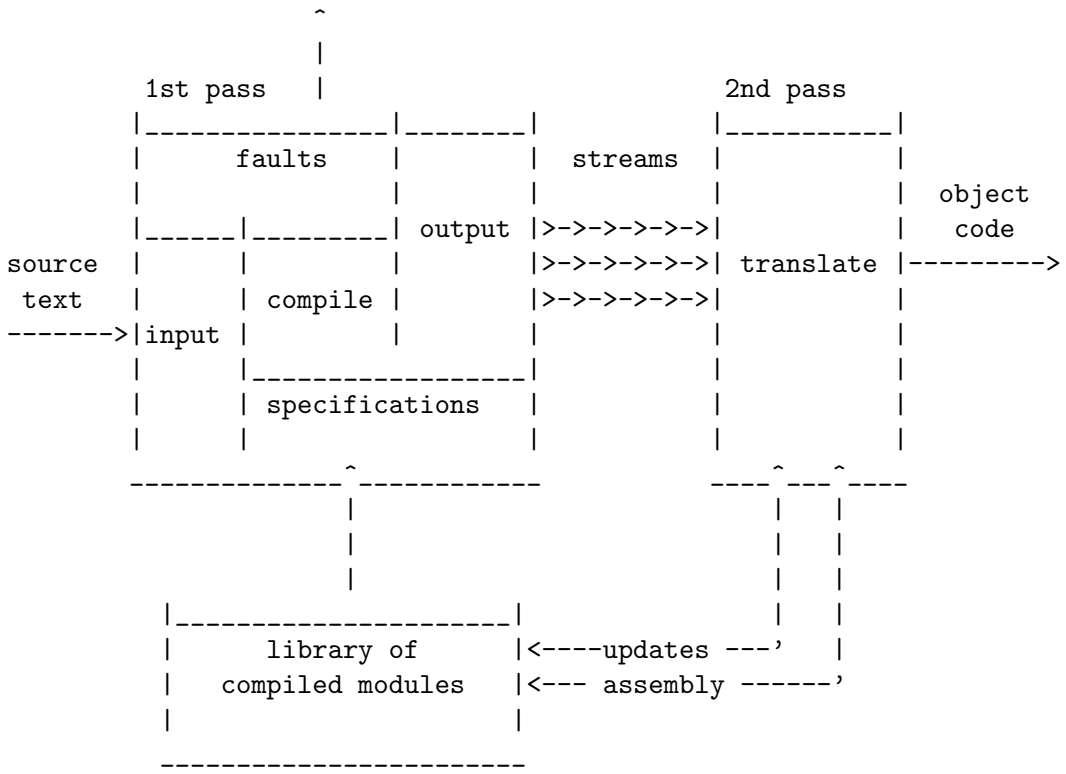
The RS Compiler for ALGOL 68

Published 1978

Defence Research Agency, Malvern, UK

1 Introduction

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 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.



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. The Figure shows the whole system diagramatically, 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.

1.1 The Source Language

The language accepted by the compiler is Algol 68 as defined in the Revised Report with some deviations, principally that modes and identifiers must be declared before they are used. There are restrictions on the use of `LONG` and `SHORT` (see Appendix 2). 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 and compiler 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 Appendix 4.

2 Stream Language Output

Stream language consists of a sequence of commands ("imperatives") which are generated by the compiler one at a time. One of the parameters supplied to the compile procedure is a procedure used for the output of stream language. It has a parameter of mode `OUTPUT`; this is a union which decomposes into one of a number of more specialised 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 Part A, this is exactly how we shall look at it.

2.1 The Structure Of Stream Language

Stream language defines objects and operations to produce further objects. The operations arise from the operators of Algol 68, from coercions and from operations such as assignment. Operands are loaded on to a conceptual reverse Polish 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)`. (Here we are using the notation `X(Y)` to serve as a reminder that `Y`, the mode under consideration, is a constituent of a union `X`.) The Mode `XCONTROL` is 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, 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 indication

`XDEC` declaration of identifier or label

`XLOAD` load of operand

`XOPER` operation

`XROUTINE` routine text

The mode `XDEC` is itself a union of `XIDDEC` and `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 a vector containing information about all the modes used in the program. Thereafter, any one of these can be represented as an index to the vector.

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 that it organises its information in the obvious manner.

There is a special imperative at the beginning of a stream language program which tells the translator the sizes required for its various vectors.

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 other, 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 this section.

Operands appear on the reverse Polish stack in two ways. They may have been placed there as a 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.

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 can handle easily.

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 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 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` initialisation is always a static assignment, ie assignment of a fixed size stream language object only, with no regard for any array elements. Normally, `xivardec`s 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 A2.4. 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`.

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 complicated declarations, the compiler invents and calls specially tailored routines to break down complicated assignments.

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 initialised variable declaration of A2.3, as the assignment embodied in the `xivardec` is always of the static type, irrespective of mode. It is not 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 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 the Figure in section A1, 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 made for run-time diagnostics, not shown in the Figure but clearly a crucial part of the system.

The development of a system initially depends of 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 machine, which will normally be different from the target machine. Since the RS compiler is written in its own language, the simplest method is to use an existing RS implementation, although there is nothing to prevent the use of some other machine for bootstrapping 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. Alternatively, it can be written for the new machine from the start, provided that the new machine supports a suitable high-level language. Either way we are now equipped with a means of producing code for the new machine from Algol 68 source text.

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. 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 was adopted, it will probably not have been written in Algol 68.

4 The Compiler Shell

The "compile" procedure is specified as follows:

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

This procedure must be called in pass 1 with actual parameters written to suit the translator and the hardware in which the system is to run. These actual parameters constitute the 'shell' of pass 1 (see Figure in A1).

The procedures `give module details` and `give spec` are concerned with inter-module checking and are described in Part C. The remaining parameters are dealt with in this present chapter.

Most one-dimensional arrays in the RS compiler have lower bound 1, so they have been written as vectors, which are normally more efficient than arrays. Since the same will be true of most translators, vectors are used in the interface wherever possible. One-dimensional arrays may be used throughout if desired.

4.1 Input Of Source Text

Source text is input by the procedure

```
PROC input = (REF VECTOR [] CHAR source, REF INT size) BOOL:
```

which is called by the compiler. In `source`, the compiler provides space for `linesize` characters of text (see Appendix 5). The input procedure must put a line of text into this space and give the actual number of characters in `size`. Blank lines are allowed. The procedure must ordinarily deliver `TRUE`, but must deliver `FALSE` if nothing has been supplied in `source` because there were no more lines of text for the compilation. A suitable amount of source text must be retained by the input procedure for output by the fault procedure (B1.5) when required.

The compiler's first task is to assemble characters into the larger entities, such as identifiers, required for syntax analysis. It has no knowledge of the character set or representations of symbols, and all such information has to be made available through its parameters `charset` and `lookup`. These are described in sections B1.2 and B1.4.

4.2 The `charset` Parameter

The `charset` parameter is an array such that `charset[i]` describes the properties of the character whose `ABS` value is `i`. Each element of the array has a type field and a value field, which must be filled in according to certain fixed conventions. For the compiler to assemble identifiers, it needs to know which characters are the letters, which the digits and which the space character. This is because identifiers must start with a letter and continue with letters or digits, with spaces ignored; the identifier is thus terminated by the first non-letter non-digit character encountered. The letters a to z must all be given type 3 and values 10 to 35 respectively. The digits 0-9 are given type 4 and values 0-9. The space character is given type 2 and value 37. If there are any further alphabetic characters besides a to z needed in identifiers, such as accented letters, they must be given type 3 and value 36, and cannot be used other than in identifiers.

Bold symbols are assembled by the compiler in one of two ways. If the character set contains only one alphabet, a 'stropping' scheme must be adopted, and this can take two forms. In 'single' stropping, the bold symbol is simply prefixed by a strop character, whilst in "matched" stropping it is enclosed in a pair of identical strop characters. Aside from the strop characters, the form of a bold symbol is exactly similar to that of an identifier, except that spaces are not allowed. When single stropping is used, the strop character is specified as type 7 and the bold symbol terminates in front of a character, such as space, which cannot be a legal continuation of the symbol. When matched stropping is used, the strop character if specified as type 8, and the bold symbol terminates in the same way except that the compiler checks that the terminator is the strop character and moves past it. (If it is not, the fault procedure is called.) The value field for the strop character can be set to 100 if the character is not to be used for any other purpose. Provided that reasonable discretion is used, the strop character can be made to do double duty and serve as a symbol in its own right, such as the decimal point symbol. The value field would then be filled in to indicate the required meaning as described in B1.3.

If a second alphabet is available in the character set, it can be defined to be the bold alphabet, and there will be no need to define a strop character. Bold letters must be given type 6 and the value field set to the negative number *-(n + 1)*, where *n* is the `ABS` either of the bold letter itself, which we shall describe as a 'direct value' or of the corresponding ordinary letter ('indirect value'). The choice, which must be the same throughout the alphabet, is governed by the way the lookup table for bold symbols is to be organised, as discussed later.

The character used as the quote symbol must be given type 5 and value 100, which debars it for any other purpose except inside comment or, doubled, inside string or character denotations. If there are any characters whose use is to be restricted solely to the insides of strings or comment, these must be given type 1 and value 100 ('illegal representation').

Once the characters of types 1 to 8 have been dealt with as described above, all the remaining characters are available, singly or in combination, to represent symbols. A character which never combines is given type 20, while a character which can occur as part of a compound symbol is given type 21, 22 or 23 according to the positions it is allowed to occupy. These are shown in the table set out below. Characters which can be used in compound symbols may be used singly as well (for consider '`:`', '`=`' and '`:=)`'. The value field in charset is normally used to specify the meaning of the character when it is used as a single-character symbol (but see B1.3 for a possible exception to this rule).

When the compiler encounters a character which may be the start of a compound symbol, it always tries to complete the symbol and terminates only when further continuation is impossible. It is therefore advisable that all monadic operators be represented by characters of types 20 or 21, as this enables consecutive monadic operators to be used in a program without space separation. To see this, consider a pair of monadic operators such as '`+-`'. If the minus can only occur alone or at the very beginning of a compound symbol, it cannot be the second character of a compound symbol starting '`+-`'. A space between the plus and minus is therefore unnecessary.

In the assembly of identifiers, bold symbols, string denotations, numbers and compound symbols, the end of a line will always terminate an item. If a string denotation is to extend over more than one line, a quote symbol must be given at the end of each line and the next line introduced by a further quote.

If an implementation is to accept both upper/lower case programs and programs written using a stropping convention, the shell must convert one representation to the other.

The field settings for `charset` are as follows (a value of 100 means '`invalid representation`').

type	*meaning of type (value)*
1	character which can only be used inside strings or comment (100)

2	space (37)
3	letter, for use in identifiers, labels and field selectors (a to z (10-35), any others (36))
4	digit (0-9)
5	string quote (100)
6	bold letter (see B1.2)
7	single stropping character (see B1.2)
8	matched stropping character (see B1.2)
20	non-combining character (see B1.3)
21	character which, when used in combination, can only be the first character of the compound symbol (see B1.3)
22	character which can be used anywhere in a compound symbol (see B1.3)
23	character which, when used in combination, can only be the last character of the compound symbol (see B1.3)

4.3 Values Of Symbols

The compiler assumes nothing about the way modes, operators or any other language symbols are to be represented in programs; it understands the meanings of symbols in terms of its own fixed integer values. From the 'type' information given in charset, it can assemble single and multiple character symbols, but cannot convert them into values by itself. The shell writer must therefore decide on the desired character representations for all the symbols listed in Appendix 1, and make arrangements for supplying the corresponding integer values to the compiler. (Appendix 1 includes symbols for all of the operators of the standard prelude, which are handled by the compiler rather than the library system.) The representations for symbols can be single characters, bold symbols or compound symbols. The value fields for characters with types less than 20 have already been laid down in B1.2. The values for all compound symbols or bold symbols - some of which might consist of a single bold letter - must be embodied in a special lookup table as described in B1.4.

The values for the single character symbols of types 20-23 are normally put into the value fields of the charset array. The reason for this is simply to optimise by reducing the number of calls of the lookup procedure - in principle all symbols could have been treated alike and included in one table. The optimisation may be overridden by giving such characters the value 99; this causes the lookup procedure to be called as for compound and bold symbols.

The input system as a whole has great flexibility. The shell writer can choose whatever representations for symbols seem desirable, although he is expected to include those given in Appendix 1, which are taken from the Algol 68 report. He may supplement the suggested representations with alternative forms for the same symbol (for instance, `&` as alternative for `AND`). And he can effectively subset the language giving any unwanted symbol (eg `#`) the value 100 ('illegal representation').

4.4 The Lookup Procedure

When the compiler has assembled a bold or compound symbol, it calls the shell procedure

```
PROC lookup = (REF VECTOR [] CHAR symbol, BOOL bold) INT:
```

The parameter `symbol` will provide the actual characters of the symbol, and incidentally must not be assigned to in the body of `lookup`. The parameter `bold` will be `TRUE` for bold and `FALSE` for compound symbols. The procedure must look up the symbol in its lookup table, and

deliver the integer value which can be found from Appendix 1. Depending on how the lookup table is organised, the information as to whether the symbol is bold or compound may help in minimising search time.

Bold symbols represented in source text by single or matched stropping are served up in the `symbol` parameter without the strop characters, and will look just like unspaced identifiers. Where a bold alphabet is in use, the `symbol` parameter will use bold letters if their values in charset were chosen to be "direct" (as defined in B1.2), but will use ordinary letters if they were "indirect". The choice of direct versus indirect depends on the environment in which the RS system is to be implemented. Where some users' programs can use a bold alphabet and others must use stropping, the indirect system has advantages in ensuring a uniform way of keeping symbols in a library. However, in an environment where stropping need never be used at all, the direct system has the advantage that it does not transform the alphabet unnecessarily.

