ISSN 0084-6198

Algol Bulletin no. 40

AUGUST 1976

CONTENTS

The ALGOL BULLETIN is produced under the auspices of the Working Group on ALGOL of the International Federation for Information Processing (IFIP WG2.1, Chairman Professor J.E.L. Peck, Vancouver).

Facilities for the reproduction and distribution of the Bulletin have been provided by Professor Dr. Ir. W. L. Van der Poel, Technische Hogeschool, Delft, The Netherlands. Mailing in N. America is handled by the AFIPS office in New York.

The ALGOL BULLETIN is published approximately three times per year, at a subscription of $7 per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that cheques etc. are endorsed, where necessary, to conform to the currency control requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that any person should be completely debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:

Dr. C. H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
England.

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB37, AB38 and AB39 are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB40.0 Editor's Notes

Welcome to our fortieth issue! However, rather than indulge in an orgy of self-congratulation (that can wait until the fiftieth) I intend to be very brief.

My plea in the last issue for more papers for publication has met with a good response. Three of them appear in this issue, and there are more to follow. Please keep up the good work.

The principal content of this issue is the final version of the Hardware Representation for ALGOL 68, approved at the Working Group meeting in Munich. Implementors,please implement!

Of the other two documents approved at Munich, the one concerned with Modified ALGOL 60 (ALGOL 60M) appears in the August Computer Journal (again, implementors please implement). The other (ALGOL 68S) is still being worked on.

AB40.1 Announcements

AB40.1.1 ALGOL 60M

"A supplement to the ALGOL 60 Revised Report" by R.M. De Morgan, I.D. Hill and B.A. Wichmann is published in the Computer Journal, Vol. 19 No. 3, August 1976. This supplement takes the form of a series of modifications, as approved by the working group at its 1975 meeting, to the Revised ALGOL 60 Report. The full "Modified Report on the Algorithmic Language ALGOL 60", derived by elaboration of the supplement, will be published in the Computer Journal, Vol. 19 No. 4, November 1976.

AB40.1.2 Conference Proceedings: New Directions in Algorithmic Languages

The papers and discussion at the 1975 meeting of Working Group 2.1 at Munich, together with some additional papers, have been edited by Steve Schuman, and all ALGOL Bulletin subscriberswill automatically get a copy. Additional copies may be obtained, so long as stocks last, from Stephen A. Schuman, IRIA - Laboria, BP 5 - Rocquencourt, 78150 LE CHESNAY, France.

AB40.1.3 Conference Proceedings: Applications of ALGOL 68

University of East Anglia, Norwich, March 23rd - 26th, 1976. A limited number of copies of the proceedings are available, price £6.50, from Dr. V.J. Rayward-Smith, (Algol 68 Conference), School of Computing Studies, University of East Anglia, Norwich, England.

AB40.1.4 ALGOL 68 Conference: Strathclyde, 1977

A three-day conference on ALGOL 68 will be held at the University of Strathclyde from the evening of Monday, 28th to Thursday, 31st March, 1977. The main topics to be covered will be implementation, teaching and applications of ALGOL 68. Further details can be obtained from Dr. R.B. Hunter, Department of Computer Science, University of Strathclyde, Livingstone Tower, 26 Richmond Street, Glasgow, G1 1XH, U.K.

AB40.1.5 ALGOL 68 Test Set

An extensive Test Set for ALGOL 68 compilers has been assembled at the Mathematical Centre in Amsterdam, Netherlands. The Test Set comprises (at the moment) 160 programs covering the entire full language as defined in the Revised Report on the Algorithmic Language ALGOL 68. It should be useful to anyone implementing ALGOL 68 (both at desk-top level and at third-drawer level) and be interesting to everybody concerned with ALGOL 68. It has been used commercially by Control Data to test their ALGOL 68 system.

The Test Set is available in book form for $4 (including a print-out of the latest version) and on magtape for $12 if tape is supplied or $20 if no tape is supplied.

Please address requests to the editor: Dick Grune, Mathematical Centre, Tweede Boerhaavestraat 49, Amsterdam.

AB40.1.6 ALGOL 68 Bibliography

This bibliography has been prepared,on behalf of the III, by Robert Uzgalis, Computer Science Department, School of Engineering and Applied Science, U.C.L.A., Los Angeles, California 90024, U.S.A., from whom copies may be obtained.
Buz would also like suggestions for future editions, together with copies of any papers that you may write on ALGOL 68 related topics, for inclusion in his A68 Information Repository.

AB40.1.7 Textbook: A Practical Guide to ALGOL 68

This new textbook, by Frank G. Pagan,is published by Wiley in both hard-back and soft-cover editions. It is intended both as an introductory text to those with no previous programming experience, and for those transferring from other languages.

It teaches programming in the structured style (jumps are relegated to the last chapter) and there are copious exercises. The language described is strictly in accordance with the Revised Report and, although there are a host of minor misprints, technical errors are conspicuously absent.

AB40.1.8 Module protection already working in SIMULA.

The Simula Development Group (with representatives for the existing 10 Simula implementations) has accepted in September 1975 a construct for module protection similar to the proposal by C.H. Lindsey in AB39 p. 20. Attributes of a module can be made inaccessible by means of the hidden and the protected specification. The use of two specifications makes it possible to let certain attributes be accessible only inside a module, other attributes accessible within a system of cooperating modules, other attributes accessible wherever the module itself is accessible.

See further Simula Newsletter, No. 1, 1976, or (in more detail) DECsystem-10 Simula Gazette, No. 3, Vol. 2, 1976 (Can be ordered from Section 142, Swedish National Defense Research Institute, S-104 50 Stockholm 80 , Sweden.)

The Module protection scheme in SIMULA is already implemented and available in Release 3 of the DECsystem-10 Simula system. The system is available for a price of about $ 100 from the Swedish National Defense Research Institute, Section 142, S-104 50 Stockholm 80, Sweden.

The DECsystem-10 Simula system implements the full Simula language (comparable in power to Algol 68 or PL/I, and based on the same ideas, but closer to Algol 60). The system is especially aimed at conversational applications, allowing one or more conversational terminals to be connected to an executing program, and the system has a conversational debugging system where the user at the conversational terminal can set breakpoints and query about internal data values.

A Codasyl type DBMS system entirely written in Simula, for use by Simula programmers or at a user terminal, is distributed with the DECsystem-10 Simula system.

A system for using separately compiled modules in a way which in no way endangers the security of the language and which does not incur any extra run time overhead is also available with the DECsystem-10 Simula system.

Jacob Palme 1976-03-26

AB40.1.9 ALGOL 68 Revised Report - Erratum

The following erratum should be applied to the Springer (i.e Acta Information edition of the Revised Report.

P. 118 9.4.1.b. "times ten to the power symbol" ✱ $_{10}$ => $_{10}$ \ ✱

AB40.2.1 Initialized Generators

30 March 1976

The Editor,
Algol Bulletin.

Dear Sir,

Many generators in Algol 68 programs involve the execution of code at run-time (e.g. setting up dope vectors or reserving flexible storage); some also involve code to be executed at the end of the life of a value (to free flexible storage, for instance). As computers must therefore cater in some way for this, the thought comes to me that the representation of any generator could be extended to include the optional specification of user written procedures to be executed at these points in the life of the generated value. These procedures would have one parameter (the generated value) and yield a void result. I haven't worked out the scope implications, but a structure including some procedures which might be initialised in the user generated code (partial parametrisation might be very useful to loosen scope restrictions) could be the basis of another proposal for generating modules in Algol 68; some of the Simula class facilities might also become available. The main objection to this scheme is that all references to objects of the module would have to be selections; but set against this is the automatic hiding of local values (in the procedure) and the use of an identifier to choose a particular copy of the module. Thus a declaration is similar to Lindsey's invoke and a selection similar to access. If this proposal could be made workable, not only would it require less change to the language than other proposals, but it would also allow such constructions as:

```
MODE CLEARINT = [1 : n] INT
¢ brief pack containing ¢ (¢ initialisation ¢ PROC(REF CLEARINT ia) VOID:
                              FOR i TO n DO ia [i] := 0 OD,
                          ¢ closedown ¢
                            SKIP).
```

Perhaps your readers can produce some more exiting examples.

