A Reformed EXecutor - REX

Initial specification - 29Mar79

Mike Cowlishaw: (WINPA MFC) Hursley MP 182.

1. Introduction.

The CMS exec language (which has since been extended and improved upon by EXEC 2) is based on the common macro language principle that variables and controls should be distinguished (by "&") and literals should exist in 'plain text'.

When EXEC's were mainly consisting of strings of commands, with very little logic in between, this was a fair and sensible choice: however a quick scan through the EXEC's of almost any modern user quickly shows that the majority of tokens in use are symbolic (that is, they begin with "&"). This observation must cast serious doubts on the validity of using this syntax.

A further argument is the increasing use of "complicated" strings in EXEC's: for example embedded blanks are increasingly used in Editor Macros; full screen displays; and so on. EXEC 2 handles these only fairly well, whereas EXEC cannot manipulate them at all: the user is reduced to unreadable manipulations of the Underscore character to achieve the desired result.

There is perhaps at least some justification in investigating an alternative macro command language which uses the 'more conventional' notation used by the higher level programming languages such as PL/1; PL/S; PASCAL; and so on. The PL/1 Preprocessor is an example of this type of macro language, and in an IBM environment will strongly influence the syntax that would be considered acceptable.

The use of this type of notation will naturally cause users to draw comparisons with the normal programming languages. This inevitably will lead them to expect a corresponding improvement in the facilities available in the command language: this in turn certainly implies that the interpreter will be larger and slower than either EXEC or EXEC 2. Size (within reason) is not often a problem on modern virtual machines, however a performance penalty may be unacceptable in some environments. It would be hoped that by using a subset of the available facilities, the REX user would be able to attain at least a comparable performance.

What then are the major desirable features for a command macro language? My choice includes:

1) Structured flow control statements, some equivalent of
   If-then-else, Do (until/while)-end, Select-when-end
   being the most important.
2) Free format: not line-by-line
3) Case translation on output only: comparisons case-independent
4) Literal shorthand: unknown 'tokens' assumed to be enclosed

in quotes.

5) 'Complex' expressions (i.e. parentheses, multiple operators)
6) In line 'function' calls to other EXEC's or MODULEs
7) No requirement for self-modifying EXEC's

The next section discusses these topics in more detail, however the impatient (or busy) reader may prefer to skip to the language definition in section 3.

2. The language features

The following items are not intended to be rigorous definitions of the language features; and some implicit assumptions about the language syntax and the host system will be apparent. They are rather general descriptions of the syntax and the decisions leading to each choice.

2.1 Structured flow control statements.

The need for structured flow control is accepted by most programmers. The three main classes of structured flow control are the If-then-else; Do (while/until)-End; and Select-when-end. (I use the IBM (PL/1) constructions rather than any of the possibly superior alternatives described in the literature purely for consistency.) If-then-else has been implemented for EXEC by using modules; EXEC 2 has Do-While and Do-Until; but neither has any form of select (case) structure. Evidently all these features are desirable for any modern language.

2.2 Free format: not line-by-line.

A free format statement is possibly more general than fixed (line-by-line) format. The latter option implies a record-orientated file system, whereas the former is applicable both to record and character stream files or input devices. By the same token, a free format structure would in general permit better self-documentation of EXEC's, since comments could occur almost anywhere in the input stream.

This decision puts some restrictions on possible random-access of REX files, which would no longer be possible. See section 2.7.

The obvious statement delimiter to use would be ';', with /*...*/ or (*...*) for comments.

2.3 Case translation on output only.

The operating system underlying the REX interpreter may require that commands be passed in Upper case. This perverse restriction may force REX to translate strings output to the host system. Despite this, there is no necessity for any commands input to REX to be translated (always or sometimes). A cleaner and more natural algorithm is that a) no variables or input data will be translated, and b) all logical comparisons will be case-independent. Thus the expression " 'YES' = 'Yes' " will result in the value '1' ("True").

2.4 Literal shorthand & Blank operator.

A convenient convention for a macro language is that of 'literal shorthand'. My definition of this would be: If a symbol is unknown (i.e. not a variable, REX keyword, or function call) then it is assumed to represent a literal string consisting of the characters of the symbol.

A further convenient extension to 'standard' syntaxes is the

concept of the 'Blank' operator. This may be defined verbally thus: If two expressions (ie symbols, literals, etc) are separated by one or more blanks and no other operator then the operation of 'concatenate with a blank in between' will be performed. For example 'A' 'B' would be evaluated to 'A B'.

