[illegible]

RSRE
MEMORANDUM No. 3522

ROYAL SIGNALS & RADAR ESTABLISHMENT

CURT : THE COMMAND INTERPRETER LANGUAGE FOR FLEX

Authors: I F Currie and
J M Foster

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 3522

[illegible]

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3522

Title: CURT: THE COMMAND INTERPRETER LANGUAGE FOR FLEX

Authors: I F Currie and J M Foster

Date: September 1982

SUMMARY

Curt is an interactive command language for the Flex computer system. Since one feature of Flex is its treatment of procedures as first class data objects, the creation and calling of procedures is the most important part of Curt. In addition, Curt handles a wide spectrum of types of values in as uniform a manner as possible and is a very flexible tool for the construction of programs.

Curt: the command interpreter language for Flex

Contents

1. Introduction

Curt is an interactive command language for the Flex [1] computer system. Since one feature of Flex is its treatment of procedures [2, 3] as first class data objects, the creation and calling of procedures is the most important part of Curt. In addition, Curt handles a wide spectrum of types of values in as uniform a manner as possible and is a very flexible tool for the construction of programs.

Curt is implemented as an Algol 68 procedure which interprets the language. This procedure is accessible to any user of Flex, whether in Algol 68 or not. It is structured in a manner to allow one to call fresh invocations of curt from within other programs. For example, the editor calls curt to allow command language interactions within it, extending the set of recognised names by other names corresponding to subfiles within the file being edited.

Clearly, in order to use Curt successfully one requires to know more about the system than is given in this paper. One requires to know how to use the various procedures and values made available to the user in a Curt interaction as well as the structure and semantics of the Curt language. For example, one would have to know how to use the procedures denoted by the names held in common across the system e.g. edit. However, this is outside the scope of this document; besides, the total environment accessible to a user will expand as a result of his efforts, as well as that of others.

2. An informal overview of Curt

2.1 General

Any operating system is a program, or set of programs, to allow one to construct, test, and run other programs. From the user's point of view, most operating systems can be regarded as having three distinct parts:

1. The filestore. This is that part of the memory of the system which remains valid between sessions or jobs. Historically, this usually consisted of files belonging to a small number of types (e.g. text files, index sequential etc) which could be created and accessed by knowing their names in the right kind of environment. Flex breaks from this by allowing one to keep much more general values in filestore; one can construct objects in filestore with much the same level of complexity as mainstore objects. Flex filestore objects can contain pointers to other filestore objects so that tree structures, for example, are simple to construct in filestore. Indeed, most mainstore objects have direct analogues in Flex filestore (including procedures) with the same kind of properties and restrictions in use. For example, the only thing that one can do with a Flex procedure whether in filestore or in mainstore is to call it; the possession of a procedure object does not allow one to dismember it to find how it works or what other procedures in turn that it might use - this is the basis of much of the security of data in Flex filestore and mainstore.

2. Utility programs. Examples of utilities are compilers, editors, debugging aids etc. In Flex, utilities are just procedures most of which are held on filestore. These procedures are essentially no different from those that a user can construct for himself - they were certainly all constructed in more or less the same way. The utilities are only distinguished by the fact that they are named in common across the system. Every user has his own private dictionary which gives the names of his values and which he can alter at will;

the names that correspond to the values of common utilities are held in the common dictionary, which can only be altered by those users who possess the appropriate procedure to update it.

3. The command interpreter. The command intepreter for Flex is an interactive one; it is implemented by a procedure named curt and implements an interpretive language called Curt. Curt is an extremely simple language; effectively all that one can do in Curt is call procedures and give names to values - it has no loops, conditionals or variables. These latter constructions are intended to be implemented either within procedures being called or else by choosing to do either one thing or another interactively.

This paper is mainly concerned with the Curt language; however, Curt will almost always be used in an environment in which the common utilities will be available via the common dictionary and using the filestore. Most of the real work in program construction, for example, will be done inside procedures like the compilers. To use them successfully, one requires to know their detailed specification; Curt itself will only provide a convenient framework for calling the procedures with the right kind of parameters. In the examples given later the emphasis is on how Curt works rather than what is actually done by the various procedures called by Curt.