4.5 Output Of Fault Messages

The `fault` procedure deals with output of compile-time error messages. The `VECTOR [] CHAR` parameter contains the diagnostic message and the INT is the character position in the input text. The purpose of "fault" is to output the message in a presentable form, preferably with an extract of the source text indicating the location of the error.

Since the compiler generally avoids the use of strings in its source-text, names and messages have to be introduced by means of its `message` parameter. This is a procedure that delivers a reference to a particular string when given an integer parameter. The strings must be supplied by the shell writer as defined at the front of the compiler. The character '`%`' means that the following word should appear in upper case, while '`.`' followed by an integer specifies a "parameter" of the fault message. The combination '`%.`' means that the parameter is the name of a mode or operator. If "indirect" values are being used for the bold alphabet, the fault procedure will need to convert letters in the following identifier.

The purpose of the message parameter is to allow translation of error messages into languages other than English without modifying the text of the compiler. Except for the removal of '`%`', it should not be altered for any other reason.

4.6 Output Of Stream Language

Output of stream language from the compiler is effected by its repeated calls of the pass 1 shell procedure `output`,

```
PROC output = (OUTPUT imperative, INT stream) VOID:
BEGIN
  The procedure encodes the information in the imperative
  and outputs it on the specified stream
END
```

The job of this procedure is to output the imperative in whatever form is most suitable for input to the translator in pass 2. As this will depend strongly on the design of the particular translator, the task of designing the output procedure is best undertaken by the translator writer himself.

The mode `OUTPUT` is a union whose constituent modes represent different types of imperative, and the first task of the output procedure is to decompose the union. The type of the imperative and the data held within it must now be encoded by the procedure as an implementation- dependent representation of stream language. The present document cannot make any assumptions about such encoding, and will therefore describe stream language in Algol 68 terms, starting from the mode `OUTPUT`. But it is essential to realise that the output procedure gives the implementor complete control over the format and content of the information which is actually

output from pass 1. Some portion of the data provided is superfluous, being present in the `OUTPUT` imperative only for the compiler's own convenience, and this can be omitted from the stream language representation altogether. Other information is 'gratuitous', either because it is not strictly necessary or because it is duplicated. However, much of this gratuitous information is likely to be useful to the translator, and the output procedure should select what is needed in any particular implementation.

5 Stream Language In Outline

An Algol 68 program is concerned with objects and operations on objects to produce new objects. For the purpose of code generation, a translator must construct suitable representations of these objects. Such a representation will be termed a "translator value", or simply a value. It will normally include the location and size of the item in the running machine. In the case of an array or vector - whose total size is unknown — a translator value can only describe explicitly the part which has a fixed size, ie the descriptor or "static part". However, translator values do form a sufficient basis for generating code to handle all objects, whether static or dynamic.

5.1 The Imperatives

Each imperative in stream language belongs to a constituent mode of the compiler's `OUTPUT` mode, which is a union. At the first level of decomposition, these constituents are

`XEDIT` used for stream control

`XDEC` a declaration, which provides data for a new translator value and gives it a declaration number

`XROUTINE` initiates the declaration of a routine text and gives it a declaration number in a different series

`XLOAD` requires the translator to construct a new value, or take a given declared value, and stack it as a reverse Polish operand

`XCHARS` in conjunction with `XLOAD`, provides a quotation from the source-text for denotations

`XOPER` specifies an operation to be performed on the reverse Polish stack

`XWARN` supplements `XOPER` by providing advance warning of certain dyadic operations

`XPRAG` copies the start of source-text pragmats

`XCHARPOS` indicates character positions in the source-text

`XCONTROL` indicates the structure of the Algol 68 program

`REF VECTOR [] MDE`
a vector of the modes required for the program

`XSIZES` giving advance notice of sizes of vectors needed by the translator, such as the above vector of modes

`XMODINFO etc`
concerning modules - see section 3 of Part C.

We now present the above in more detail.

5.1.1 XEDIT

The input section of the translator must read in the parallel streams of imperatives as encoded by the pass 1 output procedure, taking imperatives variously from one stream and another to form a single unbranching sequence. The process of stream collation is performed by the translator in response to imperatives of mode `XEDIT`, which it will encounter on every stream. An `XEDIT` is a `STRUCT(BOOL up)` which tells the translator to switch to an adjacent stream, one up or one down from that currently being read, according as `up` is `TRUE` or `FALSE`. Reading starts on stream 0 (which contains module information only), and the next streams down are 1, 2 etc. After switching to a new stream, reading always continues from the place where that stream was last left (or from its beginning if not yet read at all) until an `XEDIT` is met. Throughout the remainder of the present account, this collation will be assumed to have been done.

5.1.2 REF VECTOR [] MDE

A vector of modes is passed to the translator in a preliminary `REF VECTOR [] MDE` imperative, enabling the translator to represent any mode as an integer index.

5.1.3 XDEC

An imperative of mode `XDEC` is a stream language declaration which requires the translator to construct a new value, and be able to refer to it by means of a declaration number supplied in the `XDEC`. The mode `XDEC` is itself a union of `XIDDEC` and `XLABDEC`. When the source-text declares an identifier, the stream language will generally produce an `XIDDEC` containing its source mode and a declaration number `decno`. (The exception is the Algol 68 routine text declaration, which is handled separately — see `XROUTINE`.) At the beginning of a serial clause in which a label setting occurs, the stream language will produce an `XLABDEC` which provides the label with a `labno` (in a series distinct from the `decno` series).

5.1.4 XROUTINE

Every routine text in the source, whether or not it is part of an Algol 68 declaration, is given an `rdenno` by an `XROUTINE` imperative, which represents the start of the routine. For certain purposes the compiler invents its own routines; these synthetic routines are also declared and numbered in the `rdenno` series.

5.1.5 XSIZES

Declaration numbers enable the translator to keep declared values in vectors for future use. Numbers in the `decno` series are issued and re-issued by the compiler in a stack-like fashion which corresponds to the nesting of source-text ranges - so a given `decno` may correspond to different identifiers in different non-overlapping ranges. This does not, however, apply to `labnos` or `rdennos`. The required vector sizes in the translator are given in an initial `XSIZES` imperative of the form

```
STRUCT (INT norden, nomodes, nolabs, nodecnos,
                nomodules, nolibinds )
```

The fields give the vector sizes for `rdennos`, `VECTOR [] MDE`, `labnos`, `decnos` and two vectors concerned with modules (see Part C).

5.1.6 XLOAD

Stream language differs from Algol 68 in that it breaks down all of the source text into operations which are judged to be primitive for the production of code. It adopts the principle that all of the operands necessary for any operation must be present on a stack before the operation can be carried out; the operation is thought of as replacing its operand(s) by a single result which is kept as an operand for some future operation. The imperative which produces a new operand on the stack in `XLOAD`. Any value which has been declared in an `XDEC` or `XROUTINE` can be the object of an `XLOAD` imperative. So also can values corresponding to source-text denotations (see `XCHARS`), generators, and the results of alien or code insertions.

5.1.7 XCHARS

Imperatives of mode `XCHARS` always amplify an `XLOAD` for a source-text string or format denotation, or alien or code insertion. The `XCHARS` imperatives provide quotations from the source-text.

5.1.8 XOPER

The actual operations which can be carried out on loaded values are described by imperatives of mode `XOPER`, which are ultimately defined by the object code they must produce. They fall into the following categories

monadic and dyadic operators
: From the Algol 68 standard prelude.

coercions and similar operations
: Stream language specifies these explicitly wherever they are required.

field selection and array indexing
: For example, given the operands `a`, `i` and `j`, one operation produces a value for `a[i, j]`.

procedure calls
: Which produce, for example, a value for the result of `f(x, y)` from the operands `f`, `x` and `y`. This is not a primitive operation, as it needs one `XOPER` to set up the actual parameters and another to do the procedure call.

assignment
: The primitive stream language operation assigns an object of known size, or a vector or one-dimensional array of such objects. More complicated assignments are reduced to this primitive level by the compiler. An Algol 68 assignment to a flex variable is broken down into two separate stream language operations. One finds new space for the elements, copies them into it and makes up the descriptor. The other assigns the descriptor to the flex variable.

space finding
: An `XOPER` generates space for elements of vectors or arrays whenever required as a result of a source-text declaration or or generator, or an assignment to a flex variable. "Static" space is found in various ways. The static space required for a named object is found by means of its `XIDDEC`. Unnamed objects, ie generated objects, obviously cannot be dealt with in this way. If they are of fixed size, the space is found by an `XLOAD` which puts the reference on the reverse Polish stack. Otherwise, the fixed part of the total space requirement is found by an `XOPER`. This particular `XOPER` is only used when some part of the static space is needed for descriptors.

straightening
: Two operators provide coercion of a row, vector, struct, i-struct or union to a "straight" — a language extension described in Appendix 4. A third operation provides indexing of the straight to pick out one member.

5.1.9 XWARN

For the convenience of certain translator designs, the `XWARN` imperative defines certain dyadic operations after the loading of the first operand. This is additional to the `XOPER` which occurs in the normal reverse Polish position.

5.1.10 XPRAG

The compiler normally passes the first line of a source-text pragmat to stream language.

5.1.11 XCHARPOS

The character positions of significant symbols in the source-text (eg controls) is output so that the translator will read the imperative at the appropriate time.

5.1.12 XCONTROL

Overall structure of Algol 68 source-text is mirrored in stream language down to the phrase level. The source text symbols which delimit phrases appear in stream language as imperatives of mode `XCONTROL`, which is a structure with a `fn` field for the particular delimiter (`xif`, `xsemi`, etc). The opening round bracket for an ordinary closed clause and that for a collateral are

distinguished by the compiler and given different `fn` fields. This is a typical example of advance information from the compiler. Every `XCONTROL` at the start of a serial clause gives advance information about the properties of that clause, such as the presence of declarations and the mode of the result delivered.

5.2 Syntax Analysis Of Stream Language

An abridged syntax of stream language is shown below. In this syntax, the imperatives `XLOAD`, `XOPER`, `XIDDEC` And `XLABDEC` are all grouped together, as they do not contribute to syntactic structure in any significant way. Structure is imposed on stream language by the `XCONTROL` imperatives, whose `fn` fields enable the various different types of closed clause to be recognised and their component clauses to be picked out.

The task of a translator is to read and act on stream language, and it can use procedures which mirror the syntax. To read in a closed clause, it can use a procedure cclause which calls another procedure, sc, to read the component serial clauses, as shown in the lower half of the skeleton translator. The variable `currentxc` holds the latest `XCONTROL` Imperative. Similarly, the `sc` and `cclause` procedures can call on a phrase procedure to read the constituent phrases. After a call of phrase, the current `XCONTROL` will obviously be one which terminated a phrase, and all but two of these also terminate a serial clause. These two are `xsemi` and `xexit`, which correspond to '`;`' and `EXIT` between phrases in the Algol 68 source-text.

The phrase procedure does the actual reading of the imperatives, calling `cclause` when it reads an `XCONTROL` which is a 'cclause starter', ie one of `xbegin, xif, xcase, xcaseu, xcoll, xdo (when not preceded by xwhile), xwhile` in the abridged grammar we are using. Exit from the phrase procedure occurs when an `XCONTROL` which is a 'phrase terminator' is read, ie one of the following:

```
xend        xcollcomma
xthen       xendcoll
xelse       xroutinend
xfi         xinu
xod         xdo (when preceded by xwhile)
xin         xcommau
xcomma      xoutu
xout
xesac       xsemi ) these phrase terminators do
xesacu      xexit ) not terminate serial clauses
```

5.2.1 Abridged Syntax Of Stream Language

5.2.1.1 Notation

Class names are placed on the left, with their alternative expansions on separate lines on the right.

Square brackets enclose an optional item, which, if starred, can be repeated any number of times.

`cclause` stands for closed clause, `sc` for serial clause, `enq` for enquiry clause. A stream language primary is not to be confused with a primary in Algol 68.

`xbegin`, `xend` etc are imperatives of mode `XCONTROL`, whose `fn` fields are the integers `xbegin`, `xend` etc.

