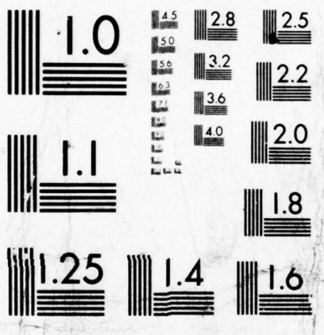
AD-A049 572 ROYAL RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
INTRODUCTION TO THE 'RS' PORTABLE ALGOL 68 COMPILER.(U)
AUG 77 S G BOND, P M WOODWARD

UNCLASSIFIED RRE-TN-802 DRIC-BR-59078 NL

| OF |

AD
A049572

END
DATE
FILMED
3-78
DDC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

1

NW

RRE TECH NOTE No. 802 INTRODUCTION TO THE 'RS' PORTABLE COMPILER

RRE

TECHNICAL NOTE

No. 802

INTRODUCTION TO THE 'RS' PORTABLE COMPILER

Authors: S G Bond and P M Woodward

Royal Radar Establishment,
Procurement Executive,
Ministry of Defence,
Malvern, Worcs.

August 1977

Unclassified

(14) RRE-TN-8Ø2

(18) DRIC

(19) BR-59Ø78

(9) RSRE Technical Note 802

(6) INTRODUCTION TO THE 'RS' PORTABLE ALGOL 68 COMPILER.

(10) by S. G. Bond [redacted] P. M. Woodward

(11) Aug 77

(12) 31p.

SUMMARY

This paper describes the portable compiler for Algol 68 developed at RSRE by Currie and Morison. Chapter 2 defines the language extensions handled by the compiler, and the system of modular compilation as seen by the user. Chapter 3 outlines the structure of a complete system, emphasizing the design of the intermediate language produced by the compiler for input to a machine-dependent translator. Though factual, this paper is not the system documentation.

CONTENTS

311750 [illegible]

1 INTRODUCTION

The aim of a portable compiler is to reduce the cost of implementing the same language on different machines. It should also reduce the time taken to shift the load of a computing service on to new hardware. This is the credit side of portability. On the debit side, the product can be inefficient in use, for no single compiler can generate a universal machine code adaptable by some simple transformation to any type of machine. Traditionally, the compromise - if attempted at all - seems to have favoured portability at the expense of a really good match to the hardware. The 'RS' compiler designed and written by I F Currie and J D Morison, and originally described by Currie (ref 1) in 1976, shifts the balance in favour of efficiency. This compiler tries to make no assumptions at all about the machine for which code is to be produced, confining its activities to things which must be done for *any* machine, such as syntax analysis of the source text. The output from the compiler, known as stream language, is far removed from 'code', and being at a comparatively high level leaves a fair amount of work undone. The implementor must write a second pass - a translator - which will take complete responsibility for code production, including representations of values and storage allocation.

Potential users are likely to ask two questions. *Is the language true Algol 68? Just how much work is required to complete a system?* Although no attempt is made in this paper to catalogue unimportant deviations from Algol 68, chapter 2 should give an idea of the language and its extensions, such as those which deal with independent compilation of program modules. The amount of space devoted to this topic is indicative of the importance we attach to the subject. As answer to the second question, chapter 3 is intended to give enough information for experienced system programmers to judge for themselves what effort is likely to be required. At the time of writing, there is little factual data, for three implementations are in progress but none is yet complete.

The present paper is not the RS system documentation required by translator writers, though it may prove useful as an introduction to such documentation.

Important note

The information given in this paper must not be taken as definitive for the 'RS' system or for the language it accepts.

2 THE SOURCE LANGUAGE

The language accepted by the compiler is Algol 68 as defined in the Revised Report (ref 2) with some deviations, principally that modes and identifiers must be declared before they are used. Arrays are not copied in identity declarations and in the corresponding parameter situations. The handling of flexible arrays differs from the report in that flexibility propagates through a mode to the right (ie inwards). The use of transient references is not checked. These are the main deviations. There are also some significant extensions to Algol 68 which influence the design of a translator. Two new types of data structure have been added to the language, mainly to increase efficiency in critical applications such as data processing, compiler writing and translator writing. Other extensions have been introduced in the light of experience to provide flexibility in system programming work. Most extensions can be concealed from the ordinary user by restricting the generally available documentation. But they cannot be concealed from the translator writer, who may wish to exploit them and must, in any case, be able to translate their stream language images into machine code. The various extensions are described in 2.1.

The unit of compilation in the RS system is known as a "module". A program can be assembled from any number of interacting modules with full mode checking across their boundaries, and as this is an extremely important part of any compiling system, it is defined at some length in section 2.2.

2.1 EXTENSIONS TO ALGOL 68

2.1.1 Vectors and indexable structures

A *vector* is a one-dimensional array with an understood lower bound of 1. A typical declaration would be

```
VECTOR [n] INT v;
```

where the size n can be any unitary clause. A vector can be flexible or not, and subscripted and trimmed like an array, though the use of "AT" is forbidden. In strong contexts, a single object can be 'rowed' to a vector. The overheads associated with vectors are smaller than for arrays, and assignment of vectors is simpler than for arrays because the elements are always contiguous.

The *indexable structure* or more briefly "i-struct" represents the ultimate step in removing array overheads whilst preserving the facility of indexing. It groups together a fixed number of objects of any specified mode; for example, a STRUCT 30 REAL consists of 30 reals, and the size 30 is *part of the mode*. The size must therefore be an integer denotation. An i-struct can be indexed with the same notation as for an array, and the indexing starts at 1. If trimmed, it gives rise to a vector (though as with vectors, the "AT" construction is prohibited). In strong contexts, a single object can be 'rowed' to an i-struct. The i-struct enables fixed length rows of characters to be handled with the efficiency expected for Algol 68 BYTES, LONG BYTES etc, but without any restrictions on length.

Coercions on i-structs and vectors are all in the direction *i-struct to vector to array*. All such coercions (including ref i-struct to ref vector etc) are allowed before uniting. However, i-structs, vectors and arrays of the same mode can exist side by side in the same union, and any seeming ambiguity when uniting is avoided by preference for minimum travel in the "i-struct to vector to array" direction. The same preference rule applies to operator selection, as shown in the following example:

```
OP £ = (VECTOR [ ] REAL p) ... ... ;

OP £ = ([ ] REAL p) ... ... ;
```

With these two declarations in force, an actual operand of mode STRUCT 4 REAL would be coerced to VECTOR [4] REAL and the first operator definition would be selected.

String denotations are i-structs (eg "ABC" is STRUCT 3 CHAR), but the above coercions ensure that users wishing to avoid the language extensions need not be aware of them. The word VECTOR will not appear in compile-time diagnostic messages unless it has already been explicitly used in the source text, and messages concerning strings like "ABC" will typically be described as "3 CHAR" rather than "STRUCT 3 CHAR".

2.1.2 The FORALL statement

The FORALL statement has been introduced for efficiency in sequencing through all the elements in one dimension of an array, or all the elements of a vector. As an example, in the unitary clause

```
FORALL xi IN x DO xi TIMESAB xi OD
```

the new identifier xi (declared by FORALL) successively takes each of the values x[i] with i going FROM LWB x TO UPB x. The effect of this example, therefore, is to square all the elements of x. It avoids explicit indexing and the associated overheads in the compiled code. There can be a sequence of parts like "xi IN x" provided each has the same number of elements. For example,

```
VECTOR [10] INT v;
[3:12, n:m] REAL w;
FORALL elemv IN v,
          elemw IN w
DO f(elemv, elemw) OD
```

applies the function f to all pairs of arguments (v[i], w[i+2,]) for i "FROM 1 TO 10".

