´kludge</u>".

It is preferable to represent ALGOL 68+ programs in the point-stropping regime and to use only one case of letters. In this document, underlining and lower-case letters are used to enhance readability.

The letter-aleph-symbol can be used in the same way as in the Revised Report for defining new "primitive" modes, as in

<u>mode</u> <u>sema</u> = <u>struct</u> (<u>ref</u> <u>int</u> ´).

It can also be used for what is accomplished in the Revised Report by a "?", as in

<u>proc</u> ´idf ok = (<u>string</u> idf) <u>bool</u> : ... .

For that purpose, also the applied occurrences have to carry an "´", unlike in the Revised Report, so one has to write

<u>if</u> ... & ´idf ok (idf) <u>then</u> ... .

The description given here is highly informal. The changes are sometimes described in the form of a hint of what they imply for the Revised Report+, sometimes as changes to the language as perceived by a user. References to sections of the Revised Report are of the form RRn, where n is the appropriate section number.

It is not the purpose of this document to describe the implementation of ALGOL 68+, but only its definition. Nevertheless, sometimes (less obvious aspects of) possible implementations may be discussed.

2. SPECIAL SYMBOLS

ALGOL 68 has an abundance of special symbols, such as "\", "≠" or "∔". Presumably, not every recipient of a machine-independent version of the standard-prelude (MISP) will be able to process these. Therefore, a representation has to be provided which only uses characters from the Standard Hardware Representation. Since a preprocessor may tailor the MISP to a local version with the locally available characters, the necessary extensions are no part of ALGOL 68+ and are not dealt with here.

3. CODE

In ALGOL 68+, a rather general mechanism is provided to allow the inclusion of non-ALGOL 68 text in the standard-prelude. It has the form

```
´code "some string".
```

This escape mechanism can, e.g., be used to define things which cannot be expressed in ALGOL 68 proper.

3.1. Identity- and operation-definitions

A source-for-MODE (not -for-routine) (RR4.4.1.d) of an identity- or operation-definition (but not a variable-definition) may be of the form

```
source for MODE +: code.
code: code token, strong row of character denoter.
```

(The notation +: stands for "add the following alternative(s) to the corresponding production rule". The notation +:: has a similar meaning for metaproduction rules. Also, metanotions like "MODE", "NEST", etc., are often omitted from the rules given in this document.)

It is assumed that for each applied-indicator the string is transmitted literally to the code generator, together with information about the parameters and operands, if any. No mode checking is performed. An example might be:

```
op (string, string) bool < = ´code "STRLT",
```

which offers the possibility of a more efficient implementation of string comparison than that in the Revised Report (although the latter is written in ALGOL 68 proper). The code escape is often necessary (e.g., for operations on semaphores). The idea is that the code generator will generate in-line code. Note that cases such as

```
proc (real) real nasty = sin
```

must be catered for also.

Disallowed are

proc sin = (real a) real : ′code "SIN"

(not a source-for-MODE but -for-routine) or

real var:= ′code "GENERATE"

(neither an identity- nor an operation-, but a variable-definition). The presumably intended effects, however, can be achieved by

proc (real) real sin = ′code "SIN"

and

ref real var = ′code "GENERATE".

3.2. Unspeakable modes

A new class of modes is introduced to cater for such things as ′book or ′buffer, modes which appear in the transput sections (cf [4]).

PLAIN +:: code TAG mode.

An actual-code-TAG-mode-TALLY-declarer (of a mode-definition) may be of the form

actual code TAG mode TALLY declarer: code.

Context condition: the string of the code must in some unspecified way determine (with an injective mapping) the ′TAG′.

Example:

mode ′book = ′code "BOOK".

Each different ′TAG′ yields a completely new, primitive, mode ′code TAG mode′, different from all other modes (of course). Only in the code-generation phase is that ′TAG′ inspected; until then the mode is merely passed on.

3.3. Yet another case

Code is also allowed at one other place, as treated in the next section (see ldec-MODE-source).

4. ARBITRARY SIZES

Rather than restricting the number of shorts and longs in the standard-prelude, we have chosen to devise a mechanism which mirrors that of the Revised Report. Together with new types of declarations, an additional type of case clause is provided to select on the number of longs and shorts. As for the transput section, genuine new modes like ´lint and ´outtype are introduced, together with the necessary coercions. Also, a coroutine-like mechanism is provided to allow for a reasonably efficient implementation of the transput.