2.2 Procedure calls in reverse Polish

Since the design of Flex is based on the uniform use of procedure values, it is to be expected that the most important function of Curt is to call procedures. A program run in Flex is always a call of a procedure with some parameters and delivering some answer; the parameters are the input data to the program and the answer to the procedure is the output data. Frequently, the output of one program forms the input to another, i.e. the answer delivered by a procedure is a parameter to another. Interactively, it makes a good deal of sense to express these procedure calls in reverse Polish i.e. a notation which allows us to write the expression for a parameter before we need mention the procedure to be called. A procedure call which would be f(x) in normal notation, is expressed in Curt by:

```
x f!
```

The !-symbol here says that the last "thing" on the line (f) is a procedure and asks for that procedure to be called with the previous "thing" (x) as its parameter. At first sight this !-symbol might appear redundant; however, since procedures delivering other procedures are commonplace in Flex, one requires some way of providing the correct binding of a procedure with its desired parameter. For example, g(f(x)) is expressed in Curt by:

```
x f! g!
```

and g(f)(x) is expressed by:

```
x f g!!
```

Clearly, the "thing"s mentioned above include the possibility that they are themselves the results of procedure calls.

2.3 The Curt line, underlining and intermediate results

Text to be interpreted by Curt is typed on a 160 character line called the Curt line which is usually displayed on the bottom two lines of the vdu. Interactions only occur when one of the control keys is pressed. When a procedure called on the Curt line is evaluated, the text corresponding to the call is underlined on the screen, for example:

$\underline{\texttt{x f!}}$

This underlining serves two purposes. First, it indicates when the procedure call has been completed and second it gives a name to the result of the procedure call. Thus, if we edited the above Curt line to:

$\underline{\text{x f!}}$ g!

and pressed the DO-key, the call on f would not be re-evaluated since $\underline{\text{x f!}}$ is a name for the already evaluated result of that call and this would be used as the parameter for the call of g. When the call of g is completed the text would be underlined as before:

$\underline{\text{x f! g!}}$

and we now have two underlined names, one for the call of f as before, $\underline{\text{x f!}}$, and one for the result of the call of g, $\underline{\text{x f! g!}}$.

We could have reached exactly the same state in one interaction by typing:

x f! g!

The sequence of underlining and the underlined names themselves would be the same as in the double interaction above.

Any error in the Curt line will result in some error message being displayed above it, with the cursor re-appearing at the offending place. The Curt line can then be edited and re-tried. The net effect of the underlining and the reverse Polish notation means that this is usually a natural and economical process.

Any underlined names remain valid until the Curt line is completed. A Curt line is completed (and also cleared) by either pressing the DEL-LINE key or doing a declaration.

2.4 Declarations

One can name values in Curt, in the sense of simply letting an identifier stand for a value. This can be performed in one of two ways. First, an identifier may be made to stand for a value so that the meaning of the identifier remains valid from session to session i.e. the value is put into the filestore. This is indicated by the "==" symbol; thus, after:

```
24 == twentyfour
```

the name twentyfour will be put into the user's permanent dictionary as a name for the integer 24. This is, of course, rather a trivial value to put into filestore, but it illustrates that the filestore can contain objects other than files. The filestore may also contain structures containing scalars (integers, characters, reals etc) and pointers to things already in filestore. Many functions and modules exist to do just that, forming pointers to objects of varying modes.

The other method of declaration can be used to name <u>any</u> object, but the scope of the name will be simply the current session i.e. the object is not put into filestore. One uses the "=" symbol for this; thus, after:

```
42 = fortytwo
```

the name fortytwo will stand for the integer 42 until the end of the session (or until it is redeclared).

Both forms of declarations may be used to name the components of a structured value, as in the example given below :

```
( 1 , 3.14 , "qwerty") = (one, nearpi, keyboard)
```

Here, one is a temporary name for the integer 1, nearpi for the real 3.14 and keyboard for the vector of characters "qwerty". Of course, one would normally use this component declaration to name results of procedures delivering structured values, rather than explicit structures as above.