5.2.1.2 Syntax rules

```
cclause = xbegin  sc  xend
          xif  enq  xthen  sc  [xelse  sc]  xfi
```

```
          xcase  enq  xin  phrase  [xcomma  phrase]*  [xout sc]  xesac
          xcoll  phrase  [xcollcomma  phrase]*  xendcoll
          xcaseu  enq  xinu phrase [xcommau phrase]* [xoutu sc] xesacu
          [phrase]  [phrase]  [phrase]  xfor  [xiddec]  loop
          [phrase]*  xforall  [xiddec]*  loop

loop    = [xwhile  sc]  xdo  sc  xod

sc      = phrase  [separator  phrase]*

enq     = phrase  [xsemi  phrase]*

phrase  = primary  [primary]*

primary = XLOAD
          XOPER
          XIDDEC
          XLABDEC
          XROUTINE  phrase  xroutinend
          cclause

separator = xsemi
            xexit
```

5.2.1.3 Skeleton translator — stage 1

```
XCONTROL currentxc;

PROC read = OUTPUT:  "deliver next imperative";


PROC phrase = VOID:
BEGIN
       DO  CASE read
           IN  (XLOAD):
                   ----,
               (XOPER):
                   ----,
               (XIDDEC):
                   ----,
               (XLABDEC):
                   -----,
               (XROUTINE):
                   phrase,
               (XCONTROL xc):
                   (currentxc := xc;
                    IF  fn OF xc = "cclause starter"
                    THEN  cclause
                    ELIF  fn OF xc = "phrase terminator"
                    THEN  GOTO out
                    FI
                   )
```

```
            ESAC
        OD;
out:    SKIP
END;


PROC sc = VOID:
WHILE  phrase;
       fn OF currentxc = xsemi  OREL  fn OF currentxc = xexit
DO  SKIP  OD;


PROC cclause = VOID:
IF  fn OF currentxc = xbegin
THEN  sc
ELIF  fn OF currentxc = xif
THEN  sc;
      sc;
      IF  fn OF currentxc = xelse  THEN  sc  FI
ELIF  fn OF currentxc = xcase
THEN  sc;
      phrase
      WHILE  fn OF currentxc = xcomma  DO  phrase  OD;
      IF  fn OF currentxc = xout  THEN  phrase  FI
ELIF  fn OF currentxc = xcoll
THEN  phrase;
      WHILE  fn OF currentxc = xcollcomma  DO  phrase  OD
ELSE  "see stage 2 for other controls"
FI;
```

5.3 The Reverse Polish Stack

In a reverse Polish language, operands "lie dormant" until an operation on them is specified, but it does not necessarily follow that the translator must maintain a reverse Polish stack. For certain types of machine it would be feasible to generate object code from each `XLOAD` immediately it appeared, and keep no record in the translator. For most types of machine, however, it will be desirable to be able to delay the production of code for operand settings at least until all the operands are present and an actual operation has been specified. This delay is a simple technique for optimisation; decisions about the best form of code cannot always be made when a value is first placed on the reverse Polish stack. Even after an `XOPER` has appeared, it may be desirable to hold up the production of code. For example, when translating for a single address machine, it has been found best to delay the operations of dereferencing, field selection and array indexing. When this is done, the resulting value on the reverse Polish stack has to embody enough information to enable the required address calculations and code production to be carried out later.

In Algol 68, a reverse Polish stack can be implemented very simply. Stage 2 of the skeleton translator given below includes the declaration of a stack reference variable `vss` local to the phrase procedure. More will be said about this localisation at a later stage, but as the translator is highly recursive, it implies that local sections of the overall reverse Polish stack are distributed in the procedural stack of the translator. A new value must be constructed and put on the local stack at each `XLOAD` imperative and for the result of each closed clause. In Algol 68, a closed clause always delivers its result to the phrase which contains it, and the translator's value for this

result must therefore be loaded on the stack local to the phrase. In fact, the value is constructed (at least in part) as soon as the XCONTROL announces the start of a closed clause. Part of the value will be the mode of the result of the closed clause, which is given in the XCONTROL imperative. Another part will be the location used for the result in the object machine. This may or may not be allocated immediately, but in any case the procedure cclause will require access to is for code generation purposes. The value is therefore passed from phrase to cclause as a parameter, and we have made it a REF VALUE parameter to permit communication in either direction. In the skeleton, this happens after the condition m OF xc > 0, which should be assumed for the time being to be TRUE. The REF VALUE is passed from cclause to every subservient procedure, including recursive calls of phrase.

A phrase delivers a result, possibly void, but always leaving a single value on its reverse Polish stack. If ths phrase is terminated by a semicolon, the result is written off (by the rules of Algol 68) and this is mirrored in the translator by the disappearance of the stack on exit from the procedure. However, if the phrase is the last in a serial clause - or is terminated by an EXIT in the source text — the result must be preserved by the object code for future use. A value already exists for it on the stack in an outer phrase and is accessible as the answer parameter of the inner phrase under discussion. The last act before exit from phrase is therefore to generate code, as necessary, to ensure that the location of the result agrees with that in answer. The local stack then serves no further purpose and can safely be allowed to pass out of scope.

The test m OF xc > 0 is an optimising device to deal with a closed clause whose result is known (by the compiler) to be the result also of an outer serial clause — described in the answer parameter of the current phrase. Thus at the arrow in

```
(a;  (b;  c))
     ^
```

a value for the result of (b; c) has already been constructed at the beginning of the outer serial clause, and can simply be passed on. To indicate this optimisation, which also has implications for code generation, the compiler sets the m field of the opening XCONTROL at the arrow negative. When this is detected in the translator, it calls cclause(answer) instead of cclause("newly constructed value"). After the first closing bracket in the above example, the stack local to the phrase (b; c) will be NIL, because nothing will have been loaded. No action must then be taken at the point in the skeleton labelled result, for it will already have been taken on exit from the phrase c.

In expanding stage 1 of the skeleton to make stage 2, we have expressed the action for an XROUTINE as a call of the procedure routine. Although the body of this procedure is trivial in the skeleton, it will of course be much larger in an actual translator. The call of routine in the translator corresponds to a routine-text in the program, not a routine call. The body of the routine-text is not being obeyed, and the answer parameter for its phrase is therefore NIL. Inside That phrase, the NIL should be detected at the label result.

5.3.0.1 Skeleton translator — stage 2

```
XCONTROL currentxc;

PROC read = OUTPUT:  "deliver next imperative";

MODE VALUELIST = STRUCT (VALUE v, REF VALUELIST rest);


PROC phrase = (REF VALUE answer) VOID:
BEGIN
       REF VALUELIST vss := NIL;
       DO  CASE  read
```

```
            IN  (XLOAD):
                    vss := LOC VALUELIST := ("given value", vss),
                (XOPER):
                    ----,
                (XIDDEC):
                    ----,
                (XLABDEC):
                    ----,
                (XROUTINE):
                    routine,
                (XCONTROL xc):
                    (currentxc := xc;
                     IF  fn OF xc = xwhile
                             OREL  fn OF xc = xdo ANDTH "no xwhile"
                     THEN  vss := LOC VALUELIST
                               := ("value for void result", vss);
                           cclause(v OF vss)
                     ELIF  fn OF xc = "cclause starter"
                     THEN  IF  m OF xc > 0
                           THEN  vss := LOC VALUELIST
                                     := ("value for result", vss);
                                 cclause(v OF vss)
                           ELSE  cclause(answer)
                           FI
                     ELIF  fn OF xc = xsemi
                     THEN  GOTO out
                     ELIF  fn OF xc = xfor  OREL  fn OF xc = xforall
                     THEN  "read and unstack information as far
                            as xwhile or xdo"
                     ELIF  fn OF xc = "any other phrase terminator"
                     THEN  GOTO result
                     FI
                    )
            ESAC
        OD;

result:  "ensure that, if vss isn't NIL, the object described by
          v OF vss is properly described by answer";
out:   SKIP
END;

PROC routine = VOID:  phrase(NIL);


PROC sc = (REF VALUE answer) VOID:
WHILE  phrase(answer);
       fn OF currentxc = xsemi  OREL  fn OF currentxc = xexit
DO  SKIP  OD;


PROC cclause = (REF VALUE answer) VOID:
IF  fn OF currentxc = xbegin
```

```
THEN  sc(answer)
ELIF  fn OF currentxc = xif
THEN  sc("value for boolean");
      sc(answer);
      IF  fn OF currentxc = xelse THEN sc(answer) FI
ELIF  fn OF currentxc = xcase
THEN  sc("value for integer");
      phrase(answer);
      WHILE  fn OF currentxc = xcomma
      DO  phrase(answer)  OD;
      IF  fn OF currentxc = xout  THEN  sc(answer)  FI
ELIF  fn OF currentxc = xcaseu
THEN  sc("value for union");
      phrase(answer);
      WHILE  fn OF currentxc = xcommau
      DO  phrase(answer)  OD;
      IF  fn OF currentxc = xoutu  THEN  sc(answer)  FI
ELIF  fn OF currentxc = xcoll
THEN  phrase("value for first field of answer");
      FOR i  FROM 2
      WHILE  fn OF currentxc = collcomma
      DO  phrase("value for ith field of answer")  OD
ELSE  CO  xwhile or xdo  CO
      IF  fn OF currentxc = xwhile
      THEN  sc("value for boolean")
      FI;
      sc(answer)
FI
```

6 Stream language in detail

This chapter completes the account of stream language by supplying all the detailed factual information. The hierarchy of imperatives is organised in two ways, partly by the use of unions (as XIDDEC, XROUTINE and XLABDEC are united under the mode XDEC), and partly by characteristic integers (as for example the fn field of an XCONTROL imperative). In this text, mnemonics are used in place of actual integer values, which can be discovered from the listing of the RS compiler.

The word "mode" frequently occurs as the selector of an integer field in an imperative. The integer should always be understood as an index to the vector of modes given in the REF VECTOR [] MDE imperative.

Certain fields of the imperative are present for the convenience of the compiler itself. Such fields will be enclosed in curly brackets; their significance for a translator is nil, and they should not be output by the compiler shell.

Some features of stream language may vary slightly between different implementations of the RS compiling system. See Appendix 5 for the list of implementation-dependent declarations in the text of the compiler.

6.1 The Vector Of Modes — REF VECTOR [] MDE modes

Each different mode used in a program is represented in stream language by a mode number. This is the sum of an integer, m say, representing a non-ref mode, and an offset (refmark) for each REF at its front. The actual source mode represented by m is held in the element modes[m] which yields a MDE. This is a union of constituent modes representing structures, procedures etc, as given in the table below. These constituent modes contain further mode numbers which can be similarly decomposed until PRIMITIVE constituents are reached. The mode number for all the PRIMITIVE modes are given in Appendix 2, and are fixed for all programs. Once these mode numbers have been reached, the mode is completely known. However, PRIMITIVE (defined to be INT) gives "type" information on the primitive mode, defined as the mode number for the primitive mode stripped of any SHORT or LONG prefixes.

6.1.1 Constituent Modes Of MDE

The mode MODELIST, defined as

```
MODE  MODELIST  =  STRUCT (INT mode, REF MODELIST rest)
```

is used in the following table, although it is not itself a constituent of MDE.

6.1.2 REF STRCT

```
MODE  STRCT  =  STRUCT (INT {rdenno}, {deflex}, REF SELIST sels),
```

in which the mode SELIST is defined as

```
MODE  SELIST  =  STRUCT (INT mode, fieldno, ID name, REF SELIST rest ).
```

This gives the mode and field number, starting with field 1, for each of the fields of a structure. The name field gives the field selector name, truncated or space-filled to maxid characters (see Appendix 5), internal spaces having been removed.

6.1.3 REF ISTRUCT

```
MODE  ISTRUCT  =  STRUCT (INT {rdenno}, imode, length, {deflex}).
```

This describes an indexable structure having length elements of mode imode.

6.1.4 REF VCTOR

```
MODE  VCTOR  =  STRUCT (INT {rdenno}, vecmode, deflex)
```

This mode describes a vector or flex vector with elements of mode `vecmode`. The `deflex` field is negative for a flex vector and positive or zero for a non-flex vector.

6.1.5 REF ARRAY

```
MODE  ARRAY  =  STRUCT (INT {rdenno}, mode, nods, deflex).
```

This describes an array or flex array of `nods` dimensions; `mode` is the mode of the array with the front row removed (ie the mode of the elements for a 1-dimensional array). The `deflex` field is negative for a flex array and positive or zero for a non-flex array.

For example

- `[] REAL` is represented in `mode` field as `REAL`
- `[ , ] REAL` is represented in `mode` field as `[]REAL`
- `FLEX [ , ] REAL` is represented in `mode` field as `[] REAL`

6.1.6 REF UNN

```
MODE  UNN  =  STRUCT (INT {rdenno}, REF MODELIST modelist)
```

This describes a union mode, where `modelist` is a list of all the constituents (any unions within the union having been decomposed).

6.1.7 REF PROCP

```
MODE  PROCP  =  STRUCT (INT deproc, REF MODELIST pars)
```

Describes a procedure with parameters; `deproc` is the mode of the result and `pars` gives the modes of the parameters.

6.1.8 REF PRC

```
MODE  PRC  =  STRUCT (INT deproc)
```

Describes a procedure with no parameters and result of mode `deproc`.

6.1.9 REF STEN

```
MODE  STEN  =  STRUCT (INT mode, REF STENLIST {stenlist})
```

Describes a straight of objects of mode `mode`.

6.1.10 REF AMODE

No reference to this mode will occur in stream language.

6.1.11 SAMEAS

Except in the table of modes, no reference to this mode will occur in stream language.

6.1.12 PRIMITIVE

```
MODE  PRIMITIVE  =  INT;
```

The `MDE` vector contains elements for all the primitive modes, which always occupy fixed places at the bottom of the vector, as given in Appendix 2. The index, ie the mode number, therefore determines the primitive mode uniquely. When a `MDE` element decomposes to `PRIMITIVE`, the integer value specifies the type of the mode, defined as the primitive mode with any `SHORT`s or `LONG`s removed. For example, the `MDE` element `modes[20] (LONG REAL)` will decompose to `PRIMITIVE` and have value 19 (`REAL`).

6.2 Identifier Declarations (XIDDEC from XDEC)

These imperatives introduce declaration numbers in the decno series and also imply storage allocation for the object machine. The definition of an identifier in the source text gives rise to an XIDDEC (except in the case of routine identity declarations covered by XROUTINE In B3.3). The mode XIDDEC is given by

```
MODE  XIDDEC  =  STRUCT (INT type, REF IDDEC iddec)
```

The iddec field refers to information in the following structure

```
MODE  IDDEC  =  STRUCT (ID name, INT decno, {level}, mode,
                        INT {scope}, REF IDDEC {rest} )
```

The name field gives the source text identifier or operator symbol (truncated or space-filled to maxid characters, internal spaces having been removed), and mode gives its mode. The declaration number decno is used to index a vector in which the translator can keep a record of values constructed in response to the XIDDEC. The decnos start at 4.