Yours faithfully,
Paul Hodges
53 Ellerton Road,
Surbiton,
Surrey KT6 7TY

AB40.2.2 The Name of the Language

The Editor,
Algol Bulletin

30 March 1976

Dear Sir,

I wonder whether there are others who feel, as I do, that the biggest mistake of Algol 68 is its name. I have found it a great disadvantage when explaining the language to unsympathetic (and some sympathetic) friends.

A large part of the computer industry considers 'Algol' to be synonymous with run-time inefficiency and a total lack of any facilities of use outside numerical analysis, and people with this view consistently think that Algol 68 is a slight botch-up of Algol 60 which is as inefficient. Thus the name works against it.

Those who know about Algol 60 and Algol 68 are not prone to this failing, but I cannot be alone in feeling that in both speech and writing, 'Algol 68' is a clumsy phrase; and after one implementation was called Algol 68R the habit has grown worse. Also the R in Algol 68R means that we have to speak of 'revised Algol 68R' in full, rather than using that Algol 60M device mentioned in AB39.

The motor car industry is forsaking numbers for names (e.g. British Leyland's 18/22 is now the Princess and its 1100/1300 the Allegro). The computer Industry has a fine record of names (Pegasus, Deuce, Fortran, Algol, Jovial, Coral) which has lasted longest in the language field. Can we not follow Wirth (Euler and Pascal) and show the outside world (and even the hardware manufacturers) that modern life with computers is not all numbers. How about a competition for a real name for revised Algol 68? - or is it too late?

Yours faithfully,

Paul Hodges
53 Ellerton Road
Surbiton,
Surrey KT6 7TY.

Editor's Note: Yes, it is indeed too late, but it is an interesting thought, nevertheless.

AB40.4.1 Integral Division once more

J. Hilden

(University Institute of Medical Genetics, Copenhagen).

Integral division once more

The proposal R_i by L.G.L.T. Meertens, AB39.4.3, was actually employed in the now obsolete IBM7094 implementation of the list processing language L7 by P. Jensen and me. Unfortunately this implementation was never documented except in Danish. To be specific, the divide and modulo operations ensured that

$$0 \leq \mathrm{val}(\ a \ \underline{\mathrm{mod}}\ b\ ) = \mathrm{val}(\ a \ \underline{\mathrm{mod}}\ (-b)\ ) \leq \mathrm{abs}(\mathrm{val}(\ b\ )) - 1,$$

and

$$\mathrm{val}(\ a\ ) = \mathrm{val}(\ (a \div b) \times b + a \ \underline{\mathrm{mod}}\ b\ ),$$

in self-explanatory notation. These equations imply that Meertens' requirement

$$\mathrm{val}((a + n \times b) \div b) = \mathrm{val}(a \div b + n)$$

is met. However, we had to warn potential users against the following pitfall:

$$\mathrm{val}(\ -8 \div 7\ ) = \mathrm{val}(\ -(8 \div 7)\ ) = -1,$$

whereas

$$\mathrm{val}(\ (^{L}\!-8) \div 7\ ) = -2.$$

Thus, whenever a divide or modulo operation is textually preceded by a unary or binary minus the programmer would have to be a bit careful.

A special kind of integer division that occurs very often in practical programming is the following: how many boxes of capacity b are needed to ship a pieces? Denote the desired operation by split. Then 100 split 20 = 5 and 101 split 20 = 6, etc. Using conventional integer division, one would have to write

```
boxes := (a-1)÷b + 1;
```

but even this clumsy piece of program may not have the desired effect unless a is strictly positive. Using L7 division, it would work even for a=0; it might even be replaced by

```
boxes :=-(-a)÷b;
```

Neither is as easy and clean as we would want. The natural remainder operation associated with split answers the question: how many empty places remain when the a pieces have been packed? E.g., 100 splitrem 20 = 0, while 101 splitrem 20 = 19.

It would be interesting to see if anyone could suggest a programming syntax that displayed the symmetry between the aforementioned Meertens-L7 divide and modulo operations on the one hand and the split-splitrem operations on the other. The problem is find a neat way of indicating whether remainders should be taken from below or from above, as it were, when b does not divide a.

AB40.4.2

A method of implementation of independently compiled routine texts in ALGOL 68

Ivan Sklenář
(INSTITUTE OF SOLID STATE PHYSICS, PRAGUE)

1. Introduction

In this paper the design is presented for the independent compilation of routine texts in TESLA ALGOL 68. (TESLA ALGOL 68 is a dialect of ALGOL 68.) Independently compiled routine texts (further ICRT) can be connected with calling module via formal parameters only.

ICRTs are linked (in load module) as relocatable segments and they are placed on the stack of local values in run-time, therefore they overlay one another or the data, too.

2. TESLA ALGOL 68

TESLA ALGOL 68 have been implemented on the TESLA 200 computer. It is a small computer with IBM 360-like structure. TESLA ALGOL 68 compiler requires 64 kByte memory at least including 10 kByte supervisor. Instruction codes can contain direct or based addresses. Due to the small memory size the segmentation of the compiler as well as of the object program is necessary. Operating system admits using relocatable segments (in the linkage loader and in the binary loader), but the TESLA 200 standard programming techniques use unrelocatable segments and direct addressing (i.e. localization of segments is fixed by linkage editor). Moreover library object modules cannot be linked into relocatable segments.

3. The use of ICRT from the user's point of view

It is supposed that interface between ICRT and calling

modules[x] is possible via formal parameters only.

The source program consists of the main modul and ICRT modules. An ICRT module is preceded by a statement defining the entry name of ICRT. This statement also defines the boundaries between modules in source program. All modules in the source program may be replaced by their object binary relocatable form.

The syntax of ICRT is the following:

mode declaration list option, go on symbol, routine text. (1)

In the module calling the procedure that is defined by means of ICRT the procedure can be referred only in the identity declaration of the following form:

procedure symbol, virtual declarer pack option,
virtual declarer option, procedure identifier, equals symbol,
pragmat symbol, extern symbol, extern name, pragmat symbol. (2)

An example:

```
proc (int, int) int gcd = pr extern gcd pr
```

The pragmat on the right of the equal symbol substitutes routine text defined by the ICRT identified by the extern name.

If ICRT is recursive it must contain the declaration of the form (2) of itself, e.g.:

```
. ICRT PROC2
mode dint = struct(int a,b);
(real a, dint b) real:
begin
proc (real,dint) real rekurs = pr extern proc2 pr;
:
```

---

x) The calling module may be the main modul or an ICRT module

```
x:=rekurs(z,(1,1));
:
end
```

(statement ". ICRT PROC2" defines the extern name of the ICRT).

4. Compiler aspects

4. 1 Compilation of ICRT

On the TESLA 200 the linkage loader and segmented program do not contain facilities allowing to change addresses during loading relocatable segments to the memory.

Therefore addresses in an ICRT are compiled according to the following rules:

- all local addresses in instructions codes are based,
- all addresses in the address-like constants (i.e. constants defined in an assembler by DC A(...)) are changed during the loading using a table TABRLD attached to the modul by the compiler,
- all external addresses (external from the point of view of the ICRT) are explicitly given in the resident segment 0,
- the communication between calling and called module is realised by means of the table of segments (TABSEG),
- the entry address of the modul is the first address of the module.

4. 2 Compilation of the declaration (2)

At the place of procedure identity declaration having a routine text at its right hand side the routine descriptor is formed in the local area in the run-time stack. The descriptor consists of the routine identification (used in run-time error messages), routine entry address (EA) and a pointer to the start of the local area of the procedure statically surrounding the considered one.