2.5 Example: How to write and test an Algol68 Module

As mentioned previously, one really has to know the detailed specifications of the common utilities of Curt in order to use the system to its maximum potential. However, one can go a long way with only a sketchy knowledge of both these specifications and how Curt treats the modes of the objects that it handles. To illustrate, this let us write and test an Algol68 procedure to sort a vector of strings into alphabetical order.

One starts by inputting the Algol68 text. This is usually done using the editor. Supposing that we required the maximum line width of our text to be 80 characters, we would do this by:

```
80 edit!
```

and typing in the text of the sort module to where the cursor appears in the area above the Curt line:

```
sort :

MODE LINE = REF VECTOR[]CHAR;

PROC sort = ( VECTOR[]LINE in ) VECTOR[]LINE:
( VECTOR [UPB in] LINE out := in;
  { some suitable way of sorting the vector out  }
  out
)

KEEP sort
FINISH
```

After completing the edit (by pressing the RESULT-key), we could name this text file thus:

```
80 edit! == sort_text
```

The text, sort_text can be re-edited thus:

```
sort_text edit! == sort_text
```

or we could compile sort_text by:

```
sort_text algol68!
```

If the compilation failed due to language errors in the text the procedure algol68 calls the editor within itself to allow the user to correct the errors, which can be stepped through in a simple way. In this case, algol68 delivers the edited text which can then be named and re-compiled. If the compilation succeeds, the result of algol68 is a compiled object; this compiled object is usually kept in filestore by putting it into a module which is a variable containing a compiled object. Since this is our first compilation of sort we have to produce a new module to contain the compiled object. This is done by calling the procedure new and in this case, we will give it the name sort_module:

```
sort_text algol68! new! == sort_module
```

Another program which required to use sort could do so by heading its text :

```
another_program USE sort_module,...
```

The point about a module being a variable containing a compiled object rather than simply being the compiled object is that we can re-compile and amend the module so that any other programs which use it need not themselves be re-compiled. We could edit sort_text, perhaps to make it faster, without changing its specification and have the new version be included in these other programs by using the procedure amend:

```
sort_text algol68! sort_module amend!!
```

Note that amend is a procedure which takes a module as a parameter and delivers a procedure; this result procedure is one which puts the result of the compilation into the module, provided that the re-compilation has not changed the Algol68 specification held in the module.

We can test the operation of the sort procedure by making it into a Curt procedure, using the procedure file68, in the following manner:

```
sort_module file68! == sort
```

The identifier, sort, now stands in Curt for the procedure kept in the Algol68 module, and we can call it on the Curt line provided we give it the correct kind of parameter:

```
["qwerty","asdfghj","zxc"] sort!
```

The parameter here is just a vector of strings as required by the sort procedure, and provided that the routine operates correctly, the result will be given by the usual underlined name. The value of this result can be displayed above the Curt line by:

```
["qwerty","asdfghj","zxc"] sort! show!
```

The procedure show will, among other things, display the value given as its parameter in a format and style depending on the mode of the value. In the above case, the display would be something like:

```
 Vector holding:
> "asdfgh"
> "qwerty"
> "zxc"
```

Of course, it is possible that a mistake in our procedure would cause it to fail, e.g. we could have an index out of bounds or an arithmetic overflow. In this case the result of the procedure call is a special diagnostic value and the processing of the Curt line is stopped for further interaction. This special diagnostic value is indicated in the following way:

```
["qwerty","asdfghj","zxc"] sort?
```

and could be further analysed using the diagnose procedure :

```
["qwerty","asdfghj","zxc"] sort? diagnose!
```

This diagnosis indicates the sequence of procedure calls involved in the failure. It also permits the user to display values corresponding to any of the program identifiers in scope at the time of the failure, possibly again using the procedure show.