The type field of an XIDDEC is one of the following

xiddec Identity declaration. When this imperative occurs, the top item on the reverse Polish stack (which must be removed) will be the value for a static object or the static part of a dynamic object. Code must be generated, if necessary, to preserve this object, and a value constructed to describe it.

xvardec Reference declaration for a static object. Space must be found for the object and a value constructed to describe the reference to it.

xivardec Initialised reference declaration. The top item on the reverse Polish stack (which must be removed) will be the initial value for a static object or the static part of a dynamic object. Space must be found for this, and a value constructed to describe the reference.

xfdec Specifies a formal parameter of a procedure.

xccdec Introduces the formal identifier in a conformity.

xfordec Introduces the identifier after FOR in a loop clause.

xforalldec
Introduces an identifier defined in a FORALL statement.

xdummydec
A pseudo declaration introduced by the compiler in order to indicate the scopes of some routines.

6.3 Routine Text Declarations, XROUTINE

Every occurrence of a routine text (which may appear at any point in stream language) gives rise to an opening XROUTINE imperative, as also do routines invented by the compiler for certain tasks. A routine text is terminated by the XCONTROL with function xroutinend. The mode XROUTINE is given by

```
MODE  XROUTINE  =  REF RDEN,
```

where

```
MODE  RDEN  =  STRUCT (ID name, BITS props, INT mode, rdenno,
                       INT maxname, {level}, REF RDEN {rest} )
```

The name field contains one of the following

the identifier
from a brief declaration of a procedure or operator, as shown below

```
PROC  p  =  routine text
```

```
OP  P  =  routine text
```

Such declarations do not give rise to `XIDDEC` imperatives. The identifier is truncated or space-filled to `maxid` characters, internal spaces having been removed.

`" anonymous "`
: when the Algol 68 routine text was not the subject of a brief declaration. The routine may have been declared fully, as in

```
PROC (INT) INT p  =  (INT n) INT: ... ,
```

in which case an `XIDDEC` imperative is produced for `p`.

`" generator "`
: for a synthetic space generation routine (ie invented by the compiler).

`" assignment "`
: for a synthetic assignment routine.

`" straight "`
: for a synthetic routine for indexing a straight.

`" format "`
: for a synthetic routine containing the bodies of `n`, `f` and `g` patterns from a format.

The `props` field is made up as a conjunction of bits values of the form *2**n*. These are denoted mnemonically (eg `ccbit`) and represent attributes of a particular routine-text.

The bits values are

`ccbit` routine has a body consisting of a closed clause

`operatorbit`
: routine is part of a brief operator declaration

`valbit` routine is to be loaded onto the translator stack, despite the absence of an `XLOAD` imperative

`holebit` routine contains a `HERE` clause (see Part C)

`globscopebit`
: routine is a straightening or assignment procedure with no non-locals

`genprocbit`
: routine is a synthetic generation routine

The `mode` field is the mode of the routine and its declaration number is `rdenno`. Declaration numbers for routines are in a separate series which starts at `startrd + 1` (see Appendix 5).

The field `maxname` is less than 3 for a routine text of unlimited lifetime and 3 if its only external identifiers are declared in modules compiled at `CONTEXT VOID` (See C2.10). Otherwise, `maxname` is the `decno` of an identifier of smallest lifetime used within the routine but not declared in it. This, in combination with information given in any `XLABDEC`s at the beginning of the routine, can be used to determine the lifetime of the routine. However, `XLABDEC` imperatives are not given at the start of synthetic routines.

6.4 Label Declarations (`XLABDEC` from `XDEC`)

In Algol 68, label identifiers can be re-used for different label settings, but in stream language each different label setting has a different label number (`Labno`). `XLABDEC` imperatives occur at positions where labels must be set and also at the beginnings of serial clauses and routines.

The mode `XLABDEC` is given by

```
MODE  XLABDEC  =  STRUCT (REF LABEL lab, BOOL notsetting),
```

where

```
MODE  LABEL  =  STRUCT (ID name, INT labno, status, REF LABEL {rest} )
```

At an actual label setting, `notsetting` is `FALSE` and `status` is undefined. All other occurrences of an `XLABDEC` are at the beginnings of serial clauses or routines, and `notsetting` is `TRUE`. The purpose of the imperative then depends on the value of `Status`. If `status` is 0, the `XLABDEC` is at the start of a serial clause and gives the `labno` of a label which will be set in it. If `status` is `s`, where `s > 1`, the `labno` is to be taken as a new label number for the label previously numbered `s`. If `status` is 1, the `XLABDEC` is at the start of a user-written routine, and gives the `labno` of an external label in a `GOTO label` occurring in the body of the routine. This information is required for defining the scope of the routine.

The field `name` is the source-text identifier, truncated or space-filled to `maxid` characters, internal spaces having been removed.

6.5 The Loading Imperative, XLOAD, And XCHARS

An `XLOAD` imperative requires a value to be loaded on the reverse Polish stack. The mode XLOAD is a union of the following modes.

6.5.1 BOOL

This derives from a boolean denotation in the source-text, and the value will be `TRUE` for `TRUE` and `FALSE` for `FALSE`.

6.5.2 INT

The integer is the `decno` or `rdenno` of a previously declared value to be loaded.

6.5.3 REF LABEL

This is the `lab` field of an `XLABDEC` imperative. The value corresponding to the `labno` must be loaded.

6.5.4 STRUCT (INT Nse)

A value for `NIL`, `SKIP` or `EMPTY`, according as `Nse` is `nilmode`, `skipmode` or `voidmode`.

6.5.5 XGEN = STRUCT (INT mode, BOOL loc)

The `mode` field will be a ref to the mode of a static object. Space must be dynamically generated for the object, locally or on the heap according as `loc` is `TRUE` or `FALSE`. A value must be loaded to describe the reference.

6.5.6 XNUMBER = STRUCT (INT mode, REF VECTOR [] CHAR nu)

This imperative represents a number denotation in the source-text. The `mode` field will be `bits`, `int` or `real` (possibly with `LONG` or `SHORT` prefixes) and the `nu` field refers to the actual denotation in a standard format as follows.

For a value of mode `BITS`, the format is:

```
radix digits
```

where `radix` is the character '`2`', '`4`', '`8`' or '`g`' (for sixteen)[1] and digits is a digit sequence using the letters '`a`' to '`f`' for the digits ten to fifteen as required in hexadecimal numbers.

[1] The characters dchar, nchar and pchar are implementation dependent (see Appendix 5). These, together with the characters used at the start of denotations, originate from literals in the compiler. By contrast, other digits, together with the letters a to f used in hexadecimals, belong to the user's source-text character set as specified by the compiler's charset parameter. Care should be taken that, during bootstrapping, the two character sets are the same.

For an integer, the format is:

```
a  digits     (where the character "a" means radix 10)*
```

For a real, the format is:

```
r  [digits]  [dchar digits]  [signletter digits]
```

The square brackets (not part of the denotation) indicate parts which may be absent, though at least one will be present. The character `r` indicates "real", `dchar` denotes decimal point, and `signletter` is the character `pchar` for plus or `nchar` for minus*. The digits after this are the exponent.

There are no space characters in the above number representations.

The following imperatives are each followed by one or more `OUTPUT(XCHARS)` imperatives giving the relevant source-text quotations (see `XCHARS`).

6.5.7 XSTRING = STRUCT (INT strmode)

For a string denotation containing one character only, `strmode` is `char`. For a denotation containing `n` characters (`n /= 1`), `strmode` is `STRUCT n CHAR` (including when `n` is zero).

6.5.8 XFORMAT = STRUCT (INT nochars, nocases, w)

The text of a format denotation is introduced by an `XLOAD(XFORMAT)` imperative, after which there are one or mode `XCHARS` imperatives, each corresponding to a line of the source-text. If there are any `n`, `f` or `g(uc)` patterns in the format, The `XFORMAT` is preceded by the loading of a routine text, whose body consists of a case-clause. The cases are the enclosed clauses from the `n` and `f` patterns, together with any unitary clauses used in `g` patterns. The order is identical to that of the source-text.

`nochars` is the number of chars in the succeeding `XCHARS` imperatives. `nocases` is the number of `n`, `f` or `g(uc)` patterns in the format. `w` is the maximum nesting depth of collection lists in formats.

For example, `w = 2` for `$ 2 (G, 3 ("LINE" A L)) $`, `w = 0` for `$ d $`

6.5.9 XALIEN = STRUCT (INT almode)

Describes the source-text construction

```
ALIEN "insertion"
```

which occurs on the right of an identity declaration. The field `almode` gives the mode, and succeeding `XCHARS` imperatives contain the insertion.

6.5.10 XCODE = STRUCT (INT mode, nopars)

A source-text code insertion has the form

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

where mode is optional, absence implying `VOID`. Stream language for the unitary clauses (uncs) comes first, and their results will be on the stack when the `XLOAD(XCODE)` imperative is given. The `mode` field gives the mode and `nopars` the number of uncs. The succeeding `XCHARS` imperatives give the code.

6.5.11 XCHARS = STRUCT (INT nochars, base, REF VECTOR [] CHAR chars)

This imperative is not a constituent of the mode `XLOAD`, but is included in this section because it always follows an `XLOAD` or a previous `XCHARS` of which it is a continuation.

Successive `XCHARS` imperatives represent the successive instalments of one source-text string or format denotation, which may have been broken, for instance by new lines. In the case of a format, these are the only source of breaks. In the case of strings, a new instalment occurs

when, in the source-text, a closing quote character is followed, after spaces or new lines or a new radix, by an opening quote character. However, any instalments containing no characters are skipped over, unless final.

`nochars` For the final instalment, this is `UPB chars` (and may of course be 0). For all preceding instalments, it is `- UPB chars` (/= 0).

`base` f the `chars` field contains a "radix string", the `base` field gives the base, which will be 2, 4, 8, 10 or 16. Otherwise, the base is 0. In particular, it is 0 for a format denotation.

`chars` n string denotations (following `XSTRING`, `XALIEN` or `XCODE`), the enclosing quote characters in the source-text are excluded. Any doubled quote within the source-text denotation appears as a single quote character in `chars`. However, these remarks do not apply to string denotations inside formats, which are reproduced literally.

In format denotations, the opening `$` is replaced by a space but the closing `$` is included. Otherwise, spaces are removed except where meaningful. Thus if two successive string denotations within a format are separated in the source-text by one or more spaces, one space is retained. Similarly, if the strings are separated by a new line in the source-text, the break will introduce a space as the final character in the `chars` field of the `XCHARS` which finishes at the new line. The format will then continue in a new `XCHARS`. Thus, when the format interpreter concatenates the "chars" fields of successive `XCHARS` imperatives, quote characters will not be brought into contact and be misinterpreted as a quote character within a string.

Comments within formats are also removed. The body of an `n` or `f` pattern is replaced by an integer giving its position in the format, eg the second pattern would become `n(2)` or `f(2)`. This applies to clauses in `g` patterns, which are also replaced by `n(x)`, where `x` is an integer giving its position in the format. All three types are numbered from 1 in the same series.

6.6 Operations, XOPER

The mode `XOPER` is defined as

```
MODE  XOPER  =  STRUCT (INT fn, m, param)
```

The `fn` field specifies an operation for which code must usually be generated. The operation applies to one or more objects (operands) for which translator values will exist on the reverse Polish stack. These values must be removed from the stack and replaced by a value for the result of the operation (except for `xparampack` which gives no result).

The `m` field gives the mode of the result, unless otherwise stated.

The `param` field is used for additional information, but if not mentioned in the tables below, it can be assumed undefined.

6.6.1 Standard prelude operators

`xmonop` 1 operand. `param` = 16 * opnumber `+` version number. The operators and corresponding opnumbers are listed in Appendix 3 (monadic operators). A given operator has different versions for different modes of operand.

`xdyop` 2 operands. `param` = 16 * opnumber `+` version number. The opnumbers and version numbers are given in Appendix 3 (dyadic operators). The mode given at the head of the table applies to one operand and is sufficient to identify what is required for a particular version. Versions for arithmetic and relational operators between ints, reals and complexes need only deal with operands of like mode, as the compiler will supply a widening coercion for one operand where necessary.

6.6.2 Coercions and similar operations

`xderef` 1 operand, an object to be dereferenced.

`xunite` 1 operand, an object to be united. The mode of the object will not itself be a union; it will be the `param`th mode (starting at 1) of the union `m`.

`xuniteu` 1 operand, an object having a union mode, to be united in mode `m`.

`xdeunite` 1 operand, a ref union to become a ref to its current constituent mode, but flexed as given by `m`.

`xwrc*` 1 operand, a real to be widened to complex.

`xwir*` 1 operand, an int to be widened to real.

`xwbvb*` 1 operand, a bits to be widened to vector of bool.

`xarr` 1 operand, a (ref) M to become a (ref) array of M.

`xarrarr` 1 operand, a (ref) array to become a (ref) array with an extra dimension.

`xvecarr` 1 operand, a (ref) vector to become a one-dimensional (ref) array.

`xisarr` 1 operand, an object of mode

```
(REF) STRUCT i STRUCT j STRUCT k  ...  M
```

where `param` is the number of i-structs before the mode M. The object is to be coerced to a param-dimensional (ref) array.

`xvec` 1 operand, a (ref) m to become a (ref) vector of `m`.

`xisvec` 1 operand, a (ref) i-struct to become a (ref) vector.

`xis` 1 operand, an object to become a `STRUCT 1 X` or a `REF STRUCT 1 X` as given by `m`. The mode of the operand is `X` in the former case and `REF X` in the latter.

`xniltom` 1 operand, a `NIL` to be coerced to mode `m`.

