<DORNBUSH>CML.NLS;8, 3-OCT-73 16:05 CFD ;

COMMAND META LANGUAGE -- CML

INTRODUCTION

The command meta-language (CML) is a vehicle for describing the syntax and semantics of the user interface to the NLS system. The syntax is described through the tree-meta alternation and succession concepts. The semantics are introduced via built-in functions and semantic conventions.

No attempt is made to describe the full semantics of any command via CML, but it is hoped that the front-end interface (parsing and feedback operations) may be explicitly accomodated with these facilities. It will still be necessary, and desirable, to use execution functions to perform the low-level semantics of the command. The CML describes how the command "looks" to the user, rather than what it does in the system.

USE OF CML

The user interface for the NLS command language is defined in the CML specification language. This "program" is then compiled by the CML compiler (written using ARC's tree-meta compiler compiler system) to produce an interpretive text which drives a command parser. The command parser is cognizant of the device dependent feedback and addressing characteristics of the user's i/o device.

ELEMENTS OF CML

PROGRAM STRUCTURE

The basic compilation structure of a CML program is described by:

```
file            = "FILE" .ID [system] (dcls / rule)
                  #subsys "FINISH";
system          = "SYSTEM" .ID %system name% '=
                  #<'/>.ID %names of subsystems % '; ;
subsys          = "SUBSYSTEM" .ID % subsystem name --  %
                   #(command / rule) "END.";
command         = ("COMMAND"/ "INITIALIZATION" /
                  "TERMINATION") rule ;
rule            =  .ID '=  exp '; ;
```

The "file" construct brackets the definition of command language subsystems and may optionally include the system

definition (which defines all subsystems contained in a particular system).

Parsing rules and declarations may appear at this global level.

The subsystem contruct brackets a set of rules or commands. Commands beginning with the keyword COMMAND are linked together to form a command language subsytem.

The subsystem may include a rule preceded by the keywords INITIALIZATION or TERMINATION. If specified, these rules will be executed once upon system initialization/termination respectively.

Each rule/command is named with an identifier. This name is a global symbol and should not conflict with any other variable names, rule names, or keywords.

DECLARATIONS

Declarations are used to associate attributes with identifier names which are used in cml programs. If not declared, identifiers are defined by their first occurrence according to the following rules.

1) Identifiers appearing on the left hand side of an assignment statement are defined as "VARIABLES".

2) Identifiers followed by a subscripted list are assumed to be of type "FUNCTION".

3) All other undefined identifiers are assumed to be names of parse rules or commands.

The syntax of the declare statement is given by:

```
dcls      = ("DCL" / "DECLARE") [dclattr] #<',> .ID;

dclattr   = ("VARIABLE" / "FUNCTION" / "PARSEFUNCTION");
```

If a declare attribute is not given, type VARIABLE is assumed. Identifiers which are implicitly defined as type variable are EXTERNAL symbols and will be linked by the loader to externally defined symbols with that name.

RECOGNIZERS

Keyword Recognition

The process of keyword recognition is independent of the description of the keywords for CML. In the CML description, each keyword is represented by the full text of the keyword. The algorithm used to match a user's typed input against any list of alternative keywords is known as keyword recognition, and is a function of the

command interpreter and is independent of the CML description of the command.

Keywords are written in the meta language as upper-case identifiers enclosed in double quote marks optionally followed by a set of keyword qualifiers.

```
keyword = .SR [ '! #qualifier '! ]
```

The qualifiers serve to control the recognition process for the keywords and to override the system supplied internal identification for the keywords.

```
qualifier       = "NOTT"          % DNLS only keyword %
                 /"NOTD"          % TNLS only keyword %
                 /"L1"            % first level keyword %
                /.NUM             % explicit value for
keyword %
```

Selection Recognition

Three types of selections are built into CML. They are DSEL, SSEL, and LSEL (see -- <userguides,commands,1> for the explicit definition of the selections). Basically, they are recognizers which require some entity type as an argument and they return a pair of text pointers in the state record. The entity type is obtained either by some previous invocation of the recognition function for some list of keyword entities, or use of the VALUEOF built in function.