In practice we probably would not have named the module and the procedure above; it is usually more convenient to hold these values in an editable file which can also hold other related texts and modules as sub-files and values. A full description of this is outside the scope of this document; suffice it to say that the editor and compiler, as well as Curt, are well adapted to the kind of manipulations required to do this. All of the utilities of the system, including these mentioned, were produced in this manner; there is very little difference between user's procedures and those which are part of the system.

3. Curt modes

It is clear from Chapter 2 that Curt has to keep track of the modes of the objects that it manipulates, at the very least to ensure that the correct types of parameters are applied to procedures. The modes of values used in most common programming languages can be mapped onto a subset of Curt modes. Thus, the function file68 used in 2.5 to create a Curt procedure from an Algol68 procedure tries to do this mapping automatically. This is not always possible to do since Curt modes have the limitation that they cannot be circular. However, they do have more constructors and the possibility of having user-defined atomic modes. For example, if we wished to manipulate Flex disc pointers in Algol68, their mode in Algol68 would have to be simply INT, while Curt modes have constructors like Disc and Filed to describe the complex structures on filestore.

3.1 Common modes

The various Algol68 primitive modes and constructions for modes that have direct analogues in Curt are:

Algol68	Curt	
INT	Int	
REAL	Real	
CHAR	Char	
BOOL	Bool	
VOID	()	or Void
REF X	Ref X	
PROC (X) Y	X -> Y	
PROC X	() -> X	or Void -> X
STRUCT(X a,Y b,...)	(X,Y,...)	
STRUCT n X	n * X	
{REF} VECTOR [] X	Vec X	
{REF} [,...] X	[,...] X	
UNION(X, Y,...)	Union(X, Y,...)	

3.2 Any, choice and Moded

Curt allows a type of union called a choice which is less structured than that of Algol68; thus a value of mode Choice(X, Y, ...) is a value of mode X or of mode Y etc. Presumably some intrinsic properties of the component modes will allow them to be differentiated within any procedure which is passed a Choice. Note that a Union(X, Y, ...) has roughly the structure (Int, Choice(X, Y, ...)) where the Int indicates which of the modes is actually present. The procedure, edit, mentioned in 2.5 has mode Choice(Int, Edfile) -> Edfile, so that its parameter can either be an integer (giving the maximum line width for a new file) or else an existing text file.

Another Curt mode with the flavour of Choice is the mode Any which is effectively an infinite choice.

One of the main reasons for having modes in any language is to make sure that the correct kind of objects are supplied as parameters to procedures. Curt modes are no exception to this and usually the mode of the actual parameter of a procedure call has to match the mode of the formal parameter of the procedure exactly. The only exceptions to this (i.e. the only coercions on parameters in Curt) arise where the formal parameter of a procedure is one of the above modes Any or Choice(X, Y, ...) or a special mode called Moded. The first two are fairly straight-forward; any mode is acceptable to a procedure with a formal parameter of mode Any while any of X, Y, ... etc is acceptable to one with mode Choice(X, Y, ...). The action when the formal parameter is Moded is a little more complicated; the parameter actually passed to the procedure is a pair consisting of the value of the parameter and a representation of its mode. The mode of such a parameter in Algol68 is:

```
MODE MODED = STRUCT( VALUE value , CURTMODE mode)
```

where both VALUE and CURTMODE are REF VECTOR [] INT, but, of course both can contain values other than Ints. The intention of this kind of parameter is to allow the procedure some freedom in interpreting what to do with its parameter depending on its mode. For example, the procedure show (used in 2.5) takes a Moded parameter and its action is to display a print-out of the parameter. The procedure show takes any parameter coded as a MODED and uses its mode field to decide how to display the the value; clearly one prints out an integer differently from a vector of reals. The inverse operation occurs if the answer mode of the procedure is Moded; Curt interprets the mode part of the pair actually delivered by the procedure as the mode of the value part which it takes as the answer to the procedure. Thus the answer part of the mode of the procedure, file68, is Moded; the mode actually delivered will depend on the Algol68 module on which it operates.