4.1. L (L, K, S) outside transput

To treat the arbitrary sizes, new modes are introduced:

```
PLAIN +:: pseudo SIZETY SIZABLE;
          SIZETY bits; SIZETY bytes.
SIZABLE :: integral; real; complex; bits; bytes.
```

(The meaning of such new modes as ´bits´ and ´bytes´ should be clear. Also, ´complex´ is identified with compl, ´long complex´ with long compl, etc.) A pseudo-long-real-declarator, e.g., (see section 5) is written ´l long real. We may now have declarers like struct (´l real re, im), which may be abbreviated to ´l compl, but also union (´l int, int). The last declarer specifies an ill-formed mode. Therefore it is decreed that ´pseudo´ occur not within united modes at all. It is the responsibility of the standard-prelude author to take care that his modes are well-formed in this respect.

These new modes are used in a new type of declaration, which the standard-prelude author is supposed to use only in the outermost reach:

```
COMMON +:: ldec MODE identity;
           ldec MODE operation.

ldec MODE identity declaration:
    ldec token, formal MODE declarer,
        ldec MODE identity joined definition.
ldec MODE identity definition:
    defining identifier, is defined as token, ldec MODE source.

ldec MODE operation declaration:
    ldec token, operator token, formal MODE plan,
        ldec MODE operation joined definition.
ldec MODE operation definition:
    defining operator, is defined as token, ldec MODE source.
```

```
ldec MODE source:
    code;
    choice token, ldec MODE source choice list brief pack.
ldec MODE source choice:
    relational, length denoter, colon symbol, source for MODE1.
relational:
    is less than cum equals token;
    equals token;
    is greater than cum equals token.
length denoter:
    minus token option, integral denoter.
```

An example is given by

```
'ldec op ('l int) bool odd = 'choice
    (<= 0: 'code "CBIT(0)",
     >= 1: (long int a) bool:
               long bits width elem bin abs a
    ).
```

The above syntax rules allow the compiler writer to "move up" the ldec token in the context-free grammar used in syntax analysis in the following way:

```
LDECETY :: ldec; EMPTY.

declaration: LDECETY COMMON declaration.
ldec COMMON declaration:
    ldec token, COMMON declaration.
```

Furthermore, the source of an identity- or operation-declaration may then be an ldec-MODE-source. It is the responsibility of the standard-prelude author to use ldec-declarations in a proper way, so that the compiler need not check for the proper use of this feature. Note that one now needs the choice token (<u>'choice</u>) to allow for an LL(1) parse of sources.

For identification purposes an ldec-identity-declaration-of-TAG is willing to identify any applied-identifier-with-SITHETY-TAG. The size implied by 'SITHETY' determines the actual property returned by the identification process. For example, after the declaration

```
'ldec proc ('l real) 'l real sin = ...,
```

the applied-identifier "short sin" identifies this "sin" (unless it has already been identified in a deeper range) and the property corresponding to <u>proc</u> (<u>short</u> <u>real</u>) <u>short</u> <u>real</u> short sin is found. As for an efficient implementation, the stripping of 'SITHETY's need only be done after an initial failure to identify the tag.

A similar mechanism applies for ldec-operation-declarations: here the declaration is willing to identify an applied-operator provided that
(i) the 'TAO's are the same (of course) and
(ii) the reduced PRAM-modes of the formula are firmly coercible to those of the operation, where a mode is reduced by replacing all 'pseudo SIZETY's by 'EMPTY's.

By comparing the PRAM-modes of the defining and the applied side, a size can be determined by finding a (the first) occurrence of 'pseudo' at the defining side, and taking that 'SIZETY' that would have to be substituted for 'pseudo' here to make the modes agree. Again, this size determines the actual property returned by the operator identification process. Of course, it may happen that the actual operands are not strongly coercible to the operand modes of that property. This is not different from problems in identification if the representative mode determined by an h-function [5] is used. It is the responsibility of the standard-prelude author not to include ldec-declarations which would not be "independent" by virtue of different substitutions for 'pseudo' (and the Revised Report gives him no reason to do so).

An example. Consider the ldec-declaration

'ldec op ('l long compl) 'l compl shorten = ...,

and suppose we have a formula

shorten struct (short real re, real im) (~, ~).

The reduced PRAM-modes are in both cases struct (real re, im), so the declaration is willing to identify. The first occurrence of 'pseudo' is in a 'pseudo SIZETY' 'pseudo long'. The corresponding 'SIZETY' in the source mode is 'short', so 'pseudo' is 'short short' and the size determined is -2. The property returned corresponds to

op (short compl) short short compl shorten.