The DSEL, SSEL, and LSEL functions perform all evaluation and feedback operations associated with the selection operations.

```
selection       = ("SSEL"/ "DSEL"/ "LSEL") '( param ')
```

Other Recognizers

The processes of viewspec recognition, level adjust recognition and command confirmation recognition are represented in CML by built-in parameterless functions in the meta-language.

```
others          = "VIEWSPECS"   % viewspec collection %
                 /"LEVADJ"      % leveladj collection %
                 /"CONFIRM"     % command confirmation %
```

FUNCTION EXECUTION

Functions may be invoked at any point in the parse by writing

a name of some routine and enclosing a parameter list in parentheses. All functions invoked by the interpreter must obey the groundrules set up for interpreter routines. The actual arguments are passed by address, rather than value, and two additional actual arguments are appended to the head of the argument list.

```
control    = .ID % routine name % '( $<',> param ')

param      = factor          % expression element %

           / "VALUEOF" '( .SR )    % keyword value %

           / '# .SR          % same as VALUEOF %

           / "TRUE"          % boolean TRUE value "

           / "FALSE"         % boolean FALSE value "

           / "NULL"          % null pointer value %
```

PARSING FUNCTIONS

Functions which are declared with the PARSEFUNCTION attribute are assumed to be parsing functions. They are called in "parsehelp" mode and when so called, are passed the address of a string as a third argument. The parsefunction routine then supplies a prompt string which tells what the parsing functon does. (see appendix 3 for example). Parse functions may appear as alternatives to non-failing recognizers and may themselves fail. Them must however, precede any non-failing recognizers in the list of alternatives.

FEEDBACK CONTROL

The feedback control elements of CML are used to provide feedback in addition to the normal feedback generated by the recognizers. This is used to implement additional "noise words" and help feedback.

1) adding feedback to the command feedback line.

   A string may be added to the current command feedback line by enclosing the quoted string in angle brackets.

   ```
   extra feedback = '< .SR '>
   ```

2) replacing the last word in the feedback line.

   It is possible to replace the last string in the command feedback line by using the string replace facility. This is similar to (1) above except the previous word in the feedback line is deleted before adding the new string.

```
replace extra feedback = '<"..." .SR '>
```

A function is also provided to initialize the command feedback mechanisms and clear the command feedback line.

```
clear cfl  = "CLEAR"
```

EXPRESSION DEFINITION

CML is an expression languge. Commands are defined to be a single expression and expressions are composed of successive/alternative expression factors. Alternative paths are indicated by the character '/ in the expression.

The nesting of expressions may be explicitly defined with parenthesis and brackets are used to delimit optional expression elements. The dollar sign preceeding an optional construct is used to indicate that the optional element is repeated as long as the option character is typed in.

```
exp                 = #<'/>alternative;

alternative         = #factor;

factor              = term
                     / '( exp ')
                     / '[ exp ']     % optional element %
                     / '$ '[ exp '] % repeated opt elements
%

term                = subname       % id/ assign/ function %
                     / confirm      % command confirmation %
                     / feedback     % noise word feedback %
                     / recognition  % built-in recognizers %
```

COMPLETE FORMAL SYNTAX OF CML

```
file                = "FILE" .ID [system] $(rule/ dcls)
                      #subsys "FINISH";
system              = "SYSTEM" .ID %system name% '=
                      #<'/>.ID %names of subsystems % '; ;
subsys              = "SUBSYSTEM" .ID % subsystem name --  %
                       #(command / rule) "END.";
command             = ("COMMAND" / "INITIALIZATION" / "TERMINATION")
```

```
                    rule ;
    rule            = .ID '=  exp '; ;
    dcls            = ("DCL" / "DECLARE") [declattr] #<',> .ID;
    dclattr         = ("VARIABLE" / "FUNCTION" / "PARSEFUNCTION");
    exp             = #<'/>alternative;
    alternative     = #factor;
    factor          = term/ '( exp ')/ '[ exp ']// '$ '[ exp '];
    term            = subname/ confirm/ feedback/ recognition;
    subname         = .ID  [ '← param/ '( $<',>param ')];
    confirm         = "CONFIRM"; % call routine to terminate cmd %
    recognition     = keyword/ builtinrec;
    keyword         = .SR [ '! #qualifier '! ];
    qualifier       = "NOTT"/ "NOTD"/ "L1"/ .NUM;
    builtinrec      = (("SSEL"/ "DSEL"/ "LSEL") '( param '))
                    / "VIEWSPECS"/ "LEVADJ";
    feedback        = "CLEAR"/ '< ["..."] .SR '>;
    control         = .ID '( $<',>param ');
    param           = factor/ ("VALUEOF" '( .SR ') / '# .SR)
                    /"TRUE"/ "FALSE"/ "NULL";
```