A FORALL statement can have a while part, and the range of the identifiers declared by FORALL (eg elemv, elemw) is the WHILE clause and the DO clause.

Primarily for use in conjunction with the FORALL statement, a new dyadic operator, CYCLE, is defined to act on multi-dimensional arrays. The expression n CYCLE w delivers the array w with a new descriptor, in which the dimensions are cycled to bring the (n+1)th to the front.

2.1.3 Straightening

A "straightening" facility is provided to enable Algol 68 programmers to write transput procedures with arbitrarily structured parameters. Straightening is the reduction of any type of data structure to a simple sequence - which we shall describe as a "straight". The basic step is the coercion of a simple row or structure to a straight; applied recursively, the method can be used to straighten data structures of arbitrary complexity.

The mode STRAIGHT U, where U is any Algol 68 mode (but most commonly a union), describes a set of objects of mode U. In this respect it is similar to [ ]U, but in other respects it is quite different and must be treated as a new type of mode. An actual straight is brought into existence by strong coercion of a row, vector, structure, i-struct or union. Such modes are strongly coercible to STRAIGHT U if their "members" can be coerced to U by uniting, or by straightening or any of the coercions i-struct to vector to array (2.1.1). The coercions excluded are dereferencing, deproceduring, widening and rowing.

Example 1

```
        STRAIGHT UNION(INT, CHAR) s1 = "ABCD"
```

As CHAR is coercible to UNION(INT, CHAR), the i-struct "ABCD" can be coerced to the STRAIGHT. If s1 were the formal parameter of an output procedure, acceptable actuals would be a row of characters, row of integers, structure with integer and character fields or a union of integer and character. However, a single INT or a single CHAR would *not* be accepted.

Example 2

```
        STRUCT(INT i, REAL r) p;
        STRAIGHT UNION(REF REAL, REF INT, REF CHAR) s2 = p
```

The members of p have modes REF INT and REF REAL, both of which are coercible to the given union, so p will be coercible to the mode of s2. Clearly, s2 might be the formal parameter of an input procedure and p its actual parameter. The actual could not be a *simple* real, integer or character variable.

Example 3

```
        [1:3]INT v := (1, 2, 3);
        STRAIGHT INT s = v;
```

In this example, the members of the variable v have mode REF INT, but s is a straight of plain integers. As it stands, v cannot be straightened to s because dereferencing of members is not allowed. But as v can be dereferenced before straightening, the example is correct. Considered as a formal parameter for an output procedure, s would handle any row or structure of integers, but not a single integer by itself.

As a straight cannot represent an unstructured value, most applications will demand that it be combined with basic modes in a union, eg

UNION(INT, REAL, ... , STRAIGHT UNION(INT, REAL, ...))

This mode will handle an object of data which possesses structure at no level (eg an INT) or one level (eg [ ]INT) but not more. When an object is being united to the above mode, then - regardless of the order in which the constituent modes have been written - the fit will be sought from the non-STRAIGHT modes first, so as to avoid any possible ambiguities of coercion.

To handle one object structured at any number of levels, a recursive mode is needed.

MODE PRINTMODE = UNION(INT, REAL, ... , STRAIGHT PRINTMODE)

The definition of STRAIGHT is such as permits this recursion. PRINTMODE will handle an integer, real, etc, or any row or structure built up from all these to any depth. For a corresponding input parameter mode, the basic modes would each be preceded by a REF.

The parameter of the standard "print" procedure has mode

[ ] PRINTMODE

rather than PRINTMODE. This allows the use of a collateral as the actual parameter.

A straight cannot be handled with the full generality applicable to other Algol 68 modes. The manipulations are confined to subscripting and interrogation by the operator UPB. Let M stand for any mode, and let s have mode STRAIGHT M. Then UPB s gives the number of objects in the straight, and s[i] picks out the ith object (i>=1). There is no such thing as a STRAIGHT generator or variable, ie objects of mode REF STRAIGHT do not exist - except for the possibility of NIL.

2.1.4 Low level facilities

The monadic operator SPELL takes an operand of any simple mode and delivers a vector of characters having the same machine representation. If the mode of the operand was a reference, the operator delivers a REF VECTOR[ ]CHAR. For the purpose of this definition, a "simple" mode is one which contains no vectors or arrays or which is a vector of such objects.

Code can be inserted in an Algol 68 program by the construction

mode CODE (unc , unc , ...) " code "

which is treated as a primary of the specified mode (absence of which implies mode VOID).* The unitary clauses, to which no coercions are applied, supply

* Here, and elsewhere, underlined words are symbolic, and broken underlining indicates an optional item.

Algol 68 objects for use in the code. Other alien insertions, such as non-Algol procedures, must take the form

mode identifier = ALIEN " insertion "

ALIEN is allowed only in this identity declaration context.

An alternative method of expressing a string denotation is provided. This uses the ABS values of the characters rather than the characters themselves - which might be non-printing characters. The ABS values can be to radix 2, 4, 8, 10 or 16, and must be separated by spaces; the string must be preceded by 10r or 16r or whatever the case may be. Thus the following 3-character strings (or more strictly STRUCT 3 CHAR's) are equivalent: 8r "1 15 251", 16r "1 d a9", where a-f represent the digits 10-15.

2.2 MODULES - THE UNITS OF COMPILATION

A program can be compiled in portions known as modules, of which there are three different types. The basic type is the closed clause or *cc module* which consists of an Algol 68 closed clause with a suitable heading and the word FINISH. This could be a complete program or one of a number of cc modules which are to be nested one within another. In the actual Algol text of a cc module, any place at which some inner module is later to be inserted is marked by a new type of unitary clause known as a "here-clause".

A nest of modules will be described as a 'composition'. The selection and placement of modules to make a composition is specified in a *composition module*, which contains no Algol 68 text of its own. A composition need not be completed all at once. A program can be partially composed in one composition module, leaving spaces for further cc modules to be inserted later on by another composition module.

A third type of module, the *declarations module*, enables modes, procedures and other items to be declared and compiled in advance of their use in other modules. Declarations modules are used in a very straightforward way requiring no composition, but they can never make a program by themselves. To make this distinction clear, the other types of module (cc and composition) will be described as *program modules*.

2.2.1 Keeplists

Interaction between modules demands that source-text indicators (identifiers, mode names and operators) declared in one module shall be usable with the same meanings in another module. The source-text of a module must always specify which of its indicators are to be *kept* after compilation for use in modules to be compiled later. A keeplist is a sequence of such indicators, separated by commas. To distinguish between versions of operators, the modes of operands must always be included, as in the keeplist here:

MAN, WOMAN, = (MAN, WOMAN), = (WOMAN, MAN), adam, eve

The order of the items in a keeplist is never significant.

2.2.2 Simple declarations modules

The sole purpose of a declarations module is to make declared items available for use in other modules. Consequently, a declarations module must invariably have a keeplist for such items, and if it uses *no* indicators from other modules itself (other than from automatically incorporated library modules), its form is

```
DECS decstitle
body
KEEP keeplist
FINISH
```

In the first line, decstitle stands for an identifier chosen to be the title of the module. The *body* (which is not enclosed in BEGIN-END brackets) consists of Algol 68 declarations, and other phrases which may be convenient for setting things up. Certain restrictions are enforced to ensure that declarations modules can be obeyed in any order without giving rise to side-effects. Thus no procedures and no user-defined operators may be called, except inside procedure or operator declarations. These are the only restrictions for a self-sufficient declarations module, but we must now turn attention to a more general class of module which has an added restriction.