3.3 Ptr, repetitions and Filed

The Curt mode Ptr X is the mode of a Flex mainstore pointer where the unpacked block pointed at has mode X, while Disc X is similarly a filestore pointer; if d is a Disc X, then a call of the kernel procedure d_to_b on d will produce a Ptr X.

Curt modes include indefinite repetitions of a given mode by use of the * prefix; thus the mode *Int would indicate some integers. Once again there would probably be some implicit method to allow one to deduce how many e.g. the mode (Ptr *Int) might be quite sensible.

When one produces new values from compiled modules to keep in filestore, these values will generally be of mode Filed X. For example, the actual mode delivered by file68 (via the Moded mentioned in 3.2) is Filed X, where X is a more or less direct translation to Curt modes of the Algol68 modes involved in the KEEP list of the module, given in 3.1. If one applies the procedure load to a Filed X one gets the corresponding X. This process is done implicitly if the value before the !-symbol on a Curt line has mode Filed X, so that procedures held on disc usually have mode Filed (X -> Y). The main advantage of values of type Filed (besides the fact that they can live in filestore) is that they are constructed from modules which can be amended so that Filed value can easily be changed.

3.4 Atomic modes

Curt modes also include atomic modes which can be introduced by the user. An atomic mode is one where it is considered undesirable or unnecessary to make explicit the full structure; one cannot construct a value with an atomic mode using the Curt constructors but only as the result of a procedure. An example of an existing atomic mode is the mode of an editable file, Edfile. The underlying structure in an Edfile is effectively defined by the editor, edit, whose mode is Choice(Int, Edfile) -> Edfile; most new values of mode Edfile are constructed by calling the edit procedure with an integer parameter. Other existing atomic modes include Mode and Atomic to allow us to construct modes and new atomic modes at the Curt level; thus, the most usual way of constructing modes is by using the procedure make_mode which has mode (Vec Char) -> Mode so that one can construct a value of mode Mode by:

```
"(Int,Int,Int)" make_mode !
```

This value (after it is underlined by obeying it) is the mode (Int, Int, Int). We could construct a new atomic mode, Xtra, whose values have the same size as this mode by calling the procedure new_atomic whose mode is (Mode, Vec Char) -> Atomic :

```
( "(Int,Int,Int)" make_mode ! , "Xtra" ) new_atomic !
```

After this is obeyed, make_mode will recognise the mode name Xtra as an atomic mode, different from all others.

The other procedures used in 2.5 also have modes which involve the defined atomic modes Edfile, Compiled, Module:

```
algol68 : Edfile -> Moded          { the Moded will be either Compiled
                                     or Edfile depending on whether or
                                     not the program compiled }

new : Compiled -> Module

amend : Module -> ( Compiled -> Compiled )

file68 : Module -> Moded           { the Moded will be Filed X where
                                     X depends on the specification of
                                     the Module parameter }

show : Moded -> Edfile             { The text displayed is a printout
                                     of the parameter depending on its
                                     mode }
```

3.5 Generic procedures

Finally, a Curt generic procedure has mode X => Y, where X and Y may contain formal mode names ?, ??, ???, etc, but no further generic procedure modes i.e. no further occurrence of the => symbol. Also if a particular formal name occurs in Y, it must have appeared in X. When an actual parameter is applied to a generic procedure, each of the formal modes is replaced consistently with an actual mode derived from the parameter. The equivalent non-generic procedure is then applied to this actual parameter in the normal way. For example, a procedure to compose two other procedure values could have mode:

```
((? -> ??) , (?? -> ???)) => (? -> ???)
```

Hence if x has mode Int -> Real, and y has mode Real -> Bool, the Curt expression:

```
(x , y) compose !
```

would have mode Int -> Bool by making the substitutions ? = Int , ?? = Real and ??? = Bool.

Values, like compose, with generic modes are rather difficult to construct in the standard programming languages. For this reason, most generic procedures used in practice will be either be held as a system utility or delivered from one.

4. Curt syntax and semantics

The following description indicates what happens when a Curt line is evaluated. This is expressed in terms of the curt procedure, whose initial call will give the user access to commonly held values e.g. those for compiling and editing mentioned in 2.5.