The effect of these two conventions allows a syntax that combines the advantages of both EXEC languages and the PL/1 like model. Consider the following excerpt from a REX EXEC (assume that Fn, Ft, Fm are symbols representing variables previously set up by assignments etc):

```
State fn ft fm;
If retcode=0 then erase fn ft fm;
```

which is more readable than the equivalent 'Strict.Pl/1' form:

```
'State'||' '||fn||' '||ft||' '||fm;
If retcode=0 then 'Erase'||' '||fn||' '||ft||' '||fm;
```

Note that a statement which is an expression on its own is a passed to the host system as a command.

2.5 'Complex' expressions.

Compound character and arithmetic expressions really are a necessity for any language. Even simple assemblers usually allow more than one operator in their constant expressions: so why shouldn't a command interpreter? There are three possible implementations of compound expressions: a) simple Left -> Right (or Right -> Left - APL) scanning; b) Reverse polish notation (e.g. FORTH); c) full algebraic, with parentheses and operator priorities.

Option b) is probably unacceptable to the IBM user, and is also somewhat outdated as a solution. Option a) is a considerable improvement on no compound expressions at all, but is not ideal - especially as logical operations should be treated as normal operators, rather than special cases.

Option c) is of course the best, and need not necessarily be significantly more complicated than a). The algorithms and techniques are well understood, and an EXEC interpreter must necessarily include storage management routines which should be able to handle stack(s).

I would consider the minimum set of primitive dyadic operators to include: + - * / || and ' ' as defined above, together with the logical operators = ¬= > < >= <= .

2.6 In line 'function' calls to other EXEC's or MODULEs.

The ability to define in-expression functions greatly increases the power of a language, and reduces the need for specialised builtin functions.

The host system is assumed to include at least one command executor and a stack/queue of some kind. A sub-class of commands (EXEC's or MODULE's ?) are those which accept arguments only from a (the) stack, and put their results back on the stack. This subclass

can be termed 'functions' and included in the REX language using the conventional notation of parentheses with commas to separate the argument expressions.

For example if the function 'SUBSTR' were not built-in, it could be implemented by a separate MODULE (or subcommand, or EXEC); with the arguments being the three top items on the stack, and returning the result string on the stack.

The syntax description would therefore be: If a symbol is followed immediately by a "(" then it is taken to be a function name. each expression following the "(" and separated by "," is stacked, and the function is invoked when the final ")" is interpreted.

This gives a 'normal' syntax for function calls, without the need for a new statement for every command.

Note that since 'blank' is a valid dyadic operator, there must not be a blank between the function name and the "(".

2.7 No requirement for self-modifying EXEC's.

EXEC and EXEC 2 both permit self-modifying EXEC's. This is a "nice" facility which however is typically not used. In fact, the only time it normally occurs is when one edits an 'EDIT' exec: and then it is usually more of an embarrasment than a help.

REX would therefore assume that all EXEC's are READ ONLY. This could imply: a) the entire EXEC could be read in in one file system operation (inefficient for long files, perhaps); and b) statements that might be re-interpreted (e.g. in loops) could be part compiled for improved performance.

This language restriction also opens up the attractive possibility of compilation or part compilation of the language: a possible implementation might therefore consist of a 'compiler' which produces an 'object file' which could then be very efficiently interpreted by the REX EXEC processor, with real performance improvements.

3. The language definition

Note: This definition is not a totally rigorous description of the syntax. It is amenable to change, and suggestions for changes are most welcome.

3.1 Tokens and statements.

The language (hereafter called REX) is composed of tokens (of any length, up to an implementation restricted maximum) which are separated by blanks or by the nature of the tokens themselves. The classes of tokens are:

Symbols: Groups of alphameric characters.
"Alphameric" may include certain non alphabetical or numeric characters.
A meaning of a symbol is context defined: once one has appeared as the target of an assignment statement (see below) it is a variable. If it is immediately followed by a "(" it is the name of a function. If it is a REX command keyword, it is interpreted as such. If it is none of these it is considered to be a string composed of the letters of the symbol with a quote added before and after.

(Integers: Groups of numeric characters: a subclass of symbols)

Strings: a string including any character and delimited by the single quote character ('). Two quotes ('') must be used to include a single quote in the string.

Operators: Groups of one or more non-alphameric characters.