`xvac` 1 operand, an `EMPTY` to be coerced to a vector or array.

`xmtoxtype`
1 operand, a (ref) m to be coerced to (ref) xtype.

`xytypetom`
1 operand, a (ref) ytype to be coerced to mode m.

`xskiptom` 1 operand, a `SKIP` to be coerced to mode `m` (`m` /= `voidmode`).

`xgotoproc`
1 operand, a jump to be procedured to mode `m`.

`xgotom` 1 operand, a jump to be coerced to mode `m`.

`xvoid` 1 operand to be coerced to `voidmode`.

* The mode of the operand may be preceded by `SHORT` or `LONG` prefixes, in which case the mode of the result also starts in this way.

6.6.3 Field selection and array indexing

`xselect` 1 operand, a structure or array of structures from which the `param`th field must be selected (starting from 1).

`xsimpleindex`
1 `+ param` operands comprising an array or vector and param subscripts (1 for a vector) to produce a single array or vector element.

`xtrimindex`
: 1 + `param` operands comprising an array or vector and param trimscripts (1 for a vector) to produce a subset of the array or vector. If the operands are a vector and a trimmer containing `AT`, The result is an array.

`xtrim`
: `param` operands comprising none, some or all of `a`, `b` and `c` in `a:b AT c` to produce (a value for) the trimmer. `m` = 1 if `a` is present + 2 if `b` is present + 4 if `AT c` is present + 8 if lower bound needs setting (to 1 or `c`)

Note: when selection or indexing operations are applied to a (single) reference, the result is a reference.

6.6.4 Procedure calls

`xparampack`
: `param` operands comprising the actual parameters of a procedure which is about to be called (if already on the stack) or loaded and then called (with no intervening imperative). The values for the parameters must be removed from the stack, and there is no resulting value to be put on.

  `m` = the mode of the procedure for which the operands are the actual parameters.

`xcall`
: 1 operand, a procedure. The procedure must be called and a value for its result (mode `m`) put on the stack.

  `param` =

0	when the procedure to be called is not a generator routine or an operator
1	for a call of an invented routine which generates dynamic local space
2	for a call of an invented routine which generates dynamic space on the heap
3	for a call of an invented generator routine occurring within another such generator routine
4	for a call of a user-defined (or library) operator

6.6.5 Assignment

`xassign`
: 2 operands, the destination and source for a simple assignment operation, which must be carried out leaving the value for the destination on the stack. For assignments of fixed-size objects, `param` is 1. For vector or array assignments (restricted to one dimension), the source will be a descriptor. If the destination mode is a ref flex array or ref flex vector, `param` will be 1 and code must be generated to assign the descriptor only. If the destination mode is a non-flex ref vector or ref array, `param` will be 2, and code must be generated to assign the elements, which will be objects of fixed size.

6.6.6 Space finding

`xbdpack`
: `param` operands, which, if even in number, comprise the lower and upper bounds for each dimension of an array; if `param` is 1, the operand is the upper bound of a vector (for which the lower bound is always 1). The operands on the stack are to be replaced by a value for the bound pack, used by `xdyngrab` Below. For the operation `xbdpack`, `m` is undefined.

`xdyngrab`
: [always following `xbdpack` and load of a bool value inside an invented routine] 2 operands, a bound pack and a value for a bool. Code must be produced to generate

the space for the elements of an array or vector, locally or on the heap according as the boolean at run time is true or false. The operation `xdyngrab` delivers the descriptor of the array or vector, and `m` is its mode.

`xstatgrab`

1 operand, the static part of a generated dynamic object of mode `X`, where `M = REF X`. Code must be produced to create the space (if necessary) for the static part of the object, assign the operand to it and deliver the reference. `param = ABS TRUE` or `ABS FALSE` for local or heap generation respectively.

`xcopy` [preparatory to `xassign` for a flex vector or flex array] 1 operand, the descriptor of an array of any dimensions, or the descriptor of a vector. Code must be produced to generate heap space and copy the elements of the array or vector into it. The operation delivers the descriptor; `m` is the corresponding mode and `param = ABS FALSE`.

`xdefaultbd`

No operand. The operation is to tuck, under the top item on the stack, a value for a default lower bound of 1, arising from constructions like `[n] INT` in the source text. `m` is undefined.

6.6.7 Straightening

`xprestraight`

[preparatory to `xstraight` whenever necessary] 1 operand, an object to be adjusted to have mode `m` in readiness for straightening, which is carried out by the operation `xstraight` described below. The mode adjustment will entail provision of an initial ref, if the original object was not a reference, and will introduce `FLEX` as given in `m`, if not already present. The resulting object, for which a value must be put on the stack, need exist only within the current scope.

`xstraight`

2 operands, the `composite` and `index` fields for the `STRUCT` displayed below. A straight of mode `m` must be constructed and a value for it put on the reverse Polish stack. The `param` field gives information about the object to be straightened and is used as follows

1	union
number of elements	struct or i-struct
-1	vector
-1-n	array of n dimensions

`xstrindex`

2 operands, a straight, `s`, and an integer, `i` (say). The operation gives rise to the procedure call

```
(index OF s)(i, composite OF s)
```

The result of this procedure is the `i`th member of the straight `s`, having mode `m`, and a value for it must be put on the reverse Polish stack.

A straight, of mode `STRAIGHT X` (say), is a descriptor — distinct from an array descriptor — containing a reference to the original composite object. It also contains a procedure which is provided by the compiler for indexing, and an integer for the number of members of the straight. Thus a straight descriptor is likely to have the form

```
STRUCT (REF COMP composite, PROC (INT, REF COMP) X index, INT upb)
```

where COMP is the original row, vector, struct, i-struct or union mode (but with FLEX introduced wherever applicable).

6.7 The Control Imperatives (XCONTROL)

The mode XCONTROL is defined as

```
MODE  XCONTROL  =  STRUCT (INT fn, m, BITS props, INT param);
```

fn — Distinguishes the various XCONTROLs, as listed below, and reflects the structure of Algol 68 in stream language.

m — In an XCONTROL at the start of a serial clause (or an enquiry clause, or the remainder of a serial clause after an EXIT in the source text), the m field gives the mode of the result of the clause. The mode number is negated when the result of the closed clause is also the result of the surrounding serial clause. m is defined only where indicated in the table below.

props — The letters in the table refer to properties detailed on the next section, which are concerned with the implementation of lifetimes and the handling of results. Furthermore, any XCONTROL generated by the compiler has the compgenbit set.

param — Additional information, defined in the following section.

6.7.1 Fields of an XCONTROL

xcase, etc [1]
: m defined; props = a; param = mode of result of closed clause [7]

xin, etc [2]
: m defined;props = a b;param = total number of cases [8]

xcomma, etc [3]
: m defined; props = a b e [6];param = case no of ensuing serial clause

xwhile, xdo [4]
: m defined;props = a; param = mode of result of closed clause

xesac, etc [5]
: props = b; param = undefined

xroutinend
: props = b g; param = undefined

xsemi — props = s; param = undefined

xexit — m defined;props = b;param = first exit in a serial clause is treated like an xin, subsequent exits like xcomma

xfinish — the final imperative from a successful compilation

Notes

1 — closed clause starters: xbegin, xif, xcase, xcaseu, xcoll

2 — xthen, xin, xinu

3 — serial clause separators: xelse, xcomma, xout, xcommau, xoutu, xcollcomma

4 — if xwhile is present, it is the closed clause starter; otherwise xdo is the closed clause starter

5 — closed clause terminators: xend, xfi, xesac, xesacu, xendcoll, xod (which is immediately preceded by an xsemi, so bit b will in this case be absent).

6 this property applies to `xelse`, `xout` and `xoutu` only.

7 the mode number is negated when the result of the closed clause is also the result of the surrounding closed clause

8 in an if clause, then and else count as "cases" 1 and 2; in a case clause with an out part, the `param` field at the `xin` is minus the number of cases between in and out.

6.7.2 The `props` field of an `XCONTROL`

As with the corresponding field of `XROUTINE`, the props field is made up as a conjunction of bits values of the form *2**n*. These are denoted mnemonically (eg `iddbit`), and represent attributes of the context at which they are given. The contexts (a, b, s, e, g) are those given in the above table. Bits not mentioned for a particular context (or bits of the `props` field which have no mnemonics) must be assumed undefined.

6.7.2.1 General preliminary information

The meaning of individual bits is as follows

`priobit` an Algol 68 priority declaration in the source text

`semibit` semicolon

`decbit` an Algol 68 declaration (except priority)

`vardecbit`
an `xvardec` or `xivardec`

`labbit` a label setting

`exitbit` an `EXIT`

`locgenbit`
an explicit local generator

`locdydecbit`
a declaration containing a dynamic part

6.7.2.2 Dynamic result bits

Table `a`

Presence of bit means that the following serial clause contains a discarded dynamic result ...

`dyprocbit`
... from a procedure

`dyvardecbit`
... from a closed clause containing an `xvardec` or an `xivardec`

`dydecbit` ... from a closed clause containing an Algol 68 declaration in the source-text

Table `b`

Presence of bit means ...

`dontpullbit`
... that there is a dynamic result from the preceding serial clause (or part of serial clause before an `EXIT`)

Table `s`

Presence of bit means (at an `xsemi` only) ...

`dontpullbit`
... that the preceding phrase was a declaration

`dyprocbit`
`dyvardecbit`
`dydecbit` ... that a dynamic result can now be discarded unless the `dontpullbit` is also present

6.7.2.3 Routine bits

Table `g`

Presence of bit means that ...

`genprocbit`
: ... the foregoing routine was an invented generator routine

`globscopebit`
: ... the foregoing routine was an invented assignment or straightening routine that used no non-local names

`valbit` ... the foregoing routine is to be loaded onto the reverse Polish stack; there is no `XLOAD` following

\subsubsectionSpecial bit

Table `e`

Presence of bit means that ...

`elifousebit`
: ... `xelse`, `xout` or `xoutu` are derived from elided source-text constructions `ELIF` or `OUSE`

6.7.3 Other control imperatives

\subsubsection`fn` = `xfor`

Introduces a do-statement. Values for the `FROM`, `BY` and `TO` parts (if any) will have been loaded, in that order.

`m` undefined

`props`

1	if an identifier has been given for the loop counter, in which case the next imperative will be an `XIDDEC` for the identifier with type `xfordec`
+2	if a value for `FROM` is present
+4	if a value for `BY` is present
+8	f a value for `TO` is present

`param` gives the number of values on the stack

6.7.3.1 `fn` = `xforall`

Introduces a forall-statement. Values for the arrays or vectors to be sequenced through will have been loaded, in the order in which they were given in the source text.

`m` undefined

`props` undefined

`param` the number of values on the stack

The `xforall` imperative will be followed by a series of `xiddec` imperatives for the identifiers declared by the `forall` statement. Each of these refers to the array or vector value on top of the stack, which must be removed.

6.7.3.2 fn = xuchoice

Occurs immediately after an xinu or xcommau (except for the possibility of intervening labels), and represents the declarer in a choice.

m	mode of the declarer
props	setting of the decbit indicates that a formal identifier has been given, in which case the next imperative will be an XIDDEC for the identifier, with type xccdec
param	the serial number of m in the union mode under test (say n), or -n if m is itself a union

6.8 The XWARN Imperative

The XWARN imperative gives advance warning of various stream language imperatives. Most of these are certain types of dyadic operation, for which the XWARN is given between the loading of the two operands and is additional to the XOPER which will come when both operands have been loaded. XWARN has the form

```
MODE  XWARN  =  STRUCT (INT w)
```

where w takes one of the mnemonic values specified below indicating what operation is being forewarned.

xwass	xassign operation
xwandth	xdyop operation ANDTH (see Appendix 1)
xworel	xdyop operation OREL (see Appendix 1)
xwindex	xsimpleindex, xtrimindex, xtrim and xstrindex operations
xwplusabetc	xdyop operations PLUSAB, MINUSAB, TIMESAB, OVERAB, MODAB and DIVAB
xwforall	forall-statement. The warning occurs after each loading of a vector or an array
xwloop	for-statement. The warning occurs at the beginning, ie before the loading of the FROM, BY or TO parts if present

6.9 The XPRAG Imperative

This is defined as

```
MODE  XPRAG  =  STRUCT (BOOL all, REF VECTOR [] CHAR pr)
```

where pr refers to a vector containing the first line of a pragmat in the source-text. The all field is TRUE if pr contains the whole of the pragmat, otherwise FALSE. If the first line contains an opening comment symbol, pr will contain only the characters preceding the comment symbol.

6.10 The XCHARPOS Imperative

The XCHARPOS imperative allows the shell to keep track of the character position within a line of Algol 68 source-text. It is defined as

```
MODE  XCHARPOS  =  STRUCT (INT charpos)
```

where charpos indicates the character position of a significant symbol. Such information may be required for diagnostic purposes. The XCHARPOS imperative will normally be output so that when read back, it immediately precedes an XCONTROL; the significant symbol will be the source-text analogue of the XCONTROL. However, if the XCONTROL is a closing control, there may be coercions between the XCHARPOS and the XCONTROL.

The imperatives `xfor` and `xforall` are not output at positions corresponding to the `FOR` and `FORALL` symbols; for these the symbol position is output preceding the `XWARN (Xwloop)`. A symbol position is also output before an `XROUTINE` imperative and an `XLABDEC` (where `notsetting = FALSE`).