4.1 The curt procedure

Curt is implemented as a procedure, curt, in Algol68 :

```
PROC curt = ( PROC( IDENT ) MODED find,
              PROC( INT, IDENT, MODED ) VOID keep,
              PROC( INT, LINE ) monitor,
              LINE first_line
            ) MODED :
```

where IDENT and LINE are both VECTOR[]CHAR and MODED is the value-mode pair corresponding to the Curt mode, Moded, mentioned in section 3.2.

The procedure find will deliver the value-mode pair which is associated with an identifier (or an underlined word) read on the Curt line in this call of curt. In general, any environment which calls curt will give the user access to this find (and to the other parameters) so that recursive calls of curt can use a new find procedure constructed using the old one, so giving a cascade of name spaces.

The procedure keep is called by curt at a Curt declaration to make a new association of an identifier with a value-mode pair, which should be taken note of by the corresponding find. The INT parameter indicates whether the declaration is to be temporary (1) or permanent (2).

The procedure monitor is called by curt to signal the text of lines containing declarations back to the calling environment of curt and first_line will be the first line interpreted and displayed by this call of curt.

The value-mode pair delivered by curt is given by the value of the Curt expression on the Curt line when the RESULT-key is pressed.

So far as an individual user is concerned, the procedure curt is called initially with a find parameter which allows him to get values from his local dictionary and from the common dictionary. The keep parameter will allow him to name values in his local dictionary. Procedures which he uses may call curt recursively; these calls will generally have the same keep parameter as the initial call, but the find will probably be extended to allow access to more values. For example, when calling the procedure edit, pressing the GOIN-key will cause a recursive call of curt with a find parameter which gives access all the sub-files in the file being edited as well as those values which are normally accessible. Similarly, the diagnose procedure can call curt recursively with a find parameter that gives access to the identifiers of the failed program.

4.2 The Curt line

On the Logica VDUs, the Curt line consists of 160 characters on the bottom two lines of the screen in reverse video. The syntax of one complete Curt line is given below; the conventions for this syntax are given in Chapter 5.

```
<Curt line>  ::=  <Declaration>  DO-key
             ::=  <Expression>  DO-key <further interaction>
             ::=  <Expression> RESULT-key
             ::=  <any old rubbish>  DEL LINE-key
             ::=  INS LAST-key

<Declaration>  ::=  <Expression> = <Idset>
               ::=  <Expression> == <Idset>

<Idset>  ::=  <identifier>
         ::=  ( <<identifier>-list> )
```

The <Expression>s above are evaluated when a DO- or RESULT-key is pressed. As the <Expression> is being evaluated, the text of each of the identifiers found and procedures evaluated will be underlined, and these underlined strings will become names for their corresponding values. If the <Curt line> is in error or incomplete after the control-key has been pressed the cursor will re-appear at an appropriate place and, in the case of an error a message will be "rolled" above the Curt line. In either case, one can edit the Curt line in a <further interaction> and repeat the process. In this interaction, all of the underlined names created earlier in this line will still be valid and so on.

When the RESULT key is pressed, the current call of the curt procedure is ended with the result of the <Expression> as the result of the procedure.

There are other two other ways of ending a <Curt line>; one can do a declaration or press the DEL LINE key. In both these cases the text of the line disappears, the underlined names are forgotten, and one is ready to start a new <Curt line>. In the case of finger trouble, the previous line can be reinstated together with its underlined names by pressing the INS LAST-key.

A <Declaration> will result in calls of the keep parameter of the current call of curt, with the INT parameter being the number of = symbols. In the case of a simple <identifier> on the right-hand side of the declaration, the other parameters are simply the identifier and the value-mode pair given by the evaluation of the <Expression>. In the case of a list of identifiers on the right-hand side, the mode of left-hand side must be a structure with the same number of components as the number of identifiers in the list. The procedure keep is called that many times with the corresponding identifier and component value-mode pairs.