If the routine text is replaced by an ICRT indication the sequence of instructions generating the routine descriptor contains a call of a routine loading the segment containing ICRT on the run-time stack. The routine places the procedure entry address into the run-time descriptor and into the table TABSEG. The entry into the TABSEG is derived from the ICRT segment number. The segment address is deleted from TABSEG, when the range of the declaration is left.

4. 3 The loading and deleting routines

All ICRT modules are linked as separate segments. The presence of the ICRT segments at the stack is indicated in the TABSEG table. The TABSEG contains one item for each segment. This item contains the address of the segment when it is present on the stack, or zero otherwise. The TABSEG is inicialized by zeros.

Let i be the number of a segment. The routine loading the ICRT segment into memory works as follows:

Step 1: Set TOP2 equal to TOP (TOP is a register containing the first free address on the stack).

Step 2: If TABSEG(i) $\neq$ 0 then step 6 is taken.

Step 3: Load the segment number i on the stack beginning on the address TOP.

Step 4: Add the value of TOP to all addresses referred in the TABRLD table.

Step 5: Set TOP equal to the TABRLD start (now the TABRLD is overlapped by the stack).

Step 6: Set EA equal to TOP2.

At leaving of the range of some declaration (2) the addresses of the corresponding ICRT segment must be deleted from

the TABSEG. In this case a routine is called comparing all items in the TABSEG with TOP. If TABSEG(i) is greater than the value of TOP the TABSEG(i) is set equal to zero.

The routine is called (and the corresponding instructions generated) only at exit from the range containing the declaration (2).

5. Pre-editor

5. 1 Purpose of the pre-editor

With respect to the limited facilities of the linkage loader it seems to be useful in the case of segmented programs to insert between the compiler and the linkage loader a program facilitating the interface between the compiler and the linkage loader. This program will be called a pre-editor.

The main input file of the linkage loader containing the object modules and linkage control statements is denoted ⌘5. The linkage control statements describe the overlay structure, contain references to libraries etc. The file ⌘5 is the output of compilers. It contains translated modules, copies of the pre-compiled object modules and linkage control statements creating together with the source modules the input stream of the compiler.

The function of the pre-editor is to reorganize ⌘5. The pre-editor runs in two passes: in the first one the ⌘5 is processed as an input file, in the second one the reorganized ⌘5 is created. The function of the pre-editor will be illustrated on the following example:

a) The input of the compiler

```
begin
.
.
.
proc (real) real p1 = pr extern p1 pr
.
.
.
end
```

master module in source form

```
. ICRT P1
(real x) real:
begin
.
.
.
proc (real) int p2 = pr extern p2 pr
.
.
.
z:=sin(x);
.
.
.
end
```

routine text p1 in rource form

```
. BAR P2
.
.
.
. EØF
```

routine text p2 in object form

end of file

b) The output from the compiler (=the input for pre-editor = the contents of the file #5)

```
. BAR MASTER
.
.
.
. BAR P1
.
.
.
. BAR P2
:
:
```

master module in object form

routine text p1 in object form

routine text p2 in object form

```
. EØF                                          end of file
```

c) The output of the pre-editor (= the input of the linkage loader)

```
. LINK 0                                    ⎫
. BAR MASTER                                ⎪
.                                           ⎪
.                                           ⎪
.                                           ⎪
. BAR TABSEG                                ⎪
.                                           ⎪  resident segment
.                                           ⎪
.                                           ⎪
. SETEX SIN                                 ⎭
. LINK 1,R                                  ⎫
. BAR P1                                    ⎬  relocatable
.                                           ⎪
.                                           ⎪  segment number 1
.                                           ⎭
. LINK 2,R                                  ⎫
. BAR P2                                    ⎬  relocatable
.                                           ⎪
.                                           ⎪  segment number 2
.                                           ⎭
. EØF                                          end of file
```

The control statements are marked by capitals. The program contains 3 modules, the master module and the ICRT module P1 are in source form, the third one is in object form. The statement ". BAR P2" precedes the records of object form of ICRT module P2.

Compiler output contains all modules in the object form. Pre-editor inserts some control statements among the object modules. The control statement ". LINK 0" heads the resident segment (=segment 0). To this segment all standard library modules must be attached. This is performed by the statement ". SETEX SIN" attaching the module for standard function sin from the library. Note that this function is called from ICRT P1 only.

The control statements ". LINK 1,R" and ". LINK 2,R" head

~~the ICRT segments ". LINK 1,R" and ". LINK 2,R" head~~ the ICRT segments. The number 1 (or 2) is used for identification in the table TABSEG. This number is supplied by the linkage loader into the sequence of the instructions for the calling of the segment loading (see 4.3). The module TABSEG is generated by the pre-editor and contains the table TABSEG (the length of this module depends on the number of ICRT modules).

5. 2 Mode checking

Comment records are inserted inside the object moduls during the compilation. The comments contain the initernal representation of virtual declarers used in the declarations of type (2) and the virtual declarers derived from the formal declarers in the head of ICRT. This information is collected during the first pass of the pre-editor and after the first pass this information is used in mode check.

5. 3 Libraries of the ICRT modules

All ICRT modules must be present at the output from pre-editor. From this fact it is obvious, that the user libraries of the ICRT modules cannot be processed by the linkage loader. During the first pass of the pre-editor all the extern referrences are collected. After the input file =5 is exhausted, the first pass of the pre-editor takes up the processing of the library file. (This library file contains user´s pre-compiled ICRT object modules.) The modules entry addresses of which corresponds to some unresolved external referrences are selected from the library and processed like the ICRT modules in the input file. After the library file is exhausted, the first pass is finished and all the remaining unresolved referrences are placed into the control statements ". SETEX".

6. Advantages and disadvantages

The main advantages of this concept are the following:

- code of ICRT is overlaied with the data on the stack;
- the errors produced by the conflict between the overlay structure and the logical structure of calling are eliminated. The user can never make an overlay error;
- the mode checking can be done during the compile-time (in the pre-editor);

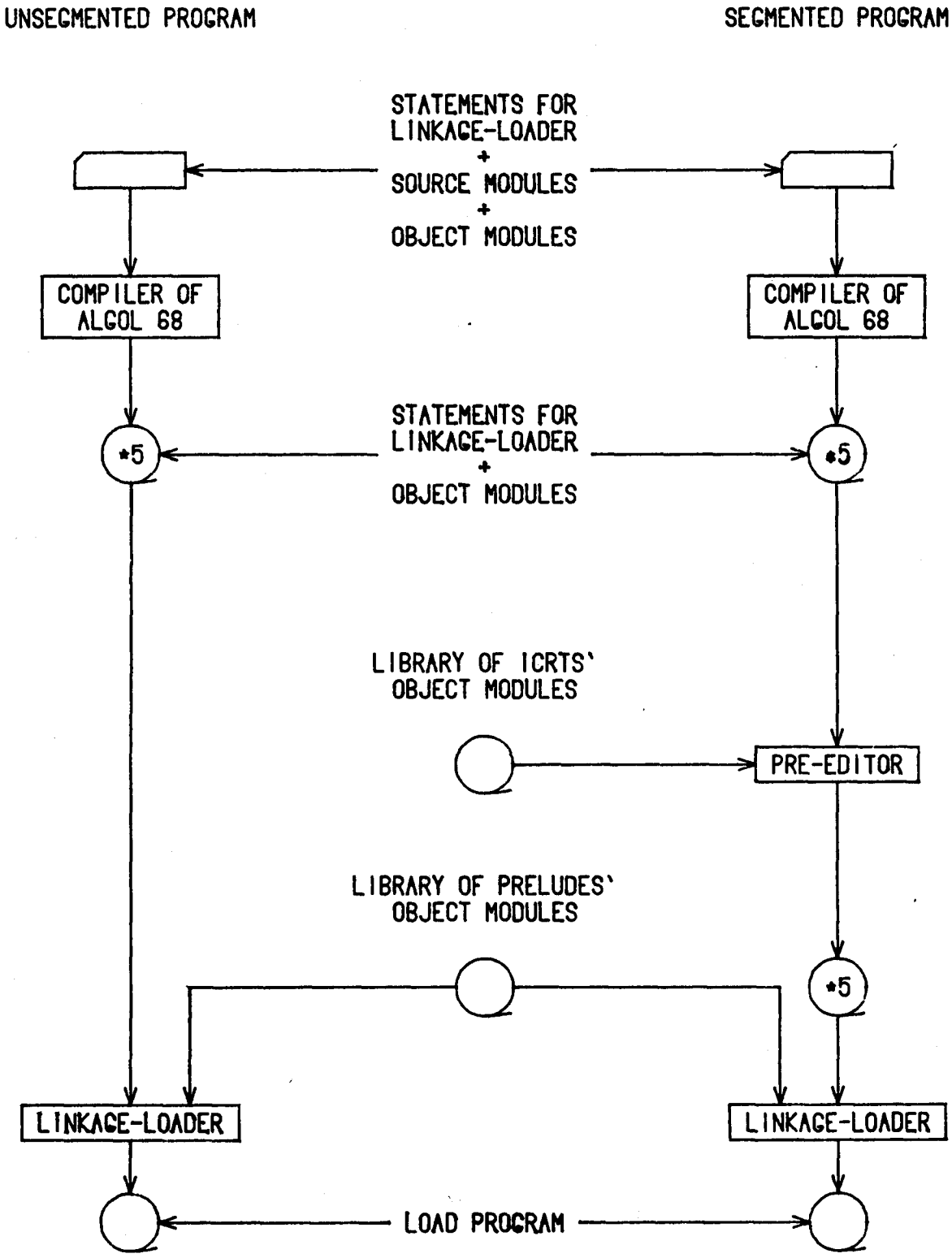
The disadvantages are the following:

- the library modules must be attached to the resident segment not only in the case that they are called from the resident segment, but also if they are called from the ICRTs only;
- the interface between the calling module and ICRT is possible via formal-actual parameters mechanism only;
- a greater administration in the range exit.

References

- Revised Report on the Algorithmic Language ALGOL 68,
Ed: A. van Wijngaarden, B.J.Mailloux, J.E.L.Peck, C.H.A.Koster, M.Sintzoff, C.H.Lindsey, I.G.L.T.Meertens and R.G.Fisker,
Acta Informatica 5, 1-236 (1975)

- K. Kleine, Segmentation of Algol 68 programs,
Conference on Experience with Algol 68, Liverpool, 1975.


UNSEGMENTED PROGRAM
SEGMENTED PROGRAM
STATEMENTS FOR LINKAGE-LOADER
+
SOURCE MODULES
+
OBJECT MODULES
COMPILER OF ALGOL 68
COMPILER OF ALGOL 68
*5
STATEMENTS FOR LINKAGE-LOADER
+
OBJECT MODULES
*5
LIBRARY OF ICRTS' OBJECT MODULES
PRE-EDITOR
LIBRARY OF PRELUDES' OBJECT MODULES
*5
LINKAGE-LOADER
LINKAGE-LOADER
LOAD PROGRAM

PROPOSAL FOR A SIMPLER SYNTAX FOR THE ALGOL68 UNIT

May 25th, 1976
R. Haentjens
KMS Brussels

1. Introduction

Units in ALGOL68 are used to program the more primitive actions. Their syntax defines how a particular unit is to be parsed (Revised Report 5).

This syntax is defined in such a way that normally the hierarchy of actions is as you think it should be; in neutral situations some choice is made.

Examples:

{1} age of person := 28
is parsed as (age of person):=28 . If it were parsed as age of (person:=28) , this would almost never reflect the programmers intention.

{2} age of person [2]
is parsed as age of (person[2]) . This is a neutral case. If the programmer wants the opposite parsing, he should use parentheses:

{3} (child of jim) [2]
The construction "child[2] of jim" does not exist in ALGOL68.

{4} ref int(pointer) := 0
In the original ALGOL68 one would have written "(ref int : pointer) := 0" because "ref int : pointer := 0" was parsed as "ref int: (pointer:=0)".

An ideal unit syntax should be simple but still define an acceptable hierarchy of primitive actions.

2. Proposal

The complexity of the unit syntax depends heavily on the syntax of the primitive actions. Though the syntax of some of these primitive actions has been changed in ALGOL68 during revision, no effort has been done to simplify the unit syntax at the same time. In our proposal we show that such a simplification is possible without many changes to the syntax of the primitive actions.

Please compare the syntax charts for ALGOL68 and the proposal. The description method of J.M. Watt, J.E.L. Peck and M. Sintzoff is used but details are left out in both charts where the proposal changes nothing to the ALGOL68 syntax.

3. Discussion

The syntax of the proposal is simpler than the ALGOL68 one. Its most interesting feature is the distinction of three levels in the unit syntax: the right associative level, the formula and the left associative level.

Changes to the syntax of primitive actions:

1. The identity relation is no longer a balance and it associates to the right. The right hand side being a unit, the "nihil" can be brought in the same category as "jump" and "skip".
2. The syntax of the cast and the selection is the inverse of the corresponding ALGOL68 syntax. That makes it possible to use a primary in the cast instead of an enclosed clause.

The examples of §1, written in the proposed syntax:

{1} person s age := 28

{2} person[2] s age

{3} jim s child[2]

{4} pointer ref int := 0

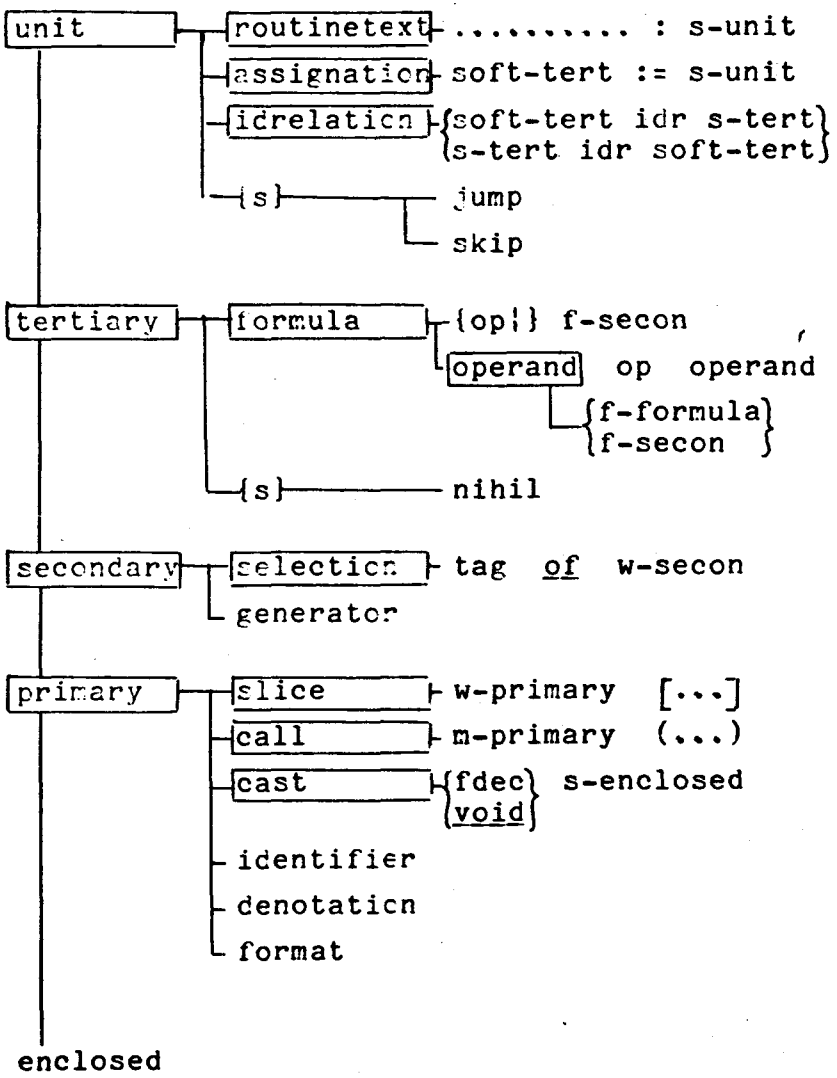
The asymmetry of the identity relation in the proposal is something one could argue about. Semantically, the identity relators are symmetric, but almost always a cast is needed to obtain the desired effect, so the syntax of the complete construction becomes asymmetric. In the proposal this asymmetry is frozen with the cast as left hand side:

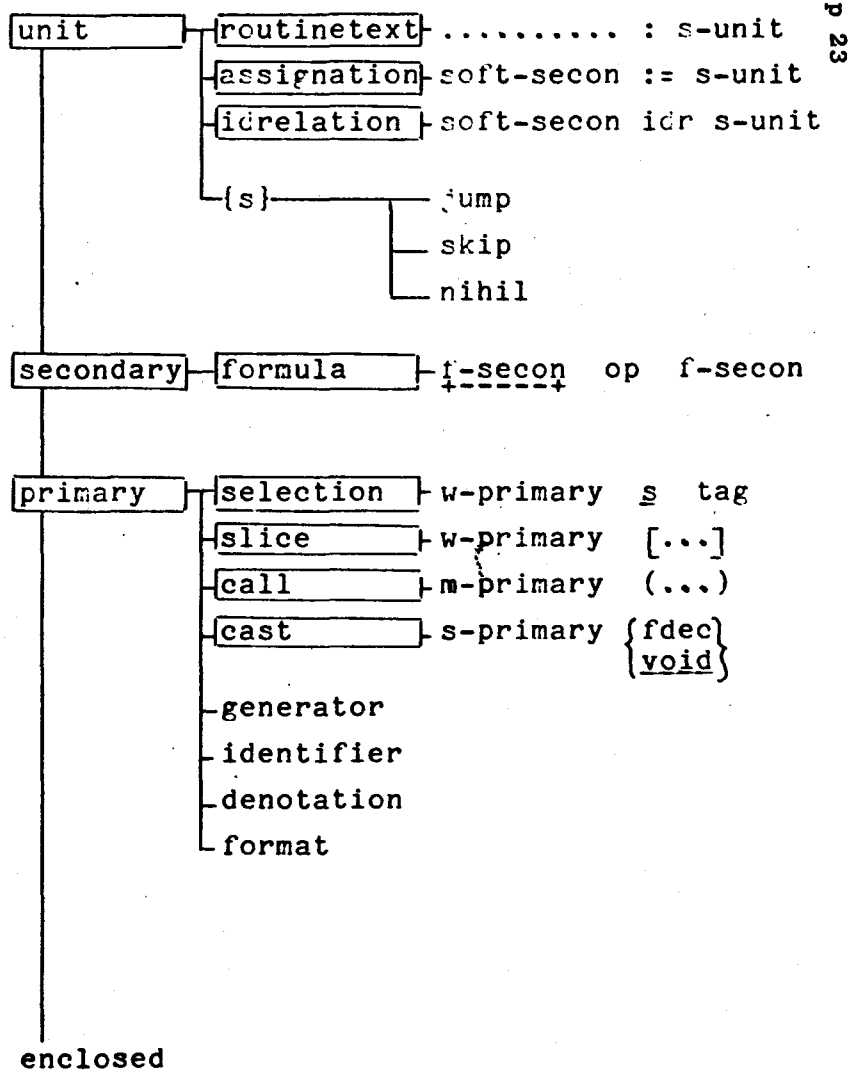
{ALGOL68} ref node(son of currentnode) :=: nil
{proposal} currentnode s son ref node :=: nil

It is as if the declarer influenced the meaning of the identity relator: "compare two ref node values".

In fact the identity relation is a parasite construction in ALGOL68. All normal values are compared using operators (equal, not equal). Due to the orthogonal conception of the modes, the concept "variable" does not exist in ALGOL68. The "value of a variable" is obtained by a coercion (dereferencing) which cannot distinguish a "pointer constant" from a "variable", nor an "indirect pointer constant" from a "pointer variable". That orthogonallity makes it impossible to use the same operator e.g. to compare the values of two integer variables and to compare two pointers to integers.

Editor's Note: It is of course too late to consider this as an actual modification to ALGOL 68, but it is an interesting example of what might have been and these points should certainly be kept in mind when designing future languages. Something similar to the s proposal was indeed considered at the Fontainebleau meeting of the Working-Group, but was dropped, with regret, as being too great a charge to the existing language structure.


unit
routinetext ..... : s-unit
assignation soft-tert := s-unit
idrelation {soft-tert idr s-tert
s-tert idr soft-tert}
{s} jump
skip
tertiary
formula {op} f-secon
operand op operand
{f-formula
f-secon}
{s} nihil
secondary
selection tag of w-secon
generator
primary
slice w-primary [...]
call m-primary (...)
cast {fdec
void} s-enclosed
identifier
denotation
format
enclosed


unit
routinetext ..... : s-unit
assignation soft-secon := s-unit
idrelation soft-secon idr s-unit
{s} jump
skip
nihil
secondary
formula f-secon op f-secon
primary
selection w-primary s tag
slice w-primary [...]
call m-primary (...)
cast s-primary {fdec
void}
generator
identifier
denotation
format
enclosed

AB 40.5

The Report on the Standard Hardware Representation for ALGOL 68

Wilfred J. Hansen
University of Illinois at Urbana-Champaign

Hendrik Boom
Mathematisch Centrum

0. Introduction

At its September, 1973, meeting in Los Angeles, Working Group 2.1 of IFIP created a Standing Subcommittee for ALGOL 68 Support. The January, 1975 meeting of this Subcommittee in Boston discussed at length a standard hardware representation and authorized a Task Force to draft a proposal incorporating the conclusions of that meeting. An initial draft was presented to the June, 1975, meeting of the Informal Information Interchange at Oklahoma State University. Many improvements and alterations suggested at that meeting have been incorporated into this final version. All suggestions were valuable, even those that served only to stimulate discussion. Subsequently, this report was accepted by the August, 1975, meeting of Working Group 2.1 in Munich and forwarded to IFIP.

A standard hardware representation is desirable for several reasons:

- First, together with the Report*, it provides a complete definition of a single language. As implementations have developed their own solutions to the problems of representation, there have arisen many related languages that differ considerably in appearance. To read or write a program for an alien implementation, a programmer has been required to make a considerable mental readjustment of deep habits. One might argue that no precise standards exist for natural language punctuation and typesetting, but the argument

---

* In this document, "the Report" refers to the Revised Report:

A. van Wijngaarden, et al., Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica, v.5, Fasc. 1-3, Springer-Verlag (Berlin, 1975).

References to it are in the form of "R" followed by a section number. To avoid confusion, references to sections in this report are prefixed with "*".

does not apply to artificial languages intended to be read by machines.

- Second, processors other than compilers may be defined for ALGOL 68 programs; for example, macro processors, cross-reference programs, and print formatters. Such processors may be used by all implementations only if the tokens they accept are defined by a standard.

-Third, a single representation convention will promote portable programming. This document specifies a minimum character set that every compiler must accept and the maximum that may be used in a portable program. Consequently, program transportation requires only one-to-one transliteration; the transliterator need not determine the extent of strings, comments, and format-texts.

Several goals have been addressed in creating this standard hardware representation: it should require only a small, widely available character set*; it should minimize parsing problems; it (or some subset)

---

* With the exception of square brackets, the set of worthy characters is a subset of most versions of ISO-code, ASCII, and EBCDIC:

ISO Standard 646: 7 bit coded character sets for information processing interchange. An earlier version of this standard was considered in Lindsey, C. H., "An ISO-code representation for ALGOL 68", ALGOL Bulletin 31 (March, 1970), pp. 37-60 (corrected in AB 32.1.3).