A DECS module can use indicators kept from previously compiled DECS modules. There are two requirements for the passage of an item from one module to another. Its indicator must be included in the keeplist of the source module, and the title of that module must be included in the heading of the using module, as shown in the second line below

```
DECS decstitle
USE decstitlelist
body
KEEP keeplist
FINISH
```

The decstitlelist is simply a list of the titles of all the other DECS modules required, separated by commas. The *body* can now use kept items from these modules, with one further restriction to ensure complete absence of side-effects. No kept item which is a reference (or a structure, array or union containing a reference) may be used, except within a procedure or operator declaration. These restrictions on the use of external references and calls of procedures or user-defined operators are peculiar to DECS modules and free the user from having to consider at what stage his DECS modules are actually obeyed.

2.2.3 Simple programs

Simple programs will usually consist of one closed clause module, possibly supported by previously compiled declarations modules. Using square brackets to indicate this option, the form in which the cc module is written is:

```
 PROGRAM progtitle
[ USE decstitlelist ]
 closed clause
 FINISH
```

2.2.4 Nested modules

Within any program module, a place can be held for a separately compiled program module to be inserted later. This is done by the new unitary clause

```
            HERE place (keeplist)
```

where place stands for some identifier to name the place, and the keeplist contains any indicators currently in scope which are to be kept for the use of the inserted program module. If there are several HERE clauses in the same module, the place identifiers must all be distinct.

The form of a cc module which contains HERE-clauses is similar to that of the simple program shown in 2.2.3, except that each place defined in a HERE-clause must also be listed in the module heading before the title, ie

```
            PROGRAM (placelist) progtitle
          [ USE decstitlelist ]
            closed clause including HERE-clauses
            FINISH
```

The places in the placelist are listed, with comma separation, in any order.

A simple cc module suitable for insertion at a given place would be

```
            PROGRAM title
          [ CONTEXT place IN progtitle ]
            closed clause
            FINISH
```

The CONTEXT part of the heading, if present, makes the keeplist at the given place accessible in the closed clause. It also prevents the module being used in any other context. (By contrast, a module with no context specification could be inserted at any place, but would be denied access to the associated keeplist. This may seem a pointless construction, but a realistic example of its use is given in section 2.2.7.)

Example of nesting

```
            PROGRAM (detail) frame
            BEGIN MODE FORM = ... ;
                  OP £ = (FORM f)INT: ... ;
                  FORM f1, f2, g1, g2;
                  ---
                  HERE detail(FORM, £(FORM), g2);
                  ---
            END FINISH
```

```
PROGRAM insert
CONTEXT detail IN frame
BEGIN FORM f := g2;
      INT n := ff;
      ---
END FINISH
```

Although "insert" is compiled in the context of the first module so as to pick up its kept indicators, it remains a separate module. A program combining the two modules has to be expressed as a composition module,

```
PROGRAM whole
COMPOSE frame(detail = insert)
FINISH
```

2.2.5 Composition

The purpose of a composition module is to assemble a nest of modules by pairing up formal place names (- the ones in the Algol 68 HERE clauses -) with actual names of program modules.

The form of module which completes a nesting, inwards from some given staring module x, say, is

```
PROGRAM progtitle
COMPOSE nest
FINISH
```

where progtitle is a new identifier to act as the title of the composition, and nest starts with the title, x, of the starting module, continuing with a bracketed list of substitutions having a place on the left and a (program) module name or a further nest on the right.

Example

Given a program module starting

```
PROGRAM (x1, x2) x
```

and a set of inner modules with the headings

```
PROGRAM a
CONTEXT x1 IN x

PROGRAM (b1, b2, b3) b
CONTEXT x2 IN x

PROGRAM (c1) c
CONTEXT b1 IN b

PROGRAM d
CONTEXT b2 IN b

PROGRAM e
CONTEXT b3 IN b

PROGRAM f
CONTEXT c1 IN c
```

the following composition module combines them all into one:

```
PROGRAM compo
COMPOSE x(x1 = a,
          x2 = b(b1 = c(c1 = f),
                 b2 = d,
                 b3 = e))
FINISH
```

This composition module may still not be a complete runnable program, for x may specify some context. If so, it will obviously apply to "compo" as well. Composition modules cannot have context specifications in their headings; the context which applies to such a module is always that specified in its outermost cc module.

2.2.6 Partial composition

A composition module may leave some places to be filled by other program modules in a further composition later. It does this by pairing a place name with a new place name of its own instead of an actual program module title. A new place name in a composition module is introduced by the word HERE, even though it is not in an Algol text setting. As an example, let us omit module c from the composition given above, and make the partial composition

```
PROGRAM (hole) p
COMPOSE x(x1 = a,
          x2 = b(b1 = HERE hole,
                 b2 = d,
                 b3 = e))
FINISH
```

Observe that there is no explicit keeplist at "HERE" in a partial composition. *The available indicators are all those kept en route from the outermost module to the word HERE in the composition.* Thus, any module now compiled at

```
CONTEXT hole IN p
```

has available to it all the indicators kept at x2 in x, as well as those at b1 in b. Combination of keeplists is the main purpose of partial composition, enabling programs to exploit several "environmental packages" simultaneously, as we shall now see.

2.2.7 Use of environmental packages

Many packages (eg for simulation or graph-plotting), besides declaring modes and procedures, have to set up some starting position before the user's program is obeyed, and tidy up afterwards (eg close files which were opened at the start). The basis of any such package must be a cc module with a "HERE" for the rest of the program.

For instance,

```
        PROGRAM (userprog)package1
        BEGIN
             --- ;
             --- ;
             HERE userprog(keeplist1);
             --- ;
             ---
        END
        FINISH

        PROGRAM my prog
        CONTEXT userprog IN package1
        closed clause FINISH
```

with

```
        PROGRAM runner
        COMPOSE package1(userprog = myprog)
        FINISH
```

as the composition.

Now consider writing a program, "our prog" say, which requires the services of *two* packages, designed independently along the lines of package1. The question will be, in what context to compile "our prog"? It cannot be userprog IN package1, which brings in only keeplist1, nor can it be userprog IN package2 for a similar reason. The only answer is a context like "user IN both", set up specially by the partial composition:

```
        PROGRAM (user)both
        COMPOSE package1(userprog = package2(userprog = HERE user))
        FINISH
```

The context "user IN both" combines the keeplists of both packages, as explained in 2.2.6. Before leaving this example, it is worth remarking that "package2" fits at "userprog IN package1" because package2 specifies no particular context. Being an independent package, it needs no access to package1's keeplist.

2.2.8 Declarations modules in a context

DECS modules, like cc modules, can specify a context in their heading:

```
        DECS decstitle
        CONTEXT place IN progtitle
        body using kepts at above place
        FINISH
```

The CONTEXT line makes the kepts at place IN progtitle accessible for use in the body of the DECS module, with the same restrictions as given in 2.2.2 earlier. No kept item which is a reference (or a structure, array or union containing a reference) may be used except within procedure or operator declarations. And as with all DECS modules, there can be no procedure calls, or calls of user-defined operators.