4.3 Procedure calls

```
<Expression>  ::=  <Unit>
              ::=  <Procedure call>

<Procedure call>  ::=  <Unit> <Expression> !

<Call with identifier parameter>
                ::=  <Unit> . <identifier>
```

A <Procedure call> is indicated by the !-symbol. The <Expression> here is the procedure value and the <Unit> will be the parameters of the call. If the mode of the <Expression> is Filed X then the load procedure is applied to its value to give a new value of mode X, and this value is now considered to be the value of the <Expression>. The value of the <Expression> must either be a procedure or a generic procedure and the mode of the <Unit> parameter must match the formal parameter in the manner indicated in 3.2. The procedure is then called with this parameter. If the procedure call is evaluated correctly, the answer to the procedure is indicated by the underlined name consisting of the Curt text of the call (including the !-symbol) underlined in situ. The mode of the answer is the answer-mode of the procedure unless this is Moded, in which case value and mode of the answer are given by the value and mode fields of the Moded value-mode pair.

If the procedure fails in its operation, the !-symbol is replaced by a ?-symbol, the whole underlined text of the call is is a name for the diagnostic value of this failure, and no further evaluation takes place on this line without further interaction.

In the <Call with identifier parameter>, the <Unit> must be convertable as above, to a procedure with a Vec Char parameter. The procedure is called with the <identifier>, considered as a Vec Char, as parameter. The underlining conventions are the same except in the case of an error, it is the .-symbol which is overwritten by the ?-symbol. In other respects, a call expressed as f.abcd is the same as one expressed as "abcd" f!

4.4 Units

```
<Unit>  ::=  <Denotation>
        ::=  <identifier>
        ::=  <underlined characters>
        ::=  <Call with identifier parameter>
        ::=  <Structure pack>
        ::=  <Vector pack>

<Structure pack>  ::=  ( <<Expression>-list-option> )

<Vector pack> ::= [ <<Expression>-list> ]
```

An <identifier> causes the find procedure to be called to find the value currently associated with the identifier. If such a value exists then the identifier will be underlined in the same manner as procedure calls. If no such value exists (indicated by the call of find failing), evaluation is stopped for further interaction.

If <underlined characters> is the result of some earlier evaluation on this line then the curt procedure will have remembered its value. Otherwise the <underlined characters> is treated exactly as an <identifier>.

The mode of the value formed by a <Structure pack> is the structure mode formed by the modes of its component <Expression>s in order and its value is just the concatenation of their values. The empty pack corresponds to the Void value.

Each of the component <Expression>s of a <Vector pack> must have the

the same mode. If this is X, say, the mode of the whole is Vec X. The value is a vector containing the <Expression>s in order, with upper bound equal to their number.

4.5 Denotations

```
<Denotation>  ::=  minus-symbol-option  <decimal integer>
              ::=  minus-symbol-option
                      <base upto 16> R <based integer>
              ::=  minus-symbol-option  <Real denotation>
              ::=  <Denotation for Vec Char>
              ::=  <Denotation for n * Char>
              ::=  <Denotation for n * Bool>

<Real denotation>
               ::=  <decimal integer> . <decimal integer>
                       <<Exponent>-option>
               ::=  <decimal integer> <Exponent>

<Exponent>   ::=  E-symbol <decimal integer>


<Denotation for Vec Char>
               ::=  " <sequence of characters repeating any "> "

<Denotation for n * Char>
               ::=  { <sequence of n characters repeating any }> }

<Denotation for n * Bool>
               ::=  less-than-symbol
                      <any sequence of n t- or f-symbols>
                      greater-than-symbol
```

The character denotations employ the usual Algol68 trick of doubling up any "-symbols.

It is hoped that the syntax of the denotations given above is self-explanatory.