Next, it will be found that struct (short real re, real im) cannot be strongly coerced to short compl.

The information that is passed to the code generator for a code source also contains the actual size. This is needed for a case like

'ldec 'l int max int = 'code "MAXINT",

since otherwise there would be no way to tell which max int was required.

In the case of an ldec-source-choice-list-brief-pack the source-for-MODE1 is used of that ldec-MODE-source-choice whose relational holds {in an obvious way} for the determined size. It is not necessary that the 'MODE1' complies with the actual mode of the operands. Of course, this is useful only if ´MODE´ and ´MODE1´ reduce to the same mode and it is known that we are past the size where the distinguishing power in- or decreases. It is the reponsibility of the standard-prelude author to use only such ´MODE1´s.

This feature is powerful enough to cater for the last-random problem. (In RR10.5.1.a, an infinite sequence of variables, named "L last random", is declared; in principle, space has to be allocated for each of these variables. Note that it is assumed here that there is no distinction between standard- and particular prelude.) For, if the standard-prelude contains

```
´ldec ref ´l int last random = ´code "ALLOCLARA",
```

the code generator is passed the code-string for each applied occurrence of each of the L last randoms and has to return {code for the value which is} the name. Clearly, it can keep track of the sizes for which room has already been allocated and allocate global room for the newcomers.

Some care must be exerted to determine if properties of inner layers make a ´PRAM´ containing ´pseudo´ invisible. For example, op (int, long int) bool + does not render op (´l int, ´l int) ´l int + invisible, but op (int, union (ref int, void)) bool + does! The simple (but to some people niggard) implementation of operator identification in two scans, one for finding a potential defining property, and one for checking the property returned as for dependence of intermediate properties, would be correct.

4.2. ⟨L int⟩ etc. in transput

A class of genuine new modes is introduced:

```
PLAIN +:: lGORDIAN; reflGORDIAN.
GORDIAN :: integral; real; complex; bits.
```

These modes are known to the compiler; some modes which can now be defined are:

```
mode ´number = union (´lint, ´lreal);
mode ´simplout =
    union (´lint, ´lreal, ´lcompl, bool, ´lbits,
           char, [] char);
mode ´simplin =
    union (´reflint, ´reflreal, ´reflcompl, ref bool,
           ´reflbits, ref char, ref [] char, ref string).
```

There are two new coercions:

```
FIRM +:: tied to.
tied to lGORDIAN FORM:
    MEEK SIZETY GORDIAN FORM.
tied to reflGORDIAN FORM:
    MEEK reference to SIZETY GORDIAN FORM.

MEEK +:: untied to.
untied to union of MOODSETY1 SIZETY GORDIAN MOODSETY2 FORM:
    MEEK lGORDIAN FORM.
untied to union of MOODSETY1
        reference to SIZETY GORDIAN
        MOODSETY2 FORM:
      MEEK reflGORDIAN FORM.
```

The ´union ...´ presumably contains all discriminated sizes.

The two steps are needed to prevent a coercion cascade

```
int --> ´lint --> union (int, ...),
```

with corresponding ambiguity (since already int --> union (int, ...)). If a change of internal representation, equal to that in uniting, takes place during tying (where equivalent values of different but undiscriminated sizes receive identical representations), untying can be a pure syntax step like unchanging.

Example: If it is known that real lengths = 2 and real shorts = 1, the standard-prelude author can write something like

```
case y in
  (union (´reflint, ´reflreal, ´reflcompl) irc):
  begin
   ...
   case irc in
     ...,
     (ref real rr):
                 (intreal; ´string to real (t, rr)),
     (ref long real rr):
                 (intreal; long ´string to real (t, rr)),
     ...
   esac
  end,
  ...
esac.
```

He has to exert care to cover all discriminated sizes in the conformity-clause, since otherwise the out-clause may be selected unduly.