Any module which has access to the same kepts (ie those at <u>place</u> IN <u>progtitle</u>) can USE this declarations module. The context specified by the using module must therefore be the same as that of the DECS module or be a dependent context resulting from partial composition - which, by the combination rule, would supply the same kepts and more besides. To see this clearly, consider once more the composition

```
        PROGRAM (hole) p
        COMPOSE x(x1 = a,
                  x2 = b(b1 = HERE hole,
                         b2 = d,
                         b3 = e))
        FINISH
```

The context "hole IN p" is derived from "b1 IN b" which is in turn derived from "x2 IN x". It follows that any module specifying "CONTEXT hole IN p" can USE declaration modules specifying one of

```
        hole IN p,
        b1 IN b,
        x2 IN x,
```

or, of course, no context at all.

2.2.9 Provision for Algol 68 standard environment

Any cc module or declarations module having no explicit context specification in its heading is assumed by the compiler to have specified a standard default context. For descriptive purposes only, we shall refer to this as

```
        CONTEXT %program IN %mkxprelude
```

Thus, a program which appears to be complete, such as

        PROGRAM pmw *closed clause* FINISH

can only run when nested in %mkxprelude. The intention is that the necessary composition should be effected automatically. The %mkxprelude will go some part of the way towards providing the Algol 68 standard environment, and will do so without any action by the user. (Its kepts are accessible to all modules without need for partial composition, see 2.2.10.)

The remainder of the standard environment will be provided by library DECS modules. With the cooperation of its shell, the compiler will supply a default USE for any library declarations modules required in a program.

2.2.10 The void context

A truly outermost cc module specifies CONTEXT VOID and is, by that token, a *prelude*. (Absence of any context specification, as we have already seen, does *not* imply a void context.) The compiler treats preludes in a special way. For simplicity of explanation, it will be assumed that a prelude has only one HERE-clause.

The first special property of a prelude is that it provides what may be described as a "universal context" for composition purposes. A cc module which specifies a prelude context (explicitly or by default) can be inserted directly in the prelude *or in any dependent context*. An example of this is to be found at the end of 2.2.7 (environmental packages).

The second property of a prelude is that its kepts are universal. Its own keeplist and those of any DECS modules compiled at that context are freely accessible to all dependent modules. This is shown below for a chain of cc modules.

module	*can access cc keeplists from*
PROGRAM (prog) ownprelude CONTEXT VOID *closed clause* FINISH	*nowhere*
PROGRAM (a1) a CONTEXT prog IN ownprelude *closed clause* FINISH	prog IN ownprelude
PROGRAM (b1) b CONTEXT a1 IN a *closed clause* FINISH	prog IN ownprelude a1 IN a
PROGRAM c CONTEXT b1 IN b *closed clause* FINISH	prog IN ownprelude b1 IN b

The last of these modules, c, shows the accessible kepts to be those from the immediately surrounding context and the outermost one. The keeplist at "a1 IN a" is *not* accessible.

Declarations modules, like cc modules, can be compiled at CONTEXT VOID. *Any* module can USE such a DECS module. This is a consequence of the general rule given at the beginning of p 13, and is in fact the limiting case of it.

Finally, the context for a composition will be VOID if the composition starts from a prelude. This means that systems programmers will be able to extend %mkxprelude if they wish, without losing any of its special properties.

2.2.11 Summary of syntax and semantics of modules

Square brackets enclose optional parts

DECS module

DECS decstitle

[CONTEXT place IN progtitle]*

[USE decstitlelist]

body

KEEP keeplist

FINISH

Except within procedure or operator declarations, the *body* must not use any externally declared references or call any procedures or user-defined operators.

PROGRAM modules

PROGRAM [(placelist)] progtitle

[CONTEXT place IN progtitle]*

[USE decstitlelist]

closed clause

FINISH

The above is a closed clause or cc module. The closed clause can include "here-clauses" of the form HERE place(keeplist). This makes a hole which can be filled by another cc module, as specified in a composition module:

PROGRAM [(placelist)] progtitle

COMPOSE nest

FINISH

Notes

decstitle, place, progtitle all stand for identifiers. An itemlist is a sequence of items with comma separation. For the definition of a keeplist, see 2.2.1. For definition of nest, see 2.2.5.

* Omission of an explicit context introduces the default context:
CONTEXT %program IN %mkxprelude
For absolutely no context at all, CONTEXT VOID must be written.

Composition rules

A program module can be composed at p IN q if its context specification - explicitly or by default - is one of the following three possibilities:

p IN q,

the prelude context from which q is derived,

VOID (applicable only to prelude writers)

The context specification of a composition module is that of its starting module.

Accessibility of kepts for use in a cc module

If the context specification is CONTEXT p IN q, the cc module can use kept indicators from

p IN q, and any DECS compiled at p IN q;

if q is a partial composition, from any hierarchical context embracing p, and any DECS compiled at any of those contexts;

the prelude context of q, and any DECS compiled there;

any DECS compiled at context VOID.

Any DECS modules required must be mentioned in the heading of the using module (at "USE decstitlelist"), unless they are library DECS which may be incorporated in the final program automatically as needed.

Accessibility of kepts for use in a DECS module

The sources are the same as for cc modules, but any kept references are debarred from use in the body of the DECS module except within procedure or operator declarations. This restriction extends to objects such as structures, arrays and unions containing references.

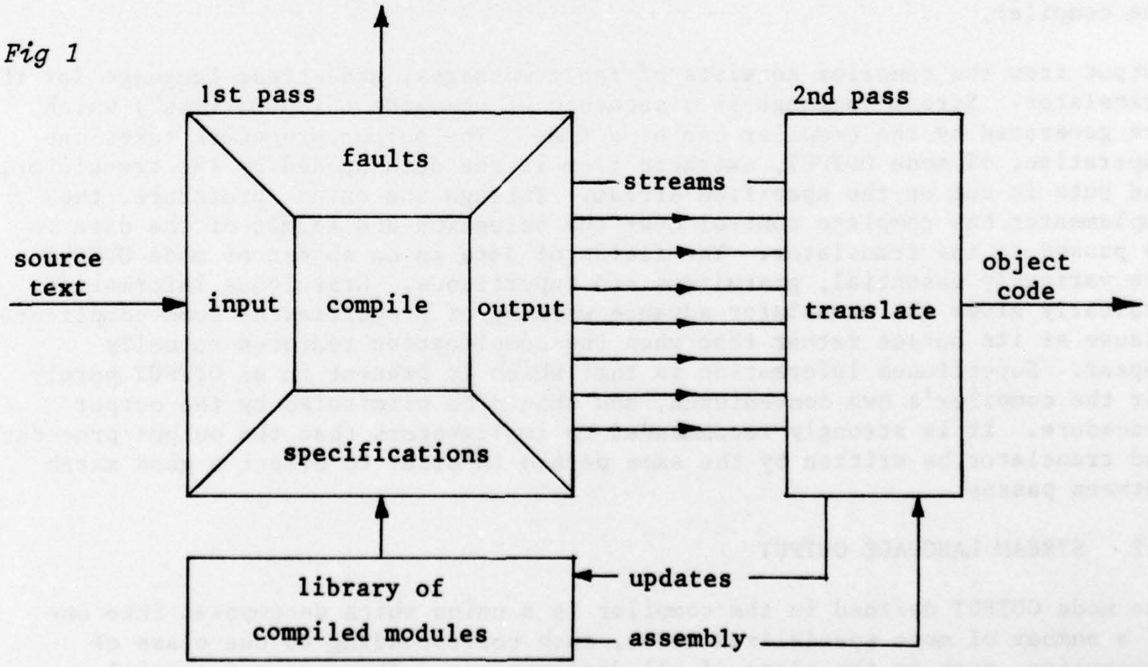
3 THE COMPILER IN CONTEXT

The RS compiler differs from most others in producing output which bears very little resemblance to machine code. The structure of the output is close to that of Algol 68 in many respects, and yet the work done by the compiler is not insubstantial. It checks the correctness of the source text, as far as this is possible by syntax and mode analysis. If an error is found, it outputs the diagnosis; otherwise the *information* in the source program is recast in a form suitable for translation. Complicated operations are broken down into sequences of the simpler steps adjudged primitive for the purpose of code generation. For example, as the modes of all objects in the source program have been determined by the compiler, it can specify every coercion explicitly. The coercions will in fact make their appearance to the translator at the precise moments required, even though the compiler may have had to see much farther ahead in the program to determine the destination mode. This is one of the fruits of the technique of using the output from the compiler as a buffer for the re-ordering of information. The compiler puts its output on one or another of several parallel streams, and arranges that the item immediately required by the translator is always at the reading point on one of the streams. This technique, reported by Currie (ref 1) to be widely used by compiler writers at the University of Grenoble, explains why we use the term *stream language* for the compiler's output.