ANSI, USA Standard Code for Information Interchange (X3.4-1968), American National Standards Institute (New York, 1968).

ANSI, American Standard Hollerith Punched Card Code (X3.26-1970), American National Standards Institute (New York, 1970) {defines a version of EBCDIC}.

IBM Corp., IBM 1403 Printer Component Description, Order no. GA24-3073, 1970 {defines the "TN-chain" version of EBCDIC}.

Hansen, Wilfred J., "A Revised ALGOL 68 Hardware Representation for ISO-code and EBCDIC", UIUCDCS-R-73-607, University of Illinois, Urbana (November, 1973); revised as "An ALGOL 68

should be teachable; it should be possible to write portable programs that process other programs; it should conform to the Report, existing usage, and usage in other languages; and, above all, it should be a practical, congenial means of expressing ALGOL 68 programs. With the exception of three representations {see *3.7} and the "string break" {see *3.1}, an implementation following this document is an "implementation of the reference language" {R9.3.c}.

1. Definitions

Worthy character - one of these sixty 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
Ø 1 2 3 4 5 6 7 8 9
space " # $ % ' ( ) * + , - . / : ; < = > @ [ ] _ |
```

{This document defines a representation of an ALGOL 68 program as a sequence of worthy characters and newlines.}

Base character - a "character" available at an installation. {Each such character is a composite of some set of marks and codes agreed upon by local convention. The input to a compiler is a sequence of base characters.}

{What I see is that, whereas there is only one form of excellence, imperfection exists in innumerable shapes....

The Republic, Plato}

Disjunctor - a typographical display feature {R9.4.d}, the start or end of a program text, or any worthy character other than a letter, digit, or underscore. {Tags and bold words are delimited by disjunctors.}

Adjacent, follow, precede - Two character strings are "adjacent" if there are no intervening characters or typographical display features. If one string is said to "follow" or "precede" another, they are also adjacent.

Hardware Representation for ISO-code, ASCII, and EBCDIC" (December, 1974).

Bold word -
i) any representation composed of bold-faced letters or digits in the reference language {R9.4} {i.e., bold-TAG-symbols and the representations shown as bold in R9.4.1}, or

ii) a symbol represented by a bold word, or

iii) the characters written for a bold word as specified below {*3.5}.

Tag - a TAG-symbol {R9.4.2.2.a} {"End of file" is a tag.}

Taggle - a nonempty sequence of letters and digits. {As used in *3.5.1, "End of file" has three taggles.}

2. Representation of ALGOL 68 Constructs

For each worthy character an implementation must provide a base character different from the base character for any other worthy character. The mapping between worthy and base characters should be chosen so as to minimize confusion while paying due regard to prevailing usage. {For example, an implementer should avoid assigning a base character to an unrelated worthy character and also avoid using a character to represent something other than that which it represents in the Report.}

An implementation may augment the worthy characters with the twenty-six lower-case letters. The two cases of a letter are equivalent except as provided in *3.1 and *3.5.2. {This equivalence promotes portability; for example, it prevents distinction between tags that differ only by the case of one letter.}

The Report specifies {R9.3.b} that a "construct in a representation language" is obtained by replacing symbols with their representations. In this document, a representation is specified for each symbol in terms of worthy characters. Constructs in the representation language are encoded for communication and computer processing by replacing each worthy character with its corresponding base character and inserting typographical display features {where permitted}.

3. Specific Representations

3.1 String-items

The set of string-items {R8.1.4.1.b} is the set of worthy characters (as possibly augmented with lower-case letters) excluding quote and apostrophe but including the quote-image-symbol and the apostrophe-image-symbol. The intrinsic value of each worthy character is itself; the upper- and lower-case versions of a letter have distinct intrinsic values. The quote-image-symbol is written as two adjacent quotes and its intrinsic value is a quote. The apostrophe-image-symbol is written as two adjacent apostrophes and its intrinsic value is an apostrophe. {A single apostrophe may be used as an escape character in some implementations.}

An additional typographical display feature, the "string break", is provided for use exclusively within string- and character-denotations. It is written as

- a quote, followed by
- one or more typographical display features other than string break, followed by
- another quote.

{When a string-denotation must be continued to more than one line, a string break permits the number of spaces at the end of one line to be indicated and permits the next line to be indented without confusion.}

3.2 Other-Pragmat-Items

Any sequence of characters {worthy or otherwise} may appear as a STYLE-PRAGMENT-item-sequence {R9.2.1.c} except one containing the sequence {including disjunctors} which constitutes the representation of the STYLE-PRAGMENT-symbol itself {because the latter would terminate the pragment}. An implementation may, however, further restrict the sequences of characters allowed in pragmats {but not in comments}.

Four standard pragmat-items are defined: PAGE, POINT, UPPER, and RES {see *3.2.1 for PAGE and *3.5 for the rest}. All implementations must recognize these items at least in the minimal form

STYLE pragmat symbol, item, STYLE pragmat symbol.

Each of these four pragmat-items is written as a sequence of upper-case letters, and may be preceded or

followed by typographical display features. {Note that in all stropping regimes a pragmat-symbol may be written as ".PR" followed by a disjunctor.}

3.2.1 Newpage

When the base character representation of a construct is printed by an ALGOL 68 processor, a pragmat containing the pragmat-item PAGE causes the line after the line containing its closing pragmat-symbol to be printed at the top of a new page {possibly after appropriate headers}. {The PAGE pragmat is, however, not a typographical display feature.}

3.3 Typographical Display Features

The typographical display features are space, newline, and string break. {Newline may be a unique base character or a physical phenomenon like end of record. String breaks are allowed only in certain denotations; see *3.1.}

3.4 Style-TALLY Objects

No representations for any style-TALLY-letter-ABC-symbols or style-TALLY-monad-symbols {R9.4.a} are defined by this document.

3.5 Tags and Bold Words

The representation of tags and bold words is determined by the "stropping regime", of which there are three. A new regime is invoked by a pragmat containing one of the pragmat-items POINT, UPPER, or RES, and takes effect following the closing pragmat-symbol. Stropping does not affect the 'STYLE' of a representation {so in UPPER and RES, ".PR" matches "PR"}. {Some rules below require disjunctors in certain positions. If necessary, these can be obtained by inserting typographical display features.} {In ALGOL 68, tags are distinct only when the concatenations of their taggles are distinct. For example, "end of file" may also be written "endo ffile".}

{"What did the rug, dog, and fish have in common?"

"Each was a car p et."

Works, Mach Tartaruca}

{Examples are shown with each regime. A few, like ".elIF", illustrate usages that cannot be recommended. These usages are allowed because they are orthogonal and they provide a measure of tolerance to unimportant errors.}

3.5.1 POINT Stropping

Bold words.

- A bold word is written as a point (".") followed, in order, by the worthy letters or digits corresponding to the bold-faced letters or digits in the word.

- A bold word must be followed by a disjunctor.

Tags.

- A tag is written as a sequence of {one or more} taggles separated by zero or more typographical display features.

- A taggle is written by writing, in order, the corresponding worthy letters and digits optionally followed by an underscore.

- If a taggle does not end with an underscore, it must be followed by a disjunctor.

{Examples:

```
Program: .PR POINT .PR .BEGIN .REAL X; X := X-1 .END
Bold:    .BEGIN, .Real, .elIF, .X1, .abs
Plain:   BEGIN, Real, end of file, end_of_file,
     X1, a_b_, a_ b
Error:   .BEGIN_, .X_1, .end_of_file, a____b,
     a _ b, a__ b}
```

3.5.2 UPPER Stropping

Tags and bold words are represented as they are in POINT stropping with the addition of these rules:

- Upper- and lower-case letters may not be intermixed in a bold word.

- The point may be omitted from an upper-case bold word if it is preceded by a disjunctor other than a point, by a lower-case letter, or by a digit that is not an "upper-case digit". An "upper-case digit" is one that follows an

upper-case letter or an upper-case digit.

- An upper-case bold word need not be followed by a disjunctor if it is followed by a lower-case letter.

- Upper-case letters may be written only in bold words and character-glyphs {R8.1.4.1.c; these are constituents of string- and character-denotations and of pragments}.

{Examples:

Program: .PR UPPER .PR BEGIN REAL x; x := x-1 END
Bold: BEGIN, .abs, X1 {even in "a3X1"}, .a3 {even in".a3X1"}, OF {even in "reOFz"}
Plain: begin, end of file, end_of_file, a3 {even in "a3X1"}, re {even in "reOFz"}
Error: REAL_, .real_, X_ij, return_value_END

".aB" is equivalent to ".a B".}

3.5.3 RES Stropping

A "reserved word" is one of the bold words specified in R9.4.1 as a representation of some symbol. {See the list in *B. By R9.4.2.2.b, these cannot be redefined and are thus already reserved in another sense.} In the RES regime, tags and bold words are represented as they are in POINT stropping, with the addition of these rules:

- The point may be omitted from a reserved word if it is preceded by a disjunctor other than a point.

- A taggle must be adjacent to an underscore if its letters and digits correspond, in order, to those of a reserved word.

{Examples:

Program: .PR RES .PR BEGIN REAL X; X := X-1 END
Bold: BEGIN, .REAL, .X1, Begin, .operator, .AMODE
Plain: begin_, end_of_file, end_of_file_, x1, AMODE, X_1, endo_ffile, X_1
Error: .BEGIN_, .X_1}

3.6 Composite Representations

Where the representation shown in R9.4.1 appears to be composed of two or more consecutive nonletter marks {"", =:, :=, |:, :=:, :/=:}, the representation is the sequence of worthy characters corresponding to those marks.

The representation of any NOTION1-cum-NOTION2-symbol is the representation of the NOTION1-symbol followed by the representation of the NOTION2-symbol. {The NOTION1-cum-NOTION2-symbols are the composite operators mentioned in R9.4.2.2.d,e.}

3.7 Other Representations

Any symbol whose representation in the Report {R9.4} corresponds to some worthy character is represented by that character. {There are no representations for the times-ten-to-the-power-symbol, the plus-i-times-symbol, or the brief-comment-symbol, but the Report provides alternate constructs for all cases where these symbols might be used.}

4. Transput

The transput representations of objects must use only worthy characters {so that input may be prepared and output interpreted without reference to an individual implementation}. The environment enquiries {R10.2.1} depend on worthy characters as follows:

```
flip:       "T"
flop:       "F"
errorchar:  "*"
blank:      " "
```

No value is defined for "null character" by this document. Since there are no worthy characters for times-ten-to-the-power-symbol and plus-i-times-symbol, "E" and "I" must be used instead. The two cases of a letter are equivalent when they appear in the transput representation of any value other than one of mode 'character' or 'row of character'.

As a result of transput and repr, string values may contain characters that do not correspond to worthy characters. This document does not define the actions taken, if any, when such characters are transput. {Ordinarily, most such characters will simply be read and written as single characters, just as will an "A".}

{ Appendices

These appendices discuss the hardware representation, but they are not to be construed as further specification.

Appendix A. Worthy and Base Characters.

A.1 Rationale for worthy characters.

A.1.1 Specific Unworthiness

The following characters were carefully considered as candidates for worthiness, but were rejected for various reasons:

! - because it may be needed as a base character for "|"

\ - because it is not in EBCDIC and "E" is an alternative.

? - no explicit function is assigned in the Report, so it was omitted to limit the size of the worthy set.

¬ ~ - there are severe difficulties with the hardware representations of logical not and tilde: they may be printed as themselves, as each other, or as circumflex, overline, beta, or even up-arrow.

& - with no monad for not or or, ampersand was deleted to reduce the set of worthy characters.

A.1.2 Specific Worthiness

The following were considered worthy, despite disadvantages:

| - because it is crucial to ALGOL 68, despite device problems almost as severe as those for logical not and tilde.

[ ] - they are traditional ALGOL characters (but see *C.2).

% - well-defined meaning and commonly available; moreover, a short snap quiz determined that even some experts cannot remember the bold

alternatives for quotient and modulus.

@ - also well-defined and commonly available.

A.1.3 Transput Environment Enquiries

Flip and flop were chosen to be letters rather than digits because the letters have more meaning when these codes represent Boolean values. Neither a string of letters nor a string of digits is easy to read as a representation of a bits value.

The asterisk was chosen as the value of "errorchar" because question mark was unworthy and asterisk is traditional.

A.2 Relationships between Worthy and Base Characters.

An important step in developing this standard was to relate worthy characters to base characters rather than to specific hardware codes. This has several advantages:

- It avoids restricting the standard to any specific character code.
- It makes the implementer responsible for device-dependent decisions, such as the representation of vertical bar (which may be printed on various devices as any one of "|", "¦", "↑", space, ù, or ð).
- By eschewing diphthongs (e.g., "(/" for "[") it facilitates transportation by strict transliteration.
- It specifies a standard external appearance of programs rather than trying to specify a standard internal appearance.

A.2.1 Disallowed Relationships.

If this report specifies one or more representations for some symbol, an implementation should not provide any additional representation for that symbol in the following situations:

a) where there is an existing special character representation for the symbol, or

b) where the new representation would be another

bold representation for a symbol that already has a bold representation.

Situation (b) would not increase expressive power, but would increase the potential for confusion. (However, in a variant language {R1.1.5.b}, alternative bold representations might be appropriate.)

Situation (a) would introduce confusion and ambiguity in transliteration of strings. For example, if "%" and "?" both represent the percent-symbol, there is no simple transliteration for "?" in a string.

To avoid similar ambiguity and transliteration problems, implementations should not provide:

- additional style-TALLY-symbols;

- dipthongs specific to the ALGOL 68 environment.

(Thus "(/" should be neither a style-ii-sub-symbol nor a diphthong for "[".)

A.2.2 Permitted relationships.

If system software commonly uses a diphthong for some representation -- such as the diphthong proposed for colon on some systems -- an ALGOL 68 compiler may have no choice but to accept it as a single character. No problem arises as long as the substitution is universal and unambiguous inside and outside strings.

An implementation may specify two or more separate base characters to represent some one worthy character. This may be necesssary, for example, if some device lacks "|" and "!" is to be allowed in its stead. The two base characters should be treated as equivalent everywhere except within strings and on program listings, where each should represent itself. When a program is transported it may be necessary to transliterate both base characters to one new character.

Difficulty arises only when trying to export a program that has attempted to utilize the distinction between the two characters. Such a program is not a portable program.

A.3 Super-set Character Sets.

A.3.1 Escape Character.

Some implementations have defined an escape convention for representing extra string-items. This standard does not prescribe any such convention but, if one is used, the apostrophe should be the escape character.

A.3.2 Admissibility of Other Characters.

After adapting the local characters to the worthy characters, an implementer may find he has "unused base characters" that do not map to worthy characters. For each such character C the implementer may choose from the following interpretations:

a) Unused. C is erroneous except possibly inside pragments.

b) As in the Report. If C appears as a representation for some symbol S in the Report and there is no worthy representation for S, then C - if allowed at all - should be a representation for S. Thus, "\", "$_{10}$", "$\cdot$", "$\circ$", "$\not\in$", and "&", "$\neg$", "~", "$\uparrow$", and the other unworthy operators in R9.4.1.c may be used only to represent themselves (unless a desperately small character set forces their use as worthy characters).

c) An unworthy representation. C may represent some symbol for which no nonletter worthy representation is given. For example, "?" could be a skip-symbol.

d) Style-TALLY-monad-symbol. For example, if "?" were not used as an unworthy representation as in (c), it could be a monad. If this option is chosen, C should look like an operator. For example, "{" might make a poor monad.

e) Style-TALLY-letter-ABC-symbol. Care should be taken that C look somewhat like a letter rather than an operator.

f) A typographical display feature. Such an additional feature should usually be ignored in strings (unlike space).

In addition to one of the above, C may be permitted

as an other-string-item.

Appendix B. Bold Symbols and Plain Tags.

B.1 Goals of Stropping Rules.

In addition to the goals listed in *Ø, the design of the representations for bold symbols and plain tags was motivated by the following criteria.

a) There should be a small number of stropping regimes to minimize the size of token scanners.

b) For compatibility with North American expectations, at least one regime must be some form of reserved words.

c) Numerous fortunate installations have two cases and desire some form of case stropping.

d) For the sake of tradition, the standard must include at least one regime where all bold words must be stropped.

e) The standard should reduce the possibility of error and enhance the probability of detecting those errors which do occur.

f) Some means of explicit stropping should apply in all stropping regimes so that, among other reasons, pragmat-symbols may be written in a regime-independent manner.

g) Because it is allowed by the Report, there must be some way to represent a tag or taggle that has exactly the same letters as a reserved word.

B.2 List of Reserved Words.

In the RES regime, all bold words listed in R9.4.1 are reserved. There are sixty-one:

at, begin, bits, bool, by, bytes, case, channel, char, co, comment, compl, do, elif, else, empty, end, esac, exit, false, fi, file, flex, for, format, from, go, goto, heap, if, in, int, is, isnt, loc, long, mode, nil, od, of, op, ouse, out, par, pr, pragmat, prio, proc, real, ref, sema, short, skip, string, struct, then, to, true, union, void, while.

Additional bold words may appear in section 9.4.1 of a document defining a superlanguage {R2.2.2.c} or variant {R1.1.5.b} of ALGOL 68. These words should be reserved in an implementation of the modified language. (Programs using them are not very portable anyway.) If a modified language does not give a meaning to some word in the above list, it should nonetheless remain reserved. Only thus can users of a sublanguage be assured of compatibility with implementations of the full language.

B.3 Other Stropping Regimes.

For compatibility with existing installation practice, implementations may implement stropping regimes in addition to those provided by the standard. However, such additional regimes should be invoked by pragmat-items distinct from those in *3.5. All modifications to the defined regimes -- including extensions -- should be avoided because they would inhibit error detection and decrease portability.

B.4 Inside Pragmats and Strings.

To simulate stropping and taggle concatenation, points and underscores may appear in pragments and strings. This may improve the readability of pragments by distinguishing between natural language words and those from ALGOL 68. However, when appearing as string- or comment-items, points and underscores represent themselves and do not indicate stropping.

B.5 Classification of Points.

The following properties of points hold in correct programs. Implementers may find them convenient.

a) Inside a format-text {10.3.4.1.1.a}, but outside any constituent unit or enclosed-clause, a point is a strop if and only if it is followed, first, by one of "co", "pr", "comment", or "pragmat", and next by a disjunctor.

b) A point is not a strop if it is a character-glyph {R8.1.4.1.b}. {Inside a pragmat an implementation may treat a point as a strop.}

c) Elsewhere a point is a strop if it is followed by a letter.

d) A stropped word is always bold.

Appendix C. Portable Programming.

Appendices *A and *B provide considerable latitude for extension of this standard in response to local conditions; however, no implementation will have all these extensions. This appendix discusses the maximum facilities that may be safely employed in a portable program.

C.1 Character Set Descriptions.

The standard is defined in terms of worthy characters in order that program conversion will require only a transliteration of character codes. To facilitate the debugging of such a routine, a program publisher should provide with published programs a file containing the following:

- one or more lines, as necessary, containing all the characters used in the program. This should begin with all of the worthy characters, in the order in which they appear in *1;

- a description of each character.

Each implementer should provide such a file describing the implemented character set.

C.2 Sub- and Bus-symbols.

Nonstandard implementations sometimes restrict the representations for sub- and bus-symbols. For a portable program, two schemes are possible.

a) Use only square brackets. This scheme is preferable because it is the one most likely to be widely portable. Note that every implementation is required to provide base characters for the square brackets, even though the characters provided may not resemble brackets.

b) Use parentheses, but follow this restriction: No local-sample-generator {R5.2.3.1.b} may begin with a style-i-sub-symbol. {This can always be achieved by inserting a local-symbol.} {Any sublanguage with this restriction is easier to parse.}

All implementations of this report will perforce accept programs written according to both of the above schemes.

C.3 UPPER Case.

Some implementations will be unable to support two alphabetic cases. Users with such implementations can usually import programs by converting all the letters to the single case; this succeeds because the standard specifies that both cases of a letter are equivalent in all but two contexts. The first such context is strings; however, as long as the string is intended only for printing, little damage will be caused by converting its letters to a single case. Programmers should be wary of any program whose correct execution depends on the fact that there are two cases of letters in a string.

The second context where case distinction is allowed is in UPPER stropping. A program so stropped is readily converted to POINT stropping, if every bold word is preceded by a blank and followed by a disjunctor. At its simplest the conversion changes "blank, upper-case-letter" to "point, letter", but this may unduly modify the contents of strings. With more complex logic, even programs without blanks before UPPER-stropped bold words can be translated to some other stropping regime, by the recipient. There is, however, the risk that the line length may be increased by the insertion of stropping points or extra disjunctors. It is possible that this may require that some lines be broken if the receiving installation imposes a maximum line length.

C.4 Newlines in Strings.

Some software environments routinely strip trailing blanks from the end of each record; others pad all records to a fixed length; others perform curious mixtures of these procedures. In either case, the number of blanks in a transported string may change if the string includes a newline. To avoid such changes, newlines in strings should appear only in string breaks.

C.5 Other Characters.

A portable program should be written entirely in worthy characters, because only these characters are available in all implementations. With care, however, it is occasionally permissible to use unworthy characters. For example, unworthy characters can be used in messages intended solely for output. Transliteration of such a character may hinder interpretation of the output, but it will not otherwise affect execution of the program. In particular, "?" and "&" are available in most character sets, so they will cause little difficulty if used within strings.

In any case, if unworthy characters are used, sufficient explanation must be provided to enable correct adaptation of the program to a new character set.

C.6 Character Code Dependence.

Use of repr should be severely restricted. Programs should not depend on the particular character code used by the implementation. This can be accomplished with cautious use of the environment enquiry abs. For example, an array, "char type", to be used to distinguish between letters, digits, and all other characters, could be defined and initialized as follows:

```
[0 : max abs char] int char type;
int kletter = 1, kdigit = 2, kother = 0;
for i from 0 to max abs char
do
   char type[i] := kother
od;
for i to 10
do
   char type[abs "0123456789"[i]] := kdigit
od;
for i to 52
do
   char type[abs "abcdefghijklmnopqrstuvwxyz"
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i]]
      := kletter
od
```

{This succeeds even if the receiving installation lacks lower case, because the lower-case letters will have been translated to upper case.}

C.7 Portability of Compiler Character Codes.

Four worthy characters -- "|", "_", "[", and "]" -- are often coded differently, even at installations which nominally use the same character code. Implementors should consider whether to provide means enabling each installation to choose codes for these characters for use in error messages, machine-readable documentation, programs, and normal transput.

C.8 Reserved Words.

Although not allowed by this report, some implementations may have reserved word lists that differ from the list in *B. A portable program using RES stropping should ignore the local list by explicitly stropping words not on the official list and placing underscores adjacent to plain taggles that appear on the list.

C.9 Minimum Form Standard Pragmats.

Because some implementations may have special syntax for pragmats, portable programs should employ only minimum form pragmats:

pragmat-symbol, standard-item, pragmat-symbol.

where "standard-item" is PAGE, RES, UPPER, or POINT. Implementers should provide PRAGMATS OFF {R9.2} (and perhaps PRAGMATS ON) to control interpretation of pragmats.

C.10 "PORTCHECK" Option.

Despite good intentions, a programmer may violate portability rules by inadvertently employing a local extension. To guard against this, each implementation should provide a PORTCHECK pragmat option. While this option is in force, the compiler prints a message for each construct that it recognizes as violating some portability constraint.

}