6.11 An example of stream language

```
 0    PROGRAM  p10
 1    CONTEXT  VOID
 2    BEGIN  INT i;   INT k = 80;
 3           REAL rrr := 2.34e5;
 4           MODE V = VECTOR [3] INT;
 5           V v := (k, i, ENTIER rrr);
 6           FLEX V fv := v;
 7
 8           PROC p = (INT n) INT:
 9           IF  n > 0
10           THEN  n * (v[n] - i)
11           ELSE  0
12           FI;
13
14           p(i := -4)
15    END
16    FINISH
```

When the above program is compiled by the RS compiler, the following stream language is obtained:

6.11.1 Sizes

module contains 130 outputs, 2 routines, 35 modes, 1 labels, 11 identifiers, 1 modules and 0 libinds

6.11.2 Modes

```
31   VECTOR [] INT
32   PROC  ( BOOL ) 31
33   STRUCT  3  INT
34   FLEX  VECTOR [] INT
35   PROC  ( INT )  INT
```

6.11.3 Other imperatives

```
          xtdtype(902)
2:        begin(5, 8r361, 5)
3:        dummydec(0, 4, " anonymous")    vardec(1039, 5, "i")
          semi(0, 8r4000, 0)    number(15, "a80")    iddec(15, 6, "k")
4:        semi(0, 8r4000, 0)    number(19, "r2d34p5")
          ivardec(1043, 7, "rrr")
5:        semi(0, 8r4000, 0)
          vardec(1056, 8, " anonymous")
          load(8)      warn-assign

          routine(1001, 32, 32778, 8, " generator")
          fdec(7, 9, " anonymous")
          begin(31, 8r420, 31)    number(15, "a3")    bdpack(0, 1)
```

```
          load(9)    dyngrab(31, 0)
          end(0, 8r4400, 31)
          endrd(0, 8r100410, 0)

          assign(1056, 1)    void(5, 0)    semi(0, 8r400, 0)
6:        semi(0, 8r4000, 0)    load(8)    deref(32, 0)    true
          ppack(32, 1)    call(31, 1)
          ivardec(1055, 9, "v")
          semi(0, 8r4000, 0)    load(9)    warn-assign
          coll(15, 8r41, 33)    load(6)
          collcomma(15, 8r41, 2)    load(5)    deref(15, 0)
          collcomma(15, 8r41, 3)    load(7)    deref(19, 0)
          monop(15, 209)
          endcoll(0, 8r0, 33)    isvec(31, 0)    assign(1055, 2)
          void(5, 0)    semi(0, 8r0, 0)
7:        semi(0, 8r4000, 0)    load(8)    deref(32, 0)    true
          ppack(32, 1)
          call(34, 1)
          ivardec(1058, 10, "fv")
          semi(0, 8r4000, 0)    load(10)    warn-assign    load(9)
          deref(31, 0)    copy(31, 0)    assign(1058, 1)    void(5, 0)
          semi(0, 8r0, 0)
9:        semi(0, 8r4000, 0)

          routine(1002, 35, 2, 9, "p")
          fdec(15, 11, "n")
10:       if(7, 8r41, 15)    load(11)
11:       number(15, "a0")    diop(7, 114)
          then(15, 8r41, 2)    load(11)    load(9)    warn-index
          load(11)    simpleindex(1039, 1)    deref(15, 0)    load(5)
          deref(15, 0)
          diop(15, 17)
12:       diop(15, 194)
          else(15, 8r41, 2)    number(15, "a0")
13:       fi(0, 8r0, 15)
          endrd(0, 8r0, 0)

15:       semi(0, 8r4000, 0)    load(1002)    load(5)    warn-assign
          number(15, "a4")    monop(15, 17)    assign(1039, 1)
          deref(15, 0)
16:       ppack(35, 1)    call(15, 0)    void(5, 0)
          end(0, 8r0, 5)
          finish(0, 8r0, 0)
```

The above is merely a visual representation of the output from a 'standard' shell. The `XEDIT` imperatives have been absorbed in order to produce a continuous series of other imperatives.

7 Introduction

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 other composition modules.

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.

8 The Source Language

8.1 keeplists

Interaction between modules demands that source-text indicators (identifiers, modenames 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.

8.2 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 is not enclosed in `BEGIN-END` brackets but is introduced by a colon. It 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 or user-defined operators may be called, except within routine texts. Also, no labels may be declared in the outermost level. 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 routine text. 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.

8.3 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
```

8.4 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 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 C2.3, 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 C2.7.)

Example of nesting

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

PROGRAM  insert
CONTEXT  detail IN frame
BEGIN  FORM f := g2;
```

```
        INT n := CONV f;
        - - -
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
```

8.5 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 starting 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 on the right a further nest or the name of a program module.

8.5.1 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 that specified in its outermost cc module.

8.6 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.

8.7 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 myprog
CONTEXT userprog IN package1
closed clause
FINISH
```

with

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

```
FINISH
```

as the composition.

Now consider writing a program, `ourprog` say, which requires the services of two packages, designed independently along the lines of `package1`. The question will be, in what context to compile `ourprog`? 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 C2.6. 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.

8.8 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 C2.2 earlier. No kept item which is a reference (or a structure, array or union containing a reference) may be used except within routine texts. 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 `place IN progtitle`) 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.

8.9 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 %stdprelude
```

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

```
PROGRAM pmw
```

```
closed clause
FINISH
```

can only be run when nested in `%stdprelude`. The intention is that the necessary composition should be effected automatically. The `%stdprelude` 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 2.10.)

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.

8.10 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 C2.7 (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.

```
PROGRAM (prog) ownprelude
CONTEXT VOID
closed clause
FINISH
```

can accept cc keeplists from nowhere.

```
PROGRAM (a1) a
CONTEXT prog IN ownprelude
closed clause
FINISH
```

can accept cc keeplists from `prog IN ownprelude`.

```
PROGRAM (b1) b
CONTEXT a1 IN a
closed clause
FINISH
```

can accept cc keeplists from `prog IN ownprelude` and `a1 IN a`.

```
PROGRAM c
CONTEXT b1 IN b
closed clause
FINISH
```

can accept cc keeplists from `prog IN ownprelude` and `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 in C2.8, and 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 modify `%stdprelude` if they wish, without losing any of its special properties.

8.11 Summary Of Syntax And Semantics Of Modules

8.11.1 `DECS` Module

```
DECS decstitle
[ CONTEXT place IN progtitle ]*
[ USE decstitlelist ]
:
body
KEEP keeplist
FINISH
```

Except within routine texts, the body must not use any externally declared references or call any procedures or user-defined operators.

8.11.2 `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
```

8.11.3 Notes

`decstitle`, `place` and `progtitle` all stand for identifiers. An itemlist is a sequence of items with comma separation. For the definition of a keeplist, see C2.1. For definition of nest, see C2.5.

* Omission of an explicit context introduces the default context:

  ```
  CONTEXT %program IN %stdprelude
  ```

For absolutely no context at all, `CONTEXT VOID` must be written.

8.12 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.

8.13 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 (in `USE decstitlelist`), unless they are library `DECS` which may be incorporated in the final program automatically as needed.

8.14 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 routine texts. This restriction extends to objects such as structures, arrays and unions containing references.

9 Stream Language

This chapter may be regarded as a continuation of B3, which specified the details of stream language for a single module.

\sectionThe imperatives

The constituent modes of OUTPUT needed for the compilation of single modules have been described in B2.1. The remaining imperatives can be divided into three groups

1. Giving information about the current compilation

   `XMODINFO` gives the name and type of the current module

   `XSPEC` gives information about a `HERE` clause, or the current module if it is a `DECS` module

   `XTDTYPE` contains the local number and type of a cc module

   `XBUTYPE` gives advance notice of items to be kept from a `DECS` module

   `XCOMPTYPE` describes the properties of a composition module

   `XKEEPS` contains the decnos of items kept from a `DECS` module

2. Giving information about other modules

   `XTDEC` defines a kept identifier available to the current module

   `XTMODULE` introduces an external module

   `XINTERF` describes a keeplist and introduces a series of `XTDEC`s

3. Representing constructions in the code

   `XOPENMODULE` gives information about the code of a module and marks its beginning

   `XCLOSEMODULE` marks the end of the code of a cc module

   `XCLOSURE` deals with a composition module

   `XCALLMODULE` starts a new cc-module running (produced by a `HERE` clause in the source-text)

9.1 The Current Compilation

The output to stream 0 is

```
XMODINFO   [XSPEC]*
```

followed by the `XEDIT down`, regardless of the type of the module. The mode definitions are as follows

```
MODE  XMODINFO  =  STRUCT (ID name, CAT l, g, INT type),
      XSPEC  =  STRUCT (ID f, INT no, nl, ng,
                        UNION (REF VECTOR [] CHAR,
                               REF VECTOR [] CAT
                              ) u
                       ),
```

where

```
MODE  CAT  =  STRUCT (ID n, f, INT level)
```

The mode CAT (an abbreviation for 'Compiled AT') is used to hold context information and corresponds to CONTEXT f IN n in some module. The name field of XMODINFO contains the name of the module and type is the number of HERE clauses or -1 for a declarations module. The contexts described by the other fields are defined below.

If the module is being compiled at CONTEXT VOID, both l and g will be null. Otherwise, suppose that the local context is CONTEXT a IN b (where for a simple program b will be the standard prelude). Then the first two items in the l field will be (b, a). If b was compiled at CONTEXT VOID, g will be null. If not, g describes the prelude context from which b was derived.

For a cc-module or composition module, there will be one XSPEC corresponding to each HERE Clause. The name of the HERE clause is f and no is its number. The remaining fields contain information on the keeplist (normally given in brackets after HERE in the source text) available to the module to be inserted.

If the module is a declarations module, the XSPEC contains its own name and the no field is 0.

The fields nl and ng are the number of modules accessible at the local and global levels respectively. The mode of u depends on the type of the module; for a cc-module or declarations module it refers to a VECTOR [] CHAR containing a coded form of the keeplist. The u field of a composition module contains a REF VECTOR [] CAT, allowing access to all of its context information.

The output to level 1 depends on the type of module

composition
: XCOMPTYPE

cc
: XSIZES, REF VECTOR [] MDE, XTDTYPE

declarations
: XSIZES, REF VECTOR [] MDE, XBUTYPE

In each case an XEDIT down switches the reader to stream 2. XSIZES and MDE have been described in part B, except for the last two fields of XSIZES. The field nomodules contains the total number of modules involved and nolibinds gives the size of the array required to hold details of identifiers kept from other modules.

The modes XCOMPTYPE, XTDTYPE and XBUTYPE are given by

```
MODE  XCOMPTYPE  =  STRUCT (INT moduleno, type, maxmodule),
      XTDTYPE  =  STRUCT (INT moduleno, type),
      XBUTYPE  =  STRUCT (REF VECTOR [] INT decnos, modes)
```

The type fields are the same as in XMODINFO (-1 for a declarations module, otherwise the number of HERE clauses) and the moduleno is the local number given to the module. maxmodule is the maximum number of modules directly involved in the composition. The purpose of XBUTYPE is to give advance notice of the identifiers and routine texts that are to be kept. The decnos array contains the declaration numbers and the modes array, which is the same size, contains the modes of these kept objects. The decnos are repeated at the end of the compilation of a declarations module, where they are output as an XKEEPS (= REF VECTOR [] INT).

9.2 Parameters Of The Compile Procedure

```
PROC give module details = (ID name, INT mn, BOOL comp) YMODINFO:

PROC give spec = (ID n, f, YM ym) YSPEC:
```

The above procedures are the parameters of the compile procedure that are concerned with modules, where

```
MODE  YMODINFO  =  STRUCT (XMODINFO xmi, YM ym),
```

```
YSPEC  =  STRUCT (XSPEC xs, YS ys),
YM  =  STRUCT (INT version, address),
YS  =  INT
```

The modes YM and YS are implementation dependent and may be extended; the above definitions are chosen so that they may contain the minimum amount of information.

The action of givemoduledetails depends on the value of the second parameter. If it is 0, the procedure must return a YMODINFO, whose xmi field is identical to the XMODINFO output on some previous compilation of the module x, or has illegal type (-2) if not found. If the second parameter is non-zero, the compiler is looking for a library declarations module whose keeplist contains the name x, as specified by the second parameter below:

1 identifier

2 compound symbol (operator)

3 bold symbol (mode or operator)

The third parameter is TRUE if the module currently being compiled is a composition module, otherwise FALSE. The ym field of YMODINFO has a version field which must change whenever the module x changes significantly. It also contains information to allow the translator efficient access to the library. It may be output as part of an XTMODULE (see next section) to check that the module concerned has not changed significantly between compilation and loading; it could also be output in an XINTERF as a help in its other possible role as a fast look-up.

When the compiler calls givespec(n, f, ym), the result must be a YSPEC whose xs field (of mode XSPEC) has name f and was output by a previous compilation of module n; ym is the ym field of givemoduledetails(N, 0,). The ys field contains the version number (as for ym) and must change if the XSPEC changes in any way. A module x changes significantly if its stream 0 is altered in any way.

9.3 Information About Other Modules

Stream 2 contains information defining the kept names, if any, available to the current module as a result of its context. This will be given by

```
XINTERF   XTDEC   [XTDEC]*, where

MODE  XINTERF  =  STRUCT (ID name, YM ym, ID formal, YS ys,
                          INT level, ownlevel
                         ),
      XTDEC  =  STRUCT (BOOL bu, INT level, REF IDDEC id),
      IDDEC  =  STRUCT (ID name, INT decno, level, mode, scope,
                        REF IDDEC rest
                       )
```

Each keeplist contributing to the context will produce one XINTERF (only in this position), immediately followed by a series of XTDECs. Each XTDEC defines one identifier kept in this keeplist available to the current module. The fields name and ym identify the module containing the keeplist, with the next two fields being provided so that the translator may check that this is compatible with the following XTDECs, which were derived from a keeplist at compilation. The level field of XINTERF is the same as in the following XTDECs; this and ownlevel will be discussed more fully later.

The above applies only to modules connected by means of a CONTEXT specification. Names derived from declarations modules, that arise from the USE construction or the default library, are introduced on stream 2 by