3.1 ORGANIZATION

In any Algol 68 system based on the RS compiler, the first pass compiles source text into stream language, and the second pass - which must be a genuinely distinct pass - translates stream language into machine code. Although the compiler is machine independent, this attribute cannot extend to the whole of the first pass, whose input and output arrangements will depend on hardware. For each new implementation, therefore, it is necessary not only to write a translator, but also to write a new interfacing shell for the compiler. Figure 1 shows the whole system diagrammatically,

Fig 1



including the library of compiled modules which will provide for the Algol 68 standard prelude and for users' own collections ("albums" in Algol 68-R). To give it the necessary interfaces, the compiler is written as a procedure, with parameters for the shell. The specification is:

```
              PROC compile =
              (PROC(REF[ ]CHAR, REF INT)BOOL input,
               PROC(OUTPUT, INT)VOID output,
               PROC([ ]CHAR, INT)VOID fault,
               PROC(ID, BOOL)YMODINFO give module details,
               PROC(ID, ID, YM)YSPEC give spec,
               PROC(REF[ ]CHAR, BOOL)INT lookup,
               REF[ ]STRUCT(INT type, value) charset
              )BOOL:
```

The parameters *give module details* and *give spec* are concerned with inter-module checking and are not described in the present paper.

The *input* parameter supplies a line of source text each time it is called by the compiler, whose first task is to assemble the stream of characters into larger units such as identifiers and compound symbols. The compiler cannot do this unaided, for it has very few built-in assumptions about the character set or symbol representation conventions. The *charset* parameter associates the ABS values of the characters with their semantic values and combining properties. Bold words are distinguished from identifiers by stropping or by the use of a different alphabet: the two methods can be made available simultaneously if desired. Decisions such as this are entirely within the province of the shell writer, the stropping convention being indicated in the charset parameter. The *lookup* parameter is a list of the representations used in the source text for language words and multi-character symbols ("compound symbols"), along with their meanings expressed in terms of a set of pre-arranged integers used by the compiler.

Output from the compiler consists of *fault* messages, and stream language for the translator. Stream language is a sequence of commands ("imperatives") which are generated by the compiler one at a time. The *output* procedure takes one imperative, of mode OUTPUT, extracts from it the data needed by the translator, and puts it out on the specified stream. Through the output procedure, the implementor has complete control over the selection and format of the data to be passed to the translator. The fields of data in an object of mode OUTPUT are variously essential, gratuitous and superfluous. Gratuitous information typically gives the translator advance warning of properties of some complicated clause at its outset rather than when the complicating features actually appear. Superfluous information is that which is present in an OUTPUT purely for the compiler's own convenience, and should be eliminated by the output procedure. It is strongly recommended to implementors that the output procedure and translator be written by the same person in order to effect a good match between passes.

3.2 STREAM LANGUAGE OUTPUT

The mode OUTPUT defined in the compiler is a union which decomposes into one of a number of more specialized modes, each corresponding to one class of imperative, such as the class of all declarations. There is one special imperative, in a class by itself, which can be disposed of immediately. This tells the translator when to switch its reader from one stream to another. After the stream collation has taken place, the translator sees stream language as one single unbranching series of imperatives, and throughout the remainder of this description, this is exactly how we shall look at it.

3.2.1 The structure of stream language

Stream language defines objects and operations (the basic operators of Algol 68, coercions determined by the compiler, assignments, etc) to produce further objects. This aspect of a program is represented in reverse Polish, as described in later sections. Operands are loaded on to a conceptual stack before the operator is specified. The other facet of stream language is control structure, which is shaped in terms of phrases, serial clauses and closed clauses, like Algol 68 itself. Clauses always deliver objects, which may possibly be void, and serial clauses determine localities in the usual technical sense. However, in spite of the resemblance to Algol 68, if the source text and compiled versions of a particular program are compared, the various structural units will not be found in exact one-to-one correspondence. In its conversion of formulae to reverse Polish form, the compiler will have removed binding brackets (though not when the enclosure is a serial clause with semi-colons), and it may have introduced extra phrases as a consequence of breaking down complicated operations into successions of more primitive ones.

The actual imperatives which impart to stream language its phrase structure reflect familiar symbols of Algol 68. These imperatives all belong to the mode OUTPUT(XCONTROL)*, a structure whose function is indicated in its principal field by one of the following mnemonic integer values.

xbegin, xsemi, xexit, xend, xroutinend,
xcoll, xcollcomma, xendcoll,
xif, xthen, xelse, xfi,
xcase, xin, xcomma, xout, xesac,
xcaseu, xinu, xuchoice, xcommau, xoutu, xesacu,
xfor, xforall, xwhile, xdo, xod,
xfinish

The meanings should for the most part be obvious. Note that *xroutinend* has no counterpart in Algol 68; it occurs immediately after the end of a routine text. Note also that the compiler always distinguishes between different types of closed clause by supplying the appropriate bracket, eg *xcoll* to open a collateral but *xbegin* for an ordinary closed clause, *xcaseu* for a conformity clause but *xcase* for an ordinary case clause. Other fields of an XCONTROL contain various items of supplementary information. Whenever the XCONTROL initiates a serial clause or a closed clause, the mode of the result to be delivered is given. At the start of a serial clause, a property word in the XCONTROL contains bits which show whether the serial clause contains a semi-colon, an EXIT, a label setting,

---

* The notation X(Y) serves as a reminder that Y, the mode under consideration, is a constituent of a union X.

variable declaration, etc. Special bits are also present in all relevant imperatives to assist the translator with dynamic storage control.

We have described XCONTROL first because it is where a top-down examination of stream language should begin. It is also where the structure of Algol 68 shows through most clearly. The OUTPUT union does in fact include quite a large number of modes, of which the five most important are

XCONTROL	control word	
XDEC	declaration of identifier or label	
XLOAD	load of operand	reverse Polish
XOPER	operation	
XROUTINE	routine text	

The mode OUTPUT(XDEC) further subdivides into XDEC(XIDDEC) and XDEC(XLABDEC). An imperative of the latter mode is a label declaration, introduced by the compiler at the beginning of the serial clause containing the actual label setting - which is indicated by another form of XLABDEC imperative. Thus, in stream language, labels are always declared before they are used. The mode XDEC(XIDDEC) corresponds to those Algol 68 declarations which define identifiers, except that the shortened forms of procedure and operator declarations are handled by XROUTINE imperatives. *Any* occurrence of a routine text in the source program gives rise to an XROUTINE imperative, which can be thought of as a declaration in stream language, whether or not it came from a declaration in the source-text. A full identity declaration in the source program, whether for a procedure or any other object, always becomes an XDEC(XIDDEC) imperative. So also does a variable declaration or any operator declaration of the unshortened variety. Priority declarations are absorbed by the compiler and used when converting expressions into reverse Polish.