5. Syntax summaries

In the syntax given in this paper, the following conventions have been observed. Non-terminals are indicated by < > brackets. Those which have an explicit expansion have names starting with an upper-case letter e.g. <Declaration>; others are meant to be self-explanatory, e.g. <identifier>. Expansions are indicated by the ::= symbol. Terminals ending -key or -symbol are the appropriate key stroke and other symbols stand for themselves. Non-terminals ending -list and -option are as follows:

```
<X-list>  ::=  X
          ::=  X , <X-list>

<X-option>  ::=  X
            ::= <empty>
```

5.1 Curt syntax

```
<Curt line>  ::=  <Declaration>  DO-key
             ::=  <Expression> DO-key <further interaction>
             ::=  <Expression> RESULT-key
             ::=  <any old rubbish>  DEL LINE-key
             ::=  INS LAST-key

<Declaration>  ::=  <Expression> = <Idset>
               ::=  <Expression> == <Idset>

<Idset>  ::=  <identifier>
         ::=  ( <<identifier>-list> )


<Expression>  ::=  <Unit>
              ::=  <Procedure call>
```

```
<Procedure call>  ::=  <Unit> <Expression> !

<Call with identifier parameter>
                   ::=  <Unit> . <identifier>


<Unit>  ::=  <Denotation>
        ::=  <identifier>
        ::=  <underlined characters>
        ::=  <Call with identifier parameter>
        ::=  <Structure pack>
        ::=  <Vector pack>

<Structure pack>  ::=  ( <<Expression>-list-option> )

<Vector pack>  ::=  [ <<Expression>-list> ]

<Denotation>  ::=  minus-symbol-option  <decimal integer>
              ::=  minus-symbol-option
                      <base upto 16> R-symbol <based integer>
              ::=  minus-symbol-option  <Real denotation>
              ::=  <Denotation for Vec Char>
              ::=  <Denotation for n * Char>
              ::=  <Denotation for n * Bool>

<Real denotation>
                ::=  <decimal integer> . <decimal integer>
                        << Exponent>-option>
                ::=  <decimal integer> <Exponent>

<Exponent>  ::=  E-symbol <decimal integer>


<Denotation for Vec Char>
                ::=  " <sequence of characters repeating any "> "
```

```
<Denotation for n * Char>
                  ::=  { <sequence of n characters repeating any }> }

<Denotation for n * Bool>
                  ::=  less-than-symbol
                         <any sequence of n t- or f-symbols>
                         greater-than-symbol
```

5.2 Syntax of modes

The following is a syntax of Curt modes as recognised by the procedure make_mode. Any potential ambiguity in a mode should be clarified by the insertion of parentheses which have no meaning other than grouping.

```
<Mode>  ::=  Int
        ::=  Word
        ::=  Real
        ::=  Char
        ::=  Bool
        ::=  Any
        ::=  Moded
        ::=  <atomic mode>
        ::=  Ref <Mode>
        ::=  Vec <Mode>
        ::=  <<integer>-option> * <Mode>
        ::=  [ <<empty>-list-option> ]  <Mode>
        ::=  Filed <Mode>
        ::=  Disc <Mode>
        ::=  Ptr <Mode>
        ::=  <Mode> -> <Mode>
        ::=  ( <<Mode>-list-option> )
        ::=  Union ( <<Mode>-list> )
        ::=  Choice ( <<Mode>-list> )
        ::=  <Generic mode>
```

```
<Generic mode> ::= <generic parameter> => <generic answer>
```

An <atomic mode> is represented by an identifier starting with an upper case letter, different from the ones already reserved in the syntax. The procedure make_mode will call the curt procedure find, with this identifier as its parameter and it expects to find that the mode of its answer is Mode. This identifier would usually have been associated with a mode by using the procedure new_atomic.

The expansions of <generic parameter> and <generic answer> are simply that of <Mode>, removing the possibility of <Generic mode> as an alternative and including the alternatives giving the formal modes ?, ??, ??? etc.

References

[1] I F Currie, J M Foster, P W Edwards, Flex Firmware , RSRE Report No 81009

[2] I F Currie , In praise of procedures, RSRE Memo No 3499

[3] J M Foster, I F Currie , P W Edwards , Flex : a working machine with an architecture based on procedure values, RSRE Memo No 2500

Acknowedgements

The authors would like to thank all the members of C2 division, RSRE, for all their help and suggestions, both in the preparation of this paper and in the work itself.