```
XTMODULE   XTDEC   [XTDEC]*,
```

where

```
MODE  XTMODULE  =  STRUCT (INT type, moduleno, ID name, YM ym)
```

The XTMODULE is a general construction introducing an external module and giving it a local module number. The name and ym fields are used to locate and check the module, with the type being the same as that of its XMODINFO. In the present case, the type will be -1 since the module is a declarations module; this is the only situation in which it will be followed by XTDECs as above. The bu field will be TRUE if the XTDEC is introduced by an XTMODULE, otherwise FALSE.

The IDDEC pointed to by the id field of an XTDEC has the same form as that of an XIDDEC (See B3.2). If the XTDEC describes a declaration from a declarations module in the default library, the decno of the IDDEC is in a series which starts at startlib + 1. Each decno is used only once in a compilation and refers to an entry in the libinds array. For XTDECs arising from a CONTEXT specification or the USE construction, the declaration numbers are normally included in the series for the current module (ie identifiers start at 4 and procedures at startrd + 1). However, the treatment is different for an XTDEC corresponding to a variable which refers to a generating routine for a kept mode. Firstly, the translator must arrange for the variable to be dereferenced; secondly, startkmp is added to its declaration number. The values of startlib, startrd and startkmp are implementation dependent (see Appendix 5).

The stream language corresponding to the code of the module starts on stream 3. There is an XEDIT down at the end of stream 2.

9.4 A Model For A Running CC Module

The model given below describes the context information that is available to a running cc module. The simplest way to present it is by means of Algol 68 mode declarations, even though they would not appear in any translator. The current module may be thought of as a CONINFO current, where

```
MODE  CONINFO  =  STRUCT( REF CONINFO last, REF [] CLOSURE holes,
                          [] REF CONINFO condisp, [] VALUE keeps
                        ),
      CLOSURE  =  STRUCT (CODE code, REF [] CLOSURE holes)
```

Suppose the current context is CONTEXT p IN q. Then the last field refers to the CONINFO of the module q, with keeps providing information about the keeplist given at the HERE clause p. holes are the HERE clauses of this module, while the condisp field contains the CONINFOs of all modules accessible to this one. There is in the condisp and keeps arrays because indexing should be possible without requiring descriptors.

A CONINFO is rather similar to a procedure environment, with holes and keeps being like parameters and condisp like a procedure display. New CONINFOs are added to this array dynamically by means of XCALLMODULE (see next section, while CLOSUREs may be constructed at load-time by means of XCLOSUREs.

The level fields of the modes in the previous section can be defined most easily in terms of this model. The level field of XINTERF is the index into condisp OF current that gives the CONINFO of the relevant module. This CONINFO contains the VALUEs which are the kepts defined by the XTDECs following the XINTERF; the level of the XINTERF is repeated in the succeeding XTDECS. The ownlevel field of an XINTERF is the size of the condisp of the CONINFO of this module.

There is a level associated with each XTMODULE, but this is given only in the succeeding XTDECs and is not part of the mode XTMODULE. The level fields of the XTDECs define the context under which the module was compiled as being 0 for CONTEXT VOID or equal to the level of some previously introduced XINTERF.

9.5 Constructions In The Code

The code of a cc-module is introduced by an XOPENMODULE and terminated by an XCLOSEMODULE, where

```
MODE  XOPENMODULE  =  STRUCT (ID name,
                              INT maxlevel, nof, moduleno
                             ),
      XCLOSEMODULE  =  STRUCT (INT moduleno, nof)
```

The name field of XOPENMODULE gives the name of the module, with the fields common to both modes, nof and moduleno, giving the number of formals and local module number respectively. The maxlevel field is similar to the ownlevel of an XINTERF, containing UPB condisp OF current; this could be used to combine the condisp with the normal procedure display by allowing space for a copy of the condisp at its base. A composition module may include several open-close pairs around each synthetically generated module, whereas a normal cc-module has only one pair round its code.

The mode XCLOSURE, defined by

```
MODE  XCLOSURE  =  STRUCT (INT body, REF VECTOR [] INT actuals,
                           INT moduleno
                          ),
```

is used to make up a new CLOSURE newcl, with the moduleno field giving it a local module number. The other fields contain local module numbers that have been previously introduced by XTMODULEs, XOPENMODULEs or other XCLOSUREs. The body field refers to the module containing the instructions that become code OF newcl. The elements of actuals refer to the modules that are supplied to the HERE clauses; these are set up as the holes OF newcl.

The code produced by a HERE clause will be an XCALLMODULE, where

```
MODE  XCALLMODULE  =  STRUCT (INTPAIR body,
                              REF VECTOR [] INTRIPLE kset,
                              REF VECTOR [] INT keeps, INT last
                             ),
      INTPAIR  =  STRUCT (INT i, j),
      INTRIPLE  =  STRUCT (INT i, j, k)
```

Its purpose is to start running a new cc-module, having set up a new CONINFO newcon, created as shown below. It will be helpful to first define a recursive procedure:

```
PROC l = (INT n) CONINFO:
IF  n = 0
THEN  current
ELSE  last OF l(n - 1)
FI
```

The Body field is used to construct the CLOSURE for the module to be run, given by

```
CLOSURE m = (holes OF l(j OF body OF c))[i OF body OF c],
```

where c is the XCALLMODULE concerned. The fields of the CONINFO newcon are set up as follows:

holes OF newcon
:= holes OF m

last OF newcon
:= l(last OF c)

keeps OF newcon
consists of the values corresponding to decnos given in keeps OF c

`condisp OF newcon`
: is constructed by the concatenation in order of `(condisp OF l(k))[i:j]` for each of the triples `I`, `J`, `k` in `kset OF c`, with `newcon` added as its final element

9.6 An Example Of Modular Compilation

The basis of this example is the use of two environmental packages, as in C2.7. There are altogether 10 modules, a brief description of which is given below.

`package1` cc, contains 2 `HERE` clauses for user programs

`package2` cc, contains `HERE` clause for user program

`comp2` composition, contains 2 `HERE` clauses, one of which allows the user program to use both packages, as in C2.7.

`cc1` cc, the module to be inserted into the first hole in the above context. It also contains a `HERE` clause and uses two `DECS` modules.

`decs3` declarations, uses no other modules

`decs4` declarations, uses `decs3` and `package1`

`cc3` cc, the module to fill the hole in `cc1`

`comp1` composition, composes `cc1` to incorporate `cc3`

`cc2` cc, the module to be inserted into the second hole in `comp2`

`starter` composition, runs the complete program

The source texts take the following forms:

```
(1)  PROGRAM (userprog, results) package1
     BEGIN ....
           ....
           HERE userprog(keeplist1)
           ....
           HERE results(keeplistr)
           ....
     END
     FINISH

(2)  PROGRAM (userprog) package2
     BEGIN ....
           ....
           HERE userprog(keeplist2)
           ....
     END
     FINISH

(3)  PROGRAM (user1, user2) comp2
     COMPOSE package1(userprog = package2(userprog = HERE user1),
                      results = HERE user2)
     FINISH

(4)  PROGRAM (hole) cc1
     CONTEXT user1 IN comp2
     USE decs3, decs4
```

```
     BEGIN ....
           ....
           HERE hole(keeplist)
           ....
     END
     FINISH

(5)  DECS decs3:
     ....
     ....
     KEEP keeplist3
     FINISH

(6)  DECS decs4
     CONTEXT userprog IN package1
     USE decs3:
     ....
     ....
     KEEP keeplist4
     FINISH

(7)  PROGRAM cc3
     CONTEXT hole IN cc1
     USE decs3
     BEGIN ....
           ....
     END
     FINISH

(8)  PROGRAM comp1
     COMPOSE cc1(hole = cc3)
     FINISH

(9)  PROGRAM cc2
     CONTEXT user2 IN comp2
     BEGIN ....
           ....
     END
     FINISH

(10) PROGRAM starter
     COMPOSE comp2(user1 = comp1, user2 = cc2)
     FINISH
```

9.6.1 Stream Language Produced For The Above Modules

The outputs are given in the order in which they would be read by the translator. `XTDEC*` indicates an arbitrary number of `XTDEC`s. The values of the fields (except for version fields) are shown on the right-hand side. The `u` field of `XSPEC` has mode `REF VECTOR [] CHAR` if it refers to a keeplist - otherwise it has mode `REF VECTOR [] CAT`.

The following declarations are used:

```
CAT c = (%stdprelude, %program, 1)     {default context}
```

```
CAT n = ("", "", 0)                    {VOID context}
CAT u1 = (package1, userprog, 1)
CAT ur = (package1, results, 1)
CAT u2 = (package2, userprog, 1)
ID item = description of a kept identifier
ID lastid = description of the last declared identifier
```

modes: the `REF VECTOR [] MDE` imperative

```
(1)  PACKAGE1

XMODINFO       name = package1, l = c, g = n, type = 2
XSPEC          f = userprog, no = 1, nl = 1, ng = 1, u -> keeplist1
XSPEC          f = results, no = 2, nl = 1, ng = 1, u -> keeplistr
XSIZES
modes
XTDTYPE        moduleno = 1, type = 2
XINTERF        name = %stdprelude, formal = %program, level = 1,
               ownlevel = 1
XTDEC*         bu = FALSE, level = 1, id = item
XOPENMODULE    name = package1, maxlevel = 1, nof = 2, moduleno = 1
XCALLMODULE    body = (1,0), kset = (1,1,0), keeps = keeplist1,
               last = 0
XCALLMODULE    body = (2,0), kset = (1,1,0), keeps = keeplistr,
               last = 0
XCLOSEMODULE   moduleno = 1, nof = 2


(2)  PACKAGE2

XMODINFO       name = package2, l = c, g = n, type = 1
XSPEC          f = userprog, no = 1, nl = 1, ng = 1, u -> keeplist2
XSIZES
modes
XTDTYPE        moduleno = 1, type = 1
XINTERF        name = %stdprelude, formal = %program, level = 1,
               ownlevel = 1
XTDEC*         bu = FALSE, level = 1, id = item
XOPENMODULE    name = package2, maxlevel = 1, nof = 1, moduleno = 1
XCALLMODULE    body = (1,0), kset = (1,1,0), keeps = keeplist2,
               last = 0
XCLOSEMODULE   moduleno = 1, nof = 1


(3)  COMP2

XMODINFO       name = comp2, l = c, g = n, type = 2
XSPEC          f = user1, no = 1, nl = 2, ng = 1, u -> (u1,u2)
XSPEC          f = user2, no = 2, nl = 1, ng = 1, u -> ur
XCOMPTYPE      moduleno = 6, type = 2, maxmodule = 6
XTMODULE       type = 2, moduleno = 1, name = package1
XTMODULE       type = 1, moduleno = 2, name = package2
XOPENMODULE    name = user1, maxlevel = 2, nof = 0, moduleno = 3
```

```
XCALLMODULE    body = (1,2), kset = ((1,2,2), (0,2,2)), keeps = NIL,
               last = 2
XCLOSEMODULE   moduleno = 3, nof = 0
XCLOSURE       body = 2, actuals = (3), moduleno = 4
XOPENMODULE    name = user2, maxlevel = 2, nof = 0, moduleno = 5
XCALLMODULE    body = (2,1), kset = (0,1,2), keeps = NIL, last = 1
XCLOSEMODULE   moduleno = 5, nof = 0
XCLOSURE       body = 1, actuals = (4,5), moduleno = 6


(4)  CC1

XMODINFO       name = cc1, l = (comp2, user1, 3), g = c, type = 1
XSPEC          f = hole, no = 1, nl = 1, ng = 1, u -> keeplist
XSIZES
modes
XTDTYPE        moduleno = 3, type = 1
XINTERF        name = %stdprelude, formal = %program, level = 1,
               ownlevel = 1
XTDEC*         bu = FALSE, level = 1, id = item
XINTERF        name = package1, formal = userprog, level = 2,
               ownlevel = 2
XTDEC*         bu = FALSE, level = 2, id = item
XINTERF        name = package2, formal = userprog, level = 3,
               ownlevel = 2
XTDEC*         bu = FALSE, level = 3, id = item
XINTERF        name = comp2, formal = user1, level = 4, ownlevel = 3
XTDEC*         bu = FALSE, level = 4, id = lastid
XTMODULE       type = -1, moduleno = 1, name = decs3
XTDEC*         bu = TRUE, level = 3, id = item
XTMODULE       type = -1, moduleno = 2, name = decs4
XTDEC*         bu = TRUE, level = 3, id = item
XOPENMODULE    name = cc1, maxlevel = 4, nof = 1, moduleno = 3
XCALLMODULE    body = (1,0), kset = (1,1,0), keeps = keeplist,
               last = 0
XCLOSEMODULE   moduleno = 3, nof = 1


(5)  DECS3

XMODINFO       name = decs3, l = c, g = n, type = -1
XSPEC          f = decs3, no = 0, nl = 1, ng = 0, u -> keeplist3
XSIZES
modes
XBUTYPE        keeplist3
XINTERF        name = %stdprelude, formal = %program, level = 1,
               ownlevel = 1
XTDEC*         bu = FALSE, level = 1, id = item
XKEEPS         keeplist3