There are no mode declarations in stream language to correspond with those in the source text of a program, though in a sense *every* mode used in a program is declared in stream language. At the outset of the collated stream, one imperative supplies the translator with an array of all the modes used in the program. Thereafter, any one of these can be represented as an index to the array.

For compactness and simplicity, all cross referencing in stream language is done by integers. Declarations are all numbered. Source text names are passed across by the compiler in stream language declarations only to enable a translator to use them in run-time diagnostic messages. The real stream language identifiers are the declaration numbers, of which there are three separate sets - one for XLABDEC, one for XIDDEC and one for XROUTINE declarations. The XIDDEC numbers are re-used for declarations whose ranges do not overlap. This keeps to a minimum the amount of information the translator holds about identifiers, assuming it organizes its information in the obvious manner.

There is a special imperative at the beginning of a stream language program which tells the translator the maximum sizes of various arrays it will need to declare.

3.2.2 The reverse Polish stack

In the present account of stream language, the reverse Polish stack is a purely conceptual device for remembering operands, and at this conceptual level, the loading of an operand does not imply action of any other kind. In reality, most translators will find it convenient to maintain a real stack in some form or another, to act as a kind of work-bench for the generation of code. However, the way in which operands would be represented on an actual translator stack lies wholly within the province of the translator designer, and is not discussed in the present paper.

Operands appear on the reverse Polish stack in two ways. They may have been placed there as the result of a previous operation, or they may be introduced by an OUTPUT(XLOAD) imperative. The mode XLOAD is a union whose various constituent modes describe the different forms of object which can be loaded. For example, XLOAD(INT) loads a declaration number standing for some object which has previously been declared. Other modes in XLOAD introduce undeclared objects, such as those expressed in the Algol 68 program as denotations (see Appendix).

Almost every kind of object in stream language is described by its Algol 68 mode represented as an integer item of data - not to be confused with the mode of the imperative which handles it. To the compiler, the mode of an object is important as a means of checking program consistency and selecting operator definitions correctly; to the translator its main importance is in determining the size of an object in the running machine. This is of course impossible for an Algol 68 array or vector, as the number of elements is unknown at compile time. However, in addition to elements, every array has a descriptor of fixed size, and in stream language it is the descriptor which is taken as the object to which an array mode applies. This is not to deny that array elements exist! Certain XOPER imperatives call for production of code to find space for array elements in the object machine, and to copy them from one place to another, and yet a set of array elements is never a reverse Polish operand. This is because the translator can obtain all the information it needs about an array from the descriptor. We may therefore conclude that stream language only operates directly on objects of known size. As we shall show, this is the very feature of its design which enables it to break down complicated declarations, generators and assignments into the rudimentary steps which a translator needs.

3.2.3 The creation of new objects

The work entailed in creating a new object is split between compiler and translator, the compiler doing as much as it can without knowing anything about the final object machine. It cannot do very much with source text denotations, as the translator must deal with machine representations. Denotations are therefore passed into stream language almost literally, in XLOAD imperatives, though number denotations are tidied up into standard formats. Routine texts are compiled like any other pieces of Algol 68, with formal parameters expressed by XIDDEC imperatives of type "xfdec", the whole routine being preceded by an XROUTINE imperative which gives it a declaration number in every case.

Jumps are treated as objects in stream language, and loaded by their label declaration numbers. No action is taken until specified by a subsequent coercion (an XOPER).

In Algol 68, a reference is created by a generator or a variable declaration; its purpose is allocation of storage space for an object of given mode. The object can be described as 'simple' if the generator or declaration contains no array bounds, for then the total amount of space to be allocated is known from the mode, and stream language does no more than reflect the Algol 68 constructions. A generator becomes an XLOAD(XGEN) imperative, which puts a new local or heap reference on the reverse Polish stack, and a variable declaration becomes an XDEC(XIDDEC) of type "xvardec", or "xivardec" if the declaration is combined with an initial assignment. Either type creates a new reference and gives it a declaration number. In addition, xivardec will find the object for initial assignment on the reverse Polish stack, and remove it. Being a declaration and not an operator, it leaves no result behind.

When the Algol 68 generator or declaration contains array bounds, space for elements has to be generated dynamically. As these may introduce further arrays, the task can be a protracted one. The compiler breaks it down so that the translator is never called upon to deal with more than one array at a time. As an example at one level only, consider the declaration

```
        [1:n]REAL r;
```

which requires the translator to

— generate space dynamically for n reals,

— create the associated fixed size object (ie the descriptor) of mode [ ]REAL,

— create an object of mode REF[ ]REAL and assign the descriptor to it (a 'static' assignment).

In outline (for the compiler actually does a little more), this maps into stream language as

XLOAD the lower bound 1
XLOAD the upper bound n
XOPER "xbdpack"

[packs the bounds into a single object]

XLOAD a boolean for local/heap, here local

XOPER "dyngrab"

[takes the boundpack and boolean operands, generates space for array elements and delivers the descriptor]

XDEC(XIDDEC) "xivardec"

[creates the variable, gives it a declaration number, takes the descriptor from the reverse Polish stack as operand and assigns it statically to the array variable]

It is particularly to be noticed that an xivardec initialization is always a static assignment, ie assignment of a fixed size stream language object only, with no regard for any array elements. Normally, xivardecs are used when there are initial assignments in the source text (eg REAL x := 0.0), but not if the object declared is an *array* variable. Initial assignment of array elements is carried out separately as a standard assignment operation, described in 2.2.4. An array generator gives the same sequence as that shown for an array declaration, except that the XIDDEC "xivardec" is replaced by the XOPER "statgrab". Instead of creating a reference and then declaring it as a variable, statgrab creates the reference but puts it on the reverse Polish stack after statically assigning the descriptor.

In actuality, all but the final step shown above is wrapped up in a stream language routine invented by the compiler for the given mode. In the example, the routine would deliver the [ ]REAL as operand for the xivardec. In a more general case, eg from the source declaration

```
STRUCT(BOOL b, [1:n]REAL r)s;
```

the routine would deliver the fixed-size object of mode

```
STRUCT(BOOL b, [ ]REAL r)
```

as operand for the ivardec.

For declarations involving array space at more than one 'depth', the routine for the whole array mode calls similar routines for any contained array modes. For example, consider the source declaration

```
[1:m]STRUCT(BOOL b, [1:n]REAL r)t;
```

The mode already considered is now contained in an array, and the inner routine (nr, say) will deliver the STRUCT(BOOL b, [ ]REAL r) inside the routine (mr, say) for the whole mode, where it will be assigned to each of the m array elements - as m fixed size objects. The routine mr will finally deliver the fixed size object

```
[ ]STRUCT(BOOL b, [ ]REAL r)
```

for assignment to t in the ivardec.

3.2.4 Assignment

The XOPER "xassign" takes two operands, a destination and a source. It leaves the first operand on the reverse Polish stack at the conclusion of the assignment, clearly imaging the Algol 68 construction. But the stream language operation gives the translator less work than would an Algol 68 assignment in its full generality. As with a complicated declaration, the compiler invents and calls specially tailored routines to break down a complicated assignment.