Comments: Strings delimited by /* ... */ or possibly (* .. *). Comments are entirely ignored by the interpreter.

For example the data " 'A'+B " is composed of three tokens: a string, an operator, and a symbol.

Each language statement is composed of: 0 or more blanks (which are ignored); a sequence of tokens; 0 or more blanks (again ignored); and the delimiter ';' (semicolon).

Within the sequence of tokens; comments (and separator blanks that are adjacent to operators or other blanks) are ignored. Any blanks left in the data are then valid (concatenation with blank) operators (see below). The statement data is then interpreted.

The statement data may be:

* Null - which has no effect

* An assignment - of the form "Symbol=expression" (section 3.3)

* A host command - consisting of an expression. (section 3.4)

* A REX command - starting with a keyword symbol (section 3.5)

3.2 Expressions and operators.

Nearly all statements may include expressions which consist of integers, symbols, or strings interspersed with operators and possibly parentheses. I shall not bore the reader with yet another BNF-like description of expressions, but will just give some examples.

Suppose that the following symbols represent variables; with the values as shown:

```
A   has the value '3'
DAY has the value 'Monday'

Then:

"A+5"          will evaluate to '8'
"Today is DAY"       ==>       'Today is Monday'
"(A+1)>7"            ==>       '0'              /* False */
"(A+1)=4"            ==>       '1'              /* True  */
"'If it is' Day"     ==>       'If it is Monday'
"Substr(Day,2,3)"    ==>        'ond'    /* Substr is a function */

The valid operators could be:

+ - / *  := (usual meanings)
||         := Concatenate
' ' (blank) := Concatenate with blank (||' '||)
= ¬=       := Equals, Not Equals  )
< > <= >=   := LT, GT, LE, GE      ) Logical operations result in
& | &&      := AND, OR, EXOR       )   '1' or '0'.
```

3.3 Assignment statements

An assignment is always of the form:

"symbol=expression"

The symbol is any valid collection of alphameric characters (as described above) and including those beginning with a number (0-9). By being the target of an assignment in this manner, it is contextually declared as a variable: in other words, in all succeeding statements this particular collection of characters represents a string in storage.

Note: since an expression may include the operator '=', and a statement may consist purely of an expression (see next section), there is a possible ambiguity here. REX will therefore take any statement whose second token is '=' to be an assignment statement, not an expression.

3.4 Host commands

The 'TARGET MACHINE' for REX is assumed to include at least one stack/queue, and at least one port for executing commands. One of these ports is assumed to be the 'main' port.

Executing commands through the main port may be achieved using a statement of the form:

```
"expression"
```

The expression is evaluated, resulting in a character string, which is then prepared as appropriate and submitted to the Host. For example, if the host were a CMS system, the string would be 8-byte tokenised, and possibly the first token would be translated to upper case.

This sort of manipulation would be carried out by an interface routine not strictly part of REX.

As an example of how a CMS command might be issued, the sequence:

```
fn=JACK; ft=RABBIT; fm=A1;
State fn ft fm;
```

would result in the PLIST: "STATE JACK RABBIT A1 " being submitted to CMS. Of course, 'State JACK RABBIT A1' would have the same effect in this case.

3.5 REX Commands.

Several of the more powerful features described above (notably functions and case handling) reduce the number of primitive REX commands and built-in functions that are needed.

A further assumption, that all input parameters to a REX program will be available as the top item in the Stack, also simplifies this requirement, and does away with the special 'parameter variables' (&1 - &n in EXEC). There will always be this top-of-stack string on entry to a REX program: it may have the value null.

This section describes a minimum set of commands which would at least allow the facilities of EXEC (and nearly all of those of EXEC 2, the exceptions being the special-case subcommand and environment statements, which could be easily added: perhaps in a more general form).

Note that the choice of keywords is fairly arbitary: as noted earlier, there are certainly arguable alternatives to most of them.

In the following diagrams, symbols (words) in capitals denote keywords, other words (such as 'expression') denote a collection of symbols as defined above. Note however that the keywords are not case dependent: the symbols "if", "If" and "iF" would all invoke the command shown below as "IF".

The characters < and > delimit optional parts of the commands.

```
IF expression THEN statement <ELSE statement>
```

If the expression evaluates to the single character '1' then the statement following the THEN is executed. For any other result, the ELSE statement is executed.