(6)  DECS4
```

```
XMODINFO        name = decs4, l = u1, g = c, type = -1
XSPEC           f = decs4, no = 0, nl = 1, ng = 1, u -> keeplist4
XSIZES
modes
XBUTYPE         keeplist4
XINTERF         name = %stdprelude, formal = %program, level = 1,
                ownlevel = 1
XTDEC*          bu = FALSE, level = 1, id = item
XINTERF         name = package1, formal = userprog, level = 2,
                ownlevel = 2
XTDEC*          bu = FALSE, level = 2, id = item
XTMODULE        type = -1, moduleno = 1, name = decs3
XTDEC*                bu = TRUE, level = 3, id = item
XKEEPS          keeplist4
```

(7) CC3

```
XMODINFO        name = cc3, l = (cc1, hole, 1), g = c, type = 0
XSIZES
modes
XTDTYPE         moduleno = 2, type = 0
XINTERF         name = %stdprelude, formal = %program, level = 1,
                ownlevel = 1
XTDEC*          bu = FALSE, level = 1, id = item
XINTERF         name = cc1, formal = hole, level = 2, ownlevel = 2
XTDEC*          bu = FALSE, level = 2, id = item
XTMODULE        type = -1, moduleno = 1, name = decs3
XTDEC*          bu = TRUE, level = 3, id = item
XOPENMODULE     name = cc3, maxlevel = 2, nof = 0, moduleno = 2
XCLOSEMODULE    moduleno = 2, nof = 0
```

(8) COMP1

```
XMODINFO        name = comp1, l = (comp2, user1, 3), g = c, type = 0
XCOMPTYPE       moduleno = 3, type = 0, maxmodule = 3
XTMODULE        type = 1, moduleno = 1, name = cc1
XTMODULE        type = 0, moduleno = 2, name = cc3
XCLOSURE        body = 1, actuals = (2), moduleno = 3
```

(9) CC2

```
XMODINFO        name = cc2, l = (comp2, user2, 2), g = c, type = 0
XSIZES
modes
XTDTYPE         moduleno = 1, type = 0
XINTERF         name = %stdprelude, formal = %program, level = 1,
                ownlevel = 1
XTDEC*          bu = FALSE, level = 1, id = item
```

```
XINTERF         name = package1, formal = userprog, level = 2,
                ownlevel = 2
XTDEC*          bu = FALSE, level = 2, id = item
XINTERF         name = comp2, formal = user2, level = 3, ownlevel = 2
XTDEC*          bu = FALSE, level = 3, id = lastid
XOPENMODULE     name = cc2, maxlevel = 3, nof = 0, moduleno = 1
XCLOSEMODULE    moduleno = 1, nof = 0
```

(10) STARTER

```
XMODINFO        name = starter, l = c, g = n, type = 0
XCOMPTYPE       moduleno = 4, type = 0, maxmodule = 4
XTMODULE        type = 2, moduleno = 1, name = comp2
XTMODULE        type = 0, moduleno = 2, name = comp1
XTMODULE        type = 0, moduleno = 3, name = cc2
XCLOSURE        body = 1, actuals = (2,3), moduleno = 4
```

Appendix A Values and representations of symbols

Note: symbols marked with a * must have a single-character representation provided for them.

The values of the symbols are given in brackets.

A.1 Note

Any representation whose meaning is undefined must be given one of the following values in the `charset` array or `lookup` procedure:

156 (other bold)
: if the representation is to be an acceptable form for a user-defined mode name or operator

176 (other op)
: if the representation is to be acceptable as a user-defined operator symbol but unacceptable as a mode name

100 (illegal)
: if the representation is to be illegal

Appendix B Fixed Mode Numbers

The mode numbers given below are fixed for all programs. For 1 to 21, 29 and 30, the mode `MDE` decomposes to `PRIMITIVE`.

These mode numbers assume that there may be up to 1 `SHORT` or 2 `LONG` symbols before a mode or denotation (although these are not all required to represent different lengths). It is possible to allow `SHORT SHORT` (at the expense of `LONG LONG`) by interchanging `LONG` and `SHORT` in the following table. If this is done, the values for the symbols `LONG` and `SHORT`, `LENG` and `SHORTEN` must also be interchanged.

1 `vacmode`

2 `skipmode`

3 `nilmode`

4 `gotomode`

5 `voidmode`

6 `faultmode` (does not occur in stream language)

7 `BOOL`

8 `CHAR`

9 `FORMAT`

10 `SHORT BITS`

11 `BITS`

12 `LONG BITS`

13 `LONG LONG BITS`

14 `SHORT INT`

15 `INT`

16 `LONG INT`

17 `LONG LONG INT`

18 `SHORT REAL`

19 `REAL`

20 `LONG REAL`

21 `LONG LONG REAL`

22 `SHORT COMPL = STRUCT (SHORT REAL re, im)`

23 `COMPL = STRUCT (REAL re, im)`

24 `LONG COMPL = STRUCT (LONG REAL re, im)`

25 `LONG LONG COMPL = STRUCT (LONG LONG REAL re, im)`

26 `VECTOR [] CHAR`

27 `[] CHAR`

28 `collatmode`

29 `XTYPE`

30 `YTYPE`

Appendix C Numbering of standard prelude operators in XOPER

Key to headings:

BL	BOOL
C	CHAR
LB	BITS (possibly with LONG or SHORT prefixes)
I	INT
LI	INT (possibly with LONG or SHORT prefixes)
LR	REAL (possibly with LONG or SHORT prefixes)
LC	COMPL (possibly with LONG or SHORT prefixes)
VC	VECTOR [] CHAR (REF FLEX for +:=, *:=)
AC	] CHAR (REF FLEX for +:=, *:=)
V	Any vector
A	Any array S Any straight

C.1 Monadic operators

		BL	C	LB	I	LI	LR	LC	VC	AC	V	A	S
	op					version number							
	number												
+	0	-	-	-	-	1	2	3	-	-	-	-	-
-	1	-	-	-	-	1	2	3	-	-	-	-	-
UPB	2	-	-	-	-	-	-	-	-	-	1	2	3
LWB	3	-	-	-	-	-	-	-	-	-	1	2	3
NOT	4	1	-	2	-	-	-	-	-	-	-	-	-
ABS	5	1	2	3	-	4	5	6	-	-	-	-	-
BIN	6	-	-	-	-	1	-	-	-	-	-	-	-
REPR	7	-	-	-	1	-	-	-	-	-	-	-	-
LENG	8	-	-	1	-	2	3	4	-	-	-	-	-
SHORTEN	9	-	-	1	-	2	3	4	-	-	-	-	-
ODD	10	-	-	-	-	1	-	-	-	-	-	-	-
SIGN	11	-	-	-	-	1	2	-	-	-	-	-	-
ROUND	12	-	-	-	-	-	1	-	-	-	-	-	-
ENTIER	13	-	-	-	-	-	1	-	-	-	-	-	-
RE	14	-	-	-	-	-	-	1	-	-	-	-	-
IM	15	-	-	-	-	-	-	1	-	-	-	-	-
ARG	16	-	-	-	-	-	-	1	-	-	-	-	-
CONJ	17	-	-	-	-	-	-	1	-	-	-	-	-

C.2 Dyadic operators

		BL	C	LB	I	LI	LR	LC	VC	AC	V	A	S
	op					version number							
	number												
+	0	-	1	-	-	2	3	4	5	6	-	-	-

```
-          1     -  -  -  -    1  2  3  -    -  -  -  -
UPB        2     -  -  -  -    -  -  -  -    -  -  1  -
LWB        3     -  -  -  -    -  -  -  -    -  -  1  -
ANDTH      4     1  -  -  -    -  -  -  -    -  -  -  -

OREL       5     1  -  -  -    -  -  -  -    -  -  -  -
<          6     -  1  -  -    2  3  -  4    5  -  -  -
>          7     -  1  -  -    2  3  -  4    5  -  -  -
<=         8     -  1  2  -    3  4  -  5    6  -  -  -
>=         9     -  1  2  -    3  4  -  5    6  -  -  -

=         10     1  2  3  -    4  5  6  7    8  -  -  -
/=        11     1  2  3  -    4  5  6  7    8  -  -  -
*         12     -  1  -  -    2  3  4  5    6  -  -  -
/         13     -  -  -  -    1  2  3  -    -  -  -  -
OVER      14     -  -  -  -    1  -  -  -    -  -  -  -

MOD       15     -  -  -  -    1  -  -  -    -  -  -  -
**        16     -  -  -  -    1  2  3  -    -  -  -  -
I         17     -  -  -  -    1  2  -  -    -  -  -  -
SHL       18     -  -  1  -    -  -  -  -    -  -  -  -
SHR       19     -  -  1  -    -  -  -  -    -  -  -  -

ELEM      20     -  -  1  -    -  -  -  -    -  -  -  -
PLUSAB    21     -  -  -  -    1  2  3  4    5  -  -  -
MINUSAB   22     -  -  -  -    1  2  3  -    -  -  -  -
TIMESAB   23     -  -  -  -    1  2  3  4    5  -  -  -
OVERAB    24     -  -  -  -    1  -  -  -    -  -  -  -

MODAB     25     -  -  -  -    1  -  -  -    -  -  -  -
DIVAB     26     -  -  -  -    -  1  2  -    -  -  -  -
IS        27       (only version number = 1)
ISNT      28       (only version number = 1)
AND       29     1  -  2  -    -  -  -  -    -  -  -  -

OR        30     1  -  2  -    -  -  -  -    -  -  -  -
PLUSTO    31     -  -  -  -    -  -  -  1    2  -  -  -
CYCLE     32     -  -  -  -    -  -  -  -    -  -  1  -
```

Appendix D Extensions To ALGOL 68

D.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` results in an array. In strong contexts, a single object can be *rowed* to a vector. The overheads associated with vectors are smaller than 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 while 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; the permissible range of values is from 1 to `maxistruct` (see Appendix 5). An i-struct can be indexed with the same notation as for an array, and the indexing starts at 1. If trimmed, it normally gives rise to a vector (although the `AT` construction produces an array). 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  Y2  =  (VECTOR [] REAL p) ... ... ;
OP  Y2  =  ([] REAL p) ... ... ;
```

With these two declarations in force, an operand of mode `STRUCT 4 REAL` would be coerced to `VECTOR [] 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` does not appear in compile-time diagnostic messages unless it has already been explicitly used in the source-text, while strings like `"ABC"` are typically described as `3 CHAR` rather than `STRUCT 3 CHAR`. The mode of the empty string `""` is `STRUCT 0 CHAR`, even though i-structs with zero size cannot be declared.

D.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 *:= 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 bounds. For example,

```
VECTOR [10] INT v;
```

```
[10, 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, ])` 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 (ref) multi-dimensional arrays. The expression `n CYCLE w` delivers the (ref) array `w` with a new descriptor, in which the dimensions are cycled to bring the `(n+1)`th to the front.

D.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, straightening or any of the coercions i-struct to vector to array (A4.1). 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

```
[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 [] REAL, STRUCT 17 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 VECTOR [] 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 because objects of mode REF STRAIGHT do not exist.

D.4 Low level facilities

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 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 CHARS) are equivalent: 8r "1 15 251", 16r "1 d a9", where a-f represent the digits 10-15.

D.5 Built-in operators

The declaration

```
OP (INT, INT) INT  ** = BIOP 1013
```

declares the operator ** in the usual way, but the definition is built into the translator. The integer after BIOP corresponds to the param field of a dyop or monop (in XOPER).

The BIOP construction may also be used in declarations of the form

```
M x = BIOP 671
```

where M is any mode, most usefully a procedure with three or more parameters. The integer corresponds to the decno of an identifier declaration.

If built-in operators or identifiers are to be used in other modules, the integer after BIOP must be less than Maxchar*maxchar (see Appendix 5).

D.6 Generalised modes

The primitive modes XTYPE and YTYPE have been introduced as representations for any simple mode. For the purpose of this definition, a 'simple' mode is one which contains no vectors, arrays or references or which is a vector of such objects. They may be used in transput routines to avoid the overheads of straightening. For scope reasons objects of modes XTYPE and YTYPE may not be assigned.

The coercions associated with these modes are in the direction YTYPE to M to XTYPE or REF YTYPE to REF M to REF XTYPE, where M is any simple mode. The coercions from REF YTYPE and REF M may also remove the initial REF; the compiler does not output an xderef in this case. Thus a procedure dealing with objects of various simple modes might have the specification

```
PROC p = (VECTOR [] XTYPE x) YTYPE:
```

so that the parameters and result could be handled easily. If coercions in the opposite direction are required, the translator must incorporate the relevant BIOPs.

The coercion to XTYPE is allowed before uniting, but as with i-structs and vectors, XTYPE may exist side by side in the same union with a simple mode or a STRAIGHT. Possible ambiguities are resolved by preference for minimum travel in the 'simple to XTYPE to STRAIGHT' direction.

D.7 Implementation dependent declarations

One section of the text of the RS compiler is marked as implementation-dependent and may be changed freely by implementors. The modes and values given below are used in the Algol 68 to C translator implementation.

`MODE ID = STRUCT 12 CHAR`
: mode used to store an identifier, label, mode or operator

`INT maxid = 12`
: maximum significant length of (1)

`INT maxchar = 64`
: size of character set

`MODE YM = STRUCT (INT album, index, version)`
: description of module

`MODE YS = INT`
: description of spec

`INT startrd = 1000`
: integer added to routine numbers

`INT startlib = 2000`
: integer added to numbers of library identifiers

`INT startkmp = 10000`
: integer added to numbers of kept modeprocs

`INT upbofmodes = 500`
: size of modes array in compiler

`INT upbofsidstack = 250`
: size of analyser stack

`INT maxistruct = 512`
: maximum allowable size of indexable structures

`CHAR dchar = "d", pchar = "p", nchar = "n"`
: for REAL denotations in stream language

`INT linesize = 160`
: maximum length of a line of source-text

Short Contents

Table of Contents