The mode of the destination determines what xassign is called upon to do. If it is a ref vector or ref array (non flexible), the actual elements are to be copied and the associated descriptors left untouched. This is 'dynamic assignment'. For every other destination mode, including ref flex array and ref flex vector, xassign means static assignment. The object to be copied is the stream language source operand, which may be a descriptor but cannot be a set of elements. From a stream language point of view, dynamic assignment is the oddity, as the operands are not the objects directly involved in the copying process. The translator's generation of code to copy array elements is a kind of side-effect to an otherwise inert stream language operation. At this point, it is worth recalling the initialized variable declaration of 2.2.3, as the assignment embodied in the ivardec is always of the *static* type, irrespective of mode. It is not just an absorbed xassign operation.

In a dynamic assignment, the elements to be copied will always be objects of known size. The translator is never asked, all in one go, to copy elements which themselves contain elements to be copied. The invented assignment routines see to that; by the use of "forall" constructions, all the loops are given explicitly.

One further operation is required to complete the subject of assignment. An Algol 68 assignment to a flexible array or vector variable implies the making of a new set of elements and a new descriptor, and the xassign in this case deals only with the descriptor. A special operation "xcopy" always precedes the static assignment to a flex variable. This operation takes the descriptor from the right-hand side of the Algol 68 assignment as its one operand, generates the required amount of space (on the heap) for a new copy of the elements, copies the elements, constructs a new descriptor and delivers this as the result of the operation. The xassign operation then picks up this new descriptor and statically assigns it to the flex variable as already described.

3.2.5 Summary of stream language operations

The OUTPUT(XOPER) imperative is a structure whose three fields define the operation, give the mode of its result and provide other useful information ("param") where necessary. The various operations are listed below, with brief descriptions of their meanings - which should not be taken as strict definitions.

Monadic and dyadic operators from Algol 68

xmonop	monadic operator
xdyop	dyadic operator

The different meanings of an operator which apply for different modes of operand are known as versions. The param field of the imperative specifies the operator and its version in each case.

Coercions and operations similar to coercion

xderef	dereference
xunite	unite (from non-union)
xuniteu	unite (from sub-union)
xwrc	widen real to complex
xwir	widen integer to real
xwib	widen integer to bits
xwbvb	widen bits to vector of booleans
xis	mode to i-struct of mode
xvec	mode to vector of mode
xarr	mode to array of mode
xisvec	i-struct to vector
xisarr	hierarchy of i-structs to param-dimensional array
xvecarr	vector to one-dimensional array
xarrarr	array to array with extra dimension
xniltom	NIL to given mode
xskiptom	SKIP to given mode
xgotoproc	coerce jump to a procedure mode
xgotom	coerce jump to other given mode
xvoid	void

Field selection and array indexing

xselect	select paramth field of operand
xsimpleindex	produce single element from given array or vector and subscripts
xtrim	produce trimmer from its component parts
xtrimindex	produce subset from given array or vector and trimscripts

Procedure calls

xparampack	remove "param" actual operands from stack
xcall	call the given procedure with the actual parameters removed by xparampack and produce the result of the procedure

Assignment

xassign	already described (3.2.4)

Space finding

xbdpack	pack the given array or vector bound(s) as a single object
xdyngrab	produce descriptor for array or vector, having generated necessary space for elements from given boundpack and boolean (local/heap)
xstatgrab	make space for object taken from stack, assign it statically to the space, and deliver the reference
xcopy	replace given descriptor with new descriptor, having generated new heap space and copied the elements into it (see 2.2.4)
xdefaultbd	tuck default lower bound of 1 under top item on stack (for Algol [n]INT)

Operations associated with straightening (xstraight, xprestraight and xstrindex) are not described in this paper. Nor is xdeunite.

3.3 IMPLEMENTATION

To harness the RS compiler for use on a new machine, the obvious need is the translator to convert stream language into machine code, but one must never lose sight of the fact that the final product is a *system* and not just a collection of programs. The importance of the shell for the compiler is obvious from Figure 1 in section 2.1, a major part of which is concerned with the updating and retrieval scheme for library modules. Not least in importance here is the actual content of the system library, including transput. Finally, proper provision must be make for run-time diagnostics, not shown in Figure 1 but clearly a crucial part of the system.

The development of a system initially depends on the process known as bootstrapping, and we conclude with an outline of its three stages.

Stage 1 begins with getting the compiler running on some existing machine. As the compiler is written in a subset of Algol 68-R, the ICL 1900 is the obvious machine for bootstrapping, but there is nothing to prevent the use of some other Algol 68 machine provided that the compiler is suitably adapted. With a temporary first pass actually running, the next phase of stage 1 is to produce a matching translator which generates code for the final object machine. It may be convenient to write this translator in the bootstrapping machine using the same language as the compiler, as shown in Figure 2. Alternatively, it can be written for the new machine from the start, provided that the new machine supports a suitable high-level language (Figure 3). Either way, we are now equipped with a means of producing code for the new machine from Algol 68 source text.