THE INTERPRETIVE TEXT

Each instruction of the interpretive text contains a structure word at least one function execution word. The structure word defines the alternation and successor paths of the grammar for the command language. The function execution words perform the actions of the interpreter.

The structure words

Each structure word consists of two pointers. The right half of the word defines the alternative node to the current node. The left half of the word points to the successor to the current node. Null paths are indicated by 0 valued pointers.

The executable function word formats

Format 1: /OP CTL MODIFIER ADDR/

This is the only interpreter instruction word format presently defined. OP is an operation code. CTL contains control bits used by the keyword regognition function. MODIFIER may contain an additional value. ADDR is the address or principal value for the function.

The functions of the interpreter.

RECOGNIZERS

KEYOP -- keyword recognition.

CTL = control bits for level 1 commands, DNLS commands, and TNLS commands.

ADDR = address of keyword literal string

The current input text is matched against the keyword string specified by the current node and all alternatives of the current node. This function performs keyword recognition on all of the alternative nodes of the current node simultaneously.

This function cannot fail. Control remains in the keyword recognition function until appropriate input is recognized or until the control is abnormally wrested via backup or command delete functions.

The value returned in the argument record is a single word containing the address of the string corresponding to the keyword actually recognized.

CONFIRM -- process command confirmation characters

This function interrogates the input text for one of the command confirmation characters. Control remains in this routine until a proper confirmation is recognized, and command termination state is appropriately set. This function always returns TRUE.

The value returned is a single word containing a command completion code which identifies the completion mode.

SSEL -- get a source selection

ADDR = not used

The sselect routine is invoked to process a source type selection. The return record contains two text pointers which delimit the selected entity.

DSEL -- get a destination selection

ADDR = not used

The dselect routine is invoked to process a destination type selection. The return record contains two text pointers which delimit the selected entity.

LSEL -- get a literal selection

ADDR = not used

The lselect routine is invoked to process a literal type selection. The selection type is passed as an actual argument. The return record contains two text pointers which delimit the selected entity.

VIEWSPECS -- process viewspecs information

The viewspec input routine is called to process the input stream for viewspec characters. The return record contains the two updated viewspec control words. This function always returns TRUE.

LEVADJ -- process level adjust information

The level adjust input routine is called to process the input stream for level adjust characters. The return record contains a single word which indicates the relative level adjust value (u = +1, d = -1, etc). This function always returns TRUE.

CONTROL FUNCTIONS

EXECUTE -- transfer of control to another point in the tree.

ADDR = address of root of tree for transfer of control

The current point in the tree is marked and control is transferred to the node pointed to by the address field. Control remains in the descendent node until it has been completely parsed, at which time control returns to the successor of the EXECUTE node.

CALL -- subroutine invocation

MODIFIER = number of actual parameters

ADDR = address of the subroutine

The appropriate number of actual arguments are popped off of the evaluation stack and passed to the routine whose address is contained in ADDR.

The resultptr from this routine is pushed onto the eval stack if it returns TRUE.

PFCALL -- parsing function invocation

MODIFIER = number of actual parameters

ADDR = address of the subroutine

The appropriate number of actual arguments are popped off of the evaluation stack and passed to the routine whose address is contained in ADDR.

The resultptr from this routine is pushed onto the eval stack if it returns TRUE.