4.3. outtype and intype

Again, two new modes with coercions are introduced:

```
PLAIN +:: outtype; intype.
  {OUTTYPE, INTYPE:: see [4]}
```

The mode of, e.g., print is then

```
proc ([] union ('outtype, proc (ref file) void)) void.
```

The coercion is similar to uniting, but meek, to allow a coercion cascade

```
int --> 'outtype --> union ('outtype, ...).

MEEK +:: transtyped to.
transtyped to outtype FORM: MEEK OUTTYPE FORM.
transtyped to intype FORM: MEEK INTYPE FORM.
```

{In an implementation, both 'intype and 'outtype could look like struct ("proc" p, ptr any el). For each element x of mode m from the data list, the transtyping coercion could then result in a pair (L, v), where L labels the code for mode m and v points to x.}

No untying or straightening operation is provided. One could envision a procedure 'straightout of the form

```
proc ('outtype, proc ('simplout) void) void 'straightout =
                                         'code ...
# ('outtype x, proc ('simplout) void treat) void:
   c begin [] 'simplout y = straightout x;
       for j to upb y
       do treat (y[j]) od
     end
   c
#,
```

and similarly 'straightin. Such would obviate the need for one extra copy, since the implementation can traverse the data structure for out(in)type values.

The above implementation results in at least one extra procedure call for each simplout(in) value. The main drawback, however, is of a different kind: the transput implementation then has to provide treat-routines for each simplout(in) value and each kind of transput. For both formatless and binary transput these can be derived rather straightforwardly from the texts given in [4]. The formatted-transput routines, however, do not select the various possibilities according to the incoming simplout(in) value, but according to the type of the pattern at hand, since it is the type of the pattern which to a large extent determines how the value is transput. Selection on the basis of the incoming value would in this case lead to treat-routines which are largely identical.

Alternatively, one may use a scheme with two coroutines. The first coroutine traverses the intype (outtype) value, and halts at each simplin (simplout) value that it meets as part of the value that is being traversed. The second coroutine is then activated, which treats this simple value. After having done so, the first coroutine is re-activated. The straightforward implementation of coroutines in ALGOL 68 using semaphores is envisaged to be rather expensive, however. Since the application in question is rather simple (the two processes take turns in a neat way), a different scheme is adopted. A routine ´change turn is introduced and called whenever the two processes alternate. (This routine has to be implemented carefully so that the invariants of the parallel processing mechanism are maintained.) Part of, say, the routine put, then becomes

```
case y[k] in
   (proc (ref file) void p): ...,
   (´outtype t):
     begin ´simplout s;
       par begin
             (´traversout (t, s); goto done),
             do c treat ´s´ c; ´changeturn od
           end;
       done: ...
     end
esac.
```

Here, "c treat ´s´ c" is shorthand for a piece of program that can be written in ALGOL 68+.

The routine ´traverseout, which must be written in code, has to traverse the element t (using p of t). Each successive simplout value is assigned to s, after which ´change turn is called. For efficiency reasons one may choose to allocate enough room for all possible p´s of the program inside the routine ´traverseout. In this way, one need not use the full procedure call mechanism to "call" p.

A routine ´traversein plays a similar role on the input side.

5. NEW DECLARATORS

Rather than considering such declarers as void as mode-indications, they are viewed as new declarators. This class contains

```
{'l} {long | short}* int,
{'l} {long | short}* real,
                     bool,
                     char,
                     format,
                     void,
{'l} {long | short}* compl,
{'l} {long | short}* bits,
{'l} {long | short}* bytes,
                     string,
                     sema,
                     file and
                     channel.
```

This renders declarations such as mode 'l real = bool or mode format = proc (int) char illegal, and thus replaces the test 'or (NEST) is (new LAYER)' in RR 4.2.1.b. For most of these declarators the mode is known at the start of the compilation process (built-in). It is desirable, however, that the modes of format, string, sema and file be definable in the standard-prelude. To that purpose the compiler determines the mode specified by, e.g., the declarator file, by simulating the situation as though it had just read the mode-indication 'file. The standard-prelude may then contain a definition such as

```
mode 'file =
    struct (struct (ref 'book book, ... ) ').
```

(Note that even within the standard-prelude itself file may still occur as declarator, since 'file could occur there too as applied-mode-indication.)

6. THE TARGET MODE OF LWB AND UPB

Instead of the union of a sufficient number of modes, a new primitive mode is introduced:

```
MODE +:: rows.
rows declarator: rows token.
```

A corresponding coercion is introduced (using an obvious abbreviation):

```
FIRM +:: descriptored to.
UROWS :: union of ROWS1 of MODE ... ROWS9 of MODE mode.
```

```
        descriptored to rows FORM:
            MEEK ROWS of MODE FORM;
            MEEK UROWS FORM.
```

It is understood that the code generator is sufficiently sophisticated not to let the implied dereferencing result in code which copies the multiple value. Conceptually, a value of the mode 'rows' is a descriptor. The size of the descriptor is in general statically unknown, and therefore it is better to implement this value as a pointer to the descriptor. If the code generator is clever enough, it can generate better code if the actual operand is simple.