Note that 'statement' may include a Do Group (list of statements).

```
DO <loop-condition>; statement-list END

where  loop-condition is: WHILE expression
                      or: UNTIL expression
and    statement-list is any list of statements
```

If no loop-condition is given, then the statement-list will be executed once.

Otherwise the expression is evaluated, and the statement-list will be repeatedly executed either While the expression='1', or Until the expression='1'.

```
SELECT expression; when-list <OTHERWISE statement> END

where  when-list   is: 1 or more when-clause
and    when-clause is: WHEN expression; statement
```

The expression following the SELECT is evaluated. Each expression following a WHEN is evaluated in turn and compared with the first expression: if identical, the following statement is evaluated and control will pass to the END.

If none of the WHEN expressions match, control will pass to the statement following OTHERWISE. In this situation, the absence of an OTHERWISE will cause an error.

```
QUEUE expression
```

The string resulting from expression will be stacked FIFO.

```
PUSH expression
```

The string resulting from expression will be stacked LIFO.

```
PULL symbol-chain

Where symbol-chain is a list of symbols separated by blanks
or operators.
```

The current top-of-stack will be read as one string. It will then be parsed according to the normal rules, and tokens assigned to the symbols given in sequence. This contextually declares the symbols given to be variables.

If there are less symbols in the symbol-chain than there are tokens in the string; the final symbol will have the remainder of the input string assigned to it as a single unedited string.

Thus in the limiting case of there only being one symbol specified, it will be assigned the entire input string.

The function of the operators in the symbol chain is to force synchronisation between the input tokens and the symbols given. In the obvious CMS-like example, if the input string is: 'AAA BBB (CCC', and the symbol chain is 'Fn Ft Fm (O1' then the two chains will be synchronised at the '(', and the variable 'Fm' will have the value '' (null).

The REX variable 'N' will be set to the number of tokens or strings assigned values in the PULL command.

```
SAY expression
```

The string resulting from expression is displayed (or spoken, or typed, etc) to the user via whatever channel is available.

```
EXIT   expression
```

The expression is converted to a number, and execution is terminated with the number being used to set the returncode.

```
RETURN expression
```

The expression is PUSHed onto the stack, and execution is terminated with returncode set to zero.

```
TRACE expression
```

The appropriate action is taken according to the value of expression:
'ON'   - all statements are traced
'ERROR' - host commands resulting in non-0 returncode are traced
anything
  else - no statements are traced

```
ERROR statement
```

If any following command returns a non-zero returncode, the given statement (which may be Do-End, etc) will be executed.

4. Built in functions and variables.

Since REX can include functions directly in expressions, the need for built-in functions in reduced. The more complicated ones (such as SUBSTR and INDEX) would probably not be included in an initial implementation. Even 'DATATYPE' could be initially implemented as an external routine.

Certain symbols will have predefined meanings - these are a conventional selection.

BLANKS	A full length string of blanks
DATE	Current Date
N	Number of symbols assigned values in the last PULL
NL	The 'New line' character (EBCDIC X'15')
Q	'1' if there is anything in the queue (Stack), else '0'
RC)	Return code from last host command
RETCODE)	
TIME	Current Time

5. Goto's and labels.

A GOTO statement and its corresponding labels are obvious omissions in the language as defined above. There is no reason why labels (symbol followed directly by ':') and a "GOTO expression" should not be implemented.

Thoughts on the desirability of this are solicited from the reader, though the author feels they should probably be added, as the 'line of least user resistance'.

6. Rescanning and arrays.

EXEC and EXEC 2 both allow a partial rescanning facility to allow subscripted variables to be manipulated. This is one of the problem areas in REX.

Arrays can be easily simulated with a SUBSTR function, or with specially written array functions, however as REX does not permit pseudo-variables, this is clumsy.

An associated problem is due to the simplified parameter handling: since we do not have the variables &1...&n, certain operations (such as scanning the argument list for keywords) are difficult. A WORD(n) function would solve this problem fairly neatly.

Alternative solutions include:

a) a 'concatenate and rescan' operator to specifically handle subscripts (messy).
b) built in array handling (implies some form of Declare statement)
c) an 'REX' command, to execute the given string as though it were a one-record REX program. (Powerful).

What are your thoughts?