This function is also called in "parsehelp" mode to find out what it does.

OPTION -- test for an optional construct.

If the next input character is the OPTION select character, then it is read and control is transferred to the node at address ADDR. If the next character is not the OPTION character, then control passes to the successor path of the current node.

ANYOF -- collect alternative optional keyword values

If the next input character is the OPTION select character, then it is read and control is transferred to the node at address ADDR. After the descendent nodes have been processed, control returns to the ANYOF node, permitting another optional selection to be made from among the set of alternatives. The result values from the succession of optional recognitions are logically OR'ed together to form the value for the ANYOF node. If the next character is not the OPTION character, then control passes to the successor path of the ANYOF node.

FEEDBACK ELEMENTS

FBCLEAR -- clear the contents of the feedback buffers.

The feedback state information and command feedback line are set to their initial or empty position.

ECHO -- appends a noise-word string to the command feedback link

ADDR = address of the text string to be appened

RECHO -- replaces the last noise-word string in the command feedback line

ADDR = address of the text string which is to replace the last item in the command feedback buffer

VALUE MANIPULATIONS

LOAD -- loads a pointer to an argument record into the top of the eval stck.

ADDR = address of the variable containing the pointer to the argument record.

The pointer value contained in the variable whose address is contained in ADDR is pushed onto the top of the eval stack.

STORE -- saves a pointer to an argument record in a variable

ADDR -- address of the variable

The address of an argument record is fetched from the top of the eval stack and is saved in the variable at address ADDR.

ENTER -- enters a constant value into the argument record pointed to by the top of the eval stack.

ADDR -- value to be entered (18 BITS only)

The value is taken from the ADDR field of the instruction and is entered into the argument record for the ENTER node in the path stack (whose address is at the top of the eval stack).

VALUEOF -- enters the system value for a keyword into the argument record .

ADDR -- address of the KEYWORD string.

The ADDR points to a string variable. The literal area is searched for a match with the argument string and the address of the literal string which matches the keyword string is entered into the argument record for VALUEOF, whose address is pushed onto the top of the eval stack.

FLOW OF CONTROL IN THE INTERPRETER

At any point in the process of parsing, the control pointer for the interpreter points to a structure word in the grammar. A path stack also exists which shows the nodes from which TRUE returns have been achieved. Some operations mark the path stack for halting the backup process. The parser has 4 distinct control states defined as follows:

1) parsing: recognition state where input text is compared with gramatical constructs to determine the parsing path in the parse tree.

2) backup: A FALSE return has been obtained from some

execution/recognition function. The path stack is backed up until a non-NULL alternative path is found, at which time the parse mode is set to parsing, and recognition of the alternative path is attempted. If no non-NULL alternative path is found, then the parse fails and the interpreter returns FALSE.

3) cleanup: A terminal parse has been achieved and control is passed to each execution routine to reset any state informations set by the routine.

4) repeat: The command is being repeated, and each execution function is given control to redo the operation it last performed (if its function is defaulted by the semantic action of the command).

The general flow of control is:

1) An initial path stack entry is constructed, and the parse mode is set to parsing. The execution function for the current node is evaluated. A pointer to the "function state record" is passed to the routine. The state record contains the return values for the function as well as a record of any state information saved by the function (for backup purposes).

2) If the function returns TRUE, then the successor to the current node becomes the current node. If this is NULL, then the ptrstk stack is backed up until a non-NULL successor path is found. If none is located before the bottom of the current parse state is reached, then the root of a parse tree has been reached, and a command has been successfully executed. In this case the command reset operation is performed and the interpreter is set to "parsing" mode once more.

3) If the function returns FALSE then the parser mode is set to "backup" and a non-NULL alternative path is sought.

After a command has been executed, the parsing path for the tree is re-evaluated in "reverse order" beginning with the terminal node of the path. Each execution function is re-invoked, in "cleanup" mode, and is passed the handle for the state information record which it generated on the forward pass through the grammar. Each execution routine has the responsibility of resetting any state information which it wishes to do at the termination of a command. Cleanup continues until a "starting point" is reached in the parse. This is generally