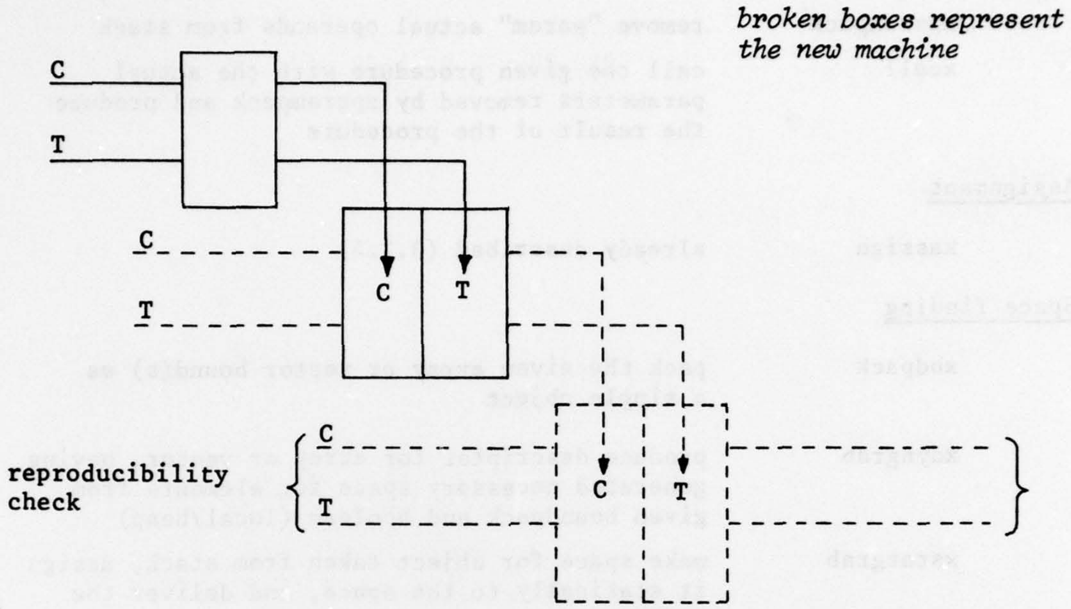


Fig 2

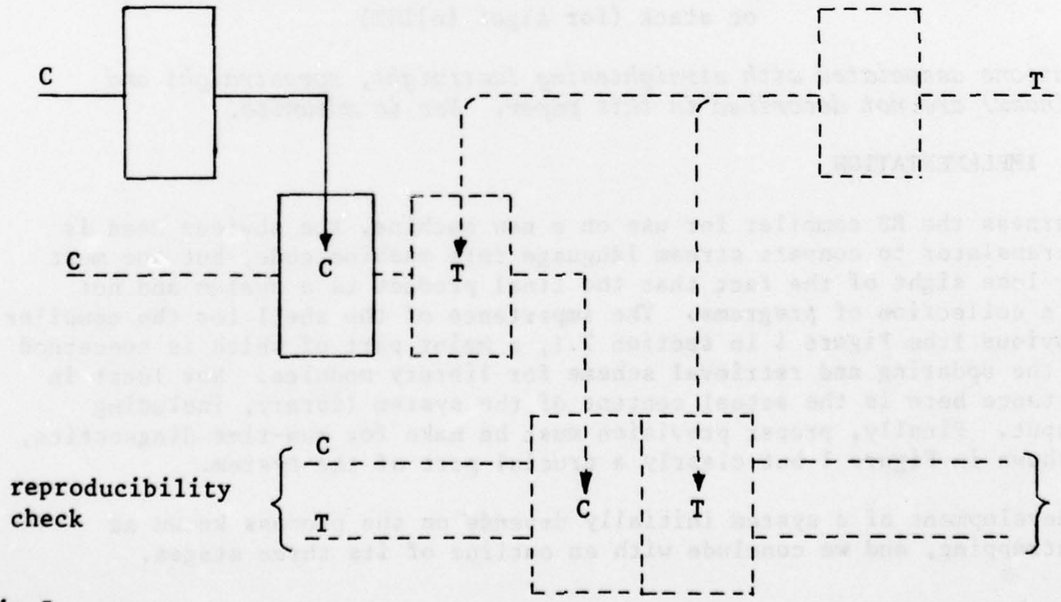


Fig 3

Stage 2 uses the result of stage 1 to compile and translate a final version of the compiler and a shell suitable for the new machine. The translator is also compiled and translated when the first of the two plans is adopted (Figure 2). The result of stage 2 is a compiler, shell and translator which can be loaded in the new machine.

Stage 3 consists of tidying up. The translator probably requires enhancement, for up to this point it has only had to deal with the system itself, which is almost certainly in a subset of stream language. The RS compiler produces a subset, and the other parts of the system can with advantage do the same. Now the compiler and translator should be compiled and translated in the new machine to check that the whole system is self-supporting. This may entail rewriting the translator, for if the second plan (Figure 3) was adopted, it will probably not have been written in Algol 68. One final adjustment will be found necessary before putting the system into service. In order to deal with the RS compiler, the shell has had to accept certain constructions peculiar to Algol 68-R. A simple shell revision now bars these constructions and completes the switch to Algol 68 of the Revised Report.

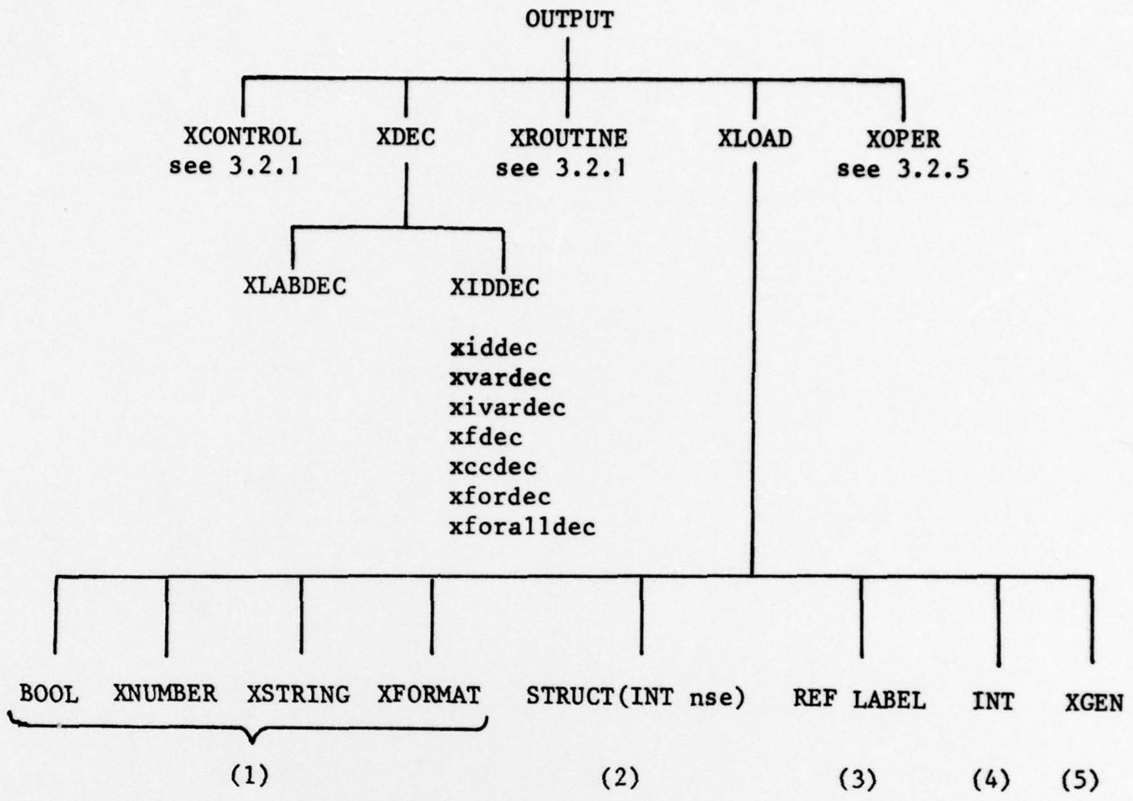
4 REFERENCES AND ACKNOWLEDGEMENTS

1 I F Currie, "A Portable ALGOL 68 System", Proceedings of the Conference on Applications of Algol 68, held at the University of East Anglia, March 22-25, 1976, sponsored by ICL and the BCS Algol Association.

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

We acknowledge help received from our colleagues J D Morison, R J Granville and I F Currie in the process of reaching what we hope is a reasonably faithful and self-consistent account of the functions of the RS compiler. We wish also to thank Robert Hutchings of Oxford University for pointing out obscurities in our source material and offering many interesting suggestions.

22 July 1977



Source text features loaded

(1) Denotations for booleans, numbers, strings and formats

(2) NIL, SKIP, EMPTY

(3) GOTO some label

(4) Declared object

(5) Generator

Types of XIDDEC

The types xvardec, xivardec are discussed in 2.2.3. The xiddec is an identity declaration taking the object to be declared as operand. The xfdec 'declares' a formal parameter of a procedure, xccdec the formal identifier in a case of a conformity, xfordec the identifier in a for-statement and xforalldec the identifier in a forall-statement.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ........UNCLASSIFIED.. ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Tech Note 802	3. Agency Reference	4. Report Security Classification Unclass
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location RSRE Malvern		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

"Introduction to the 'RS' Portable ALGOL 68 Compiler"

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Bond S G	9(a) Author 2 Woodward P M	9(b) Authors 3,4...	10. Date 7-7-77	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement

[illegible]

Descriptors (or keywords)

Computing Compiler Algol Language Software Translator

continue on separate piece of paper

Abstract

This paper describes the portable compiler for Algol 68 developed at RSRE by Currie and Morison. Chapter 2 defines the language extensions handled by the compiler, and the system of modular compilation as seen by the user. Chapter 3 outlines the structure of a complete system, emphasizing the design of the intermediate language produced by the compiler for input to a machine-dependent translator. Though factual, this paper is not the system documentation.