As for independency (i.e., the conditions that are imposed on defining-indicators to prevent ambiguous applications), we must be aware of the fact that op (union (string, char)) void upb, say, should render the standard upb invisible, even if the mode of the operands is [] bool. To this effect the test 'is firm' (RR 7.1.1.m) has to be extended with

```
        ... or MOID1 descriptors to MOID2 ...,
```

where we have

```
        WHETHER MOID1 descriptors to MOID2:
            where (MOID1) begins with (row),
                WHETHER (MOID2) is (rows);
            where (MOID1) is (NONSTOWED) or
                  (MOID1) is (structured with FIELDS mode),
                WHETHER false.
```

Note that this precludes at the same time a declarator of the form union ('rows, string). However, it is assumed that the standard-prelude author will only use 'rows for the operations lwb and upb.

7. MULTIPLE REPRESENTATIONS FOR OPERATORS

In order to avoid repetitious operation definitions which would have to be changed simultaneously (error prone), a construction is introduced which mirrors the { ... } of the Revised Report:

```
        DISPLAYETY: display; EMPTY.
```

At all places where 'defining operator' is used, one should now read 'defining operator DISPLAYETY'. Furthermore:

```
        defining operator display:
            choice token, defining operator list brief pack.
```

Example:

```
op (string, string) bool ´choice (<, lt) =
    ´code "STRLT".
```

Remarks:

1. To avoid an extra pair of brackets, the choice token has been introduced (see also section 4.1 above);
2. Some kludge will now be needed to handle the symbol ~. In a defining-operator-list-brief-pack, ~ should be interpreted as a representation for the operator not, although it would normally be interpreted as a skip-symbol in the contexts "(~,", ",~," and ",~)";
3. Note the fact that an open-symbol may now be followed by a NOMAD-symbol {<, >, /, =, x, *}. This need not present any problems for the syntax analyzer.

8. WEIRD GENERATORS

It is assumed that heap-generators have primal scope, so that the need for primal-generators disappears.

The "intermediate" generators for format handling and associated files are replaced by calls. The description in [4] needs three code routines:

```
proc (ref ref ref {sic} ´formatlist) void ´newformat =
     ´code ...
# (ref ref ref ´formatlist plist) void:
  plist:= c a newly created name which is made to refer
            to the yield of an actual-reference-to-
            formatlist-declarer and whose scope is equal
            to the scope of the value yielded by ´plist´ c
         := nil
#;

proc (int, int, format, ref ref ´formatlist) void
     ´extendformat = ´code ...
# (int count, cp, format f,
       ref ref ´formatlist plist) void:
  plist:= c a newly created name which is made to refer
            to the yield of an actual-formatlist-
            declarer and whose scope is equal to the
            scope of the value yielded by ´plist´ c
         := (count, cp, f, plist)
#;
```

```
proc (ref [][][] char) ref 'cover 'newcover = 'code ...
# (ref [][][] char sss) ref 'cover:
   c a newly created name which is made to refer to the yield of
     an actual-cover-declarer and whose scope is equal to the
     scope of the value yielded by 'sss'
   c
#.
```

The body of associate format may then be written

```
begin 'newformat (plist of ' of f);
      'extendformat (1, 1, format, plist of ' of f)
end.
```

The initialization of the file variable inside associate now becomes of the form

```
cover of f:= 'newcover(sss):= (...);
piece of f:= ...;
... .
```

REFERENCES

[1] VAN WIJNGAARDEN, A. et al, Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975), pp 1-236.

[2] LINDSEY, C.H. & H.J. BOOM, A modules and separate compilation facility for ALGOL 68, ALGOL Bulletin 43 (1978), pp 19-53.

[3] HANSEN, W.J. & H.J. BOOM, The report on the standard hardware representation for ALGOL 68, SIGPLAN Notices 12, 5 (May 1977), pp 80-87.

[4] VAN VLIET, J.C., ALGOL 68 transput, part II: An implementation model, Mathematical Centre Tracts 111, Amsterdam, 1979.

[5] BOOM, H.J., Note on balancing in ALGOL 68, ALGOL Bulletin 36 (November 1973), pp 17-24.

[6] GRUNE, D., Towards the design of a super-language of ALGOL 68 for the standard-prelude, Report IW 76/77, Mathematical Centre, Amsterdam, 1977.