7. Example EXEC's for CMS using REX.

ADDR EXEC

```
/* Displays full address and name for nicknames specified */

Do until rest='';
 Pull Nick Rest;

 If n=0 | Nick='?' then /* tell */ do;
  Say 'Correct form:  ADDR name1 <name2 <name3 ....>>'
   nl
   nl 'ADDR searches your rmsg file for the specified nickname.'
   nl 'If it finds the name, it displays the actual system and userid'
   nl 'of the user.  If the name is not found, it checks for a local'
   nl 'userid with the same name.';
  /* Multiple help for multiple ?'s !! */
  end;
 else /* we have a nickname */ do;
  Push HT;
  STATE Nick DISTRIB *;
  Push RT;
  If RC=0 then Say nick is a distribution list;
   else /* not a list */ do;
    SCANRMSG nick;
    rr=rc; /* save */
    FINIS * RMSG *;
    if rr=0 then do;
      /* some data was stacked */
      Pull nn node uid via n1 n2 n3 n4 n5;
      if uid='' then say
        Nick is the nickname for the local user node;
       else say
        Nick is the nickname for n1 n2 n3 n4 n5 (uid at node);
      end;
     else /* nothing was stacked, might be a local userid */ do;
      CPCOMM TRANSFER CL 1 FROM Nick;
      Pull; Pull;  /* clean stack after CPCOMM */
      If RC=0 then say Nick is a local VM id;
             else say Nick is an unknown name;
      end;
    end /* we had a nickname */;
  if rest¬='' then push rest;
  end /* until rest='' */;
```

SEND EXEC (from the EXEC 2 documentation):

```
/* Send file to a local user */
Pull name fn ft fm;
if name='' | name='?' then do;
  say 'Command is: SEND User Filename Filetype <Filemode>';
  exit 100;
  end;
if n<3 | n>4 then do;     /* Check number of arguments */
  say Bad SEND command;
  exit 101;
  end;
if fm='' then fm='*';     /* assume ANY if no mode given */
CP SPOOL PUN name CLASS A;
if rc¬=0 then do;         /* check SPOOL worked */
  say name is not a valid userid;
  exit 102;
  end;
PUNCH Fn Ft Fm;
if rc¬=0 then do;         /* check PUNCH worked */
  say Error rc 'from "PUNCH" (while in SEND)';
  nn=102;
  end;
 else /* Tell recipient what has been done */ do;
  CP MSG Name I Have just punched you my file Fn Ft Fm;
  nn=0;
  end;
CP SPOOL PUN * CLASS A;
Exit nn;
```

```
/* MOVE: File Copy + Erase */
Pull Fn Ft Fm (x nfn nft;
If n=0
   | Fn='?' then do;
  say
   nl'Format: MOVE Fn  < Ft < Fm >>   < (X <Nfn <Nft >>>'
   nl'        where "X" is the disk to move to.'
   nl'        Any of Fn/Ft/Fm may be specified as *'
   nl'        Additionally, you may specify the name the file is to be'
   nl'        known by on the new disk by giving "Nfn Nft". ';
  exit; end;

If Fn='' then do;
  Say 'WHAT am I meant to move ?  Next time give me a filename!';
  Exit 28;
  end;

if Ft='' then Ft='*';  if Fm='' then Fm='*';
FINDFILE Fn Ft Fm;
If RETCODE¬=0 do;
  pull; /* Clean up after findfile */
  say 'Nothing to move!';
  exit 8;
  end;
pull fn ft fm; /* Get fn etc to use */
/* now check target */
if nfn='' then nfn=fn;
if nft='' then nft=ft;
if nfm='' then do;
  nfm=bigdisk();
  Say Move FN FT FM to disk NFM - '(Enter NULL, MODE, or "QUIT")';
  Pull ans;
  if ans='Quit' then exit;
  if ans='' then nfm=ans;
  end;

odk = substr(fm,1,1);
ndk = substr(nfm,1,1);
If NDK=ODK then do;
  Say '   You may not move the file onto itself.';
  Say '   NB:  Source disk could be a R/O extension of target';
  Exit 32;
  end;
STATE NFN NFT NFM ;
IF RC¬=28 then do;
  if retcode¬=0 exit retcode;
  Say nl NFN NFT NFM 'already exists: type "Y" to replace';
  pull ans;
  if ans ¬='Y' then do;
    Say  File not moved ;
    Exit;
    end;
  end;
```

```
ERROR Exit RETCODE;
FCOPY FN FT FM NFN NFT NFM;
Say  NFN NFT now on disk NFM;
ERASE FN FT FM;
Say  FN FT erased from disk ODK;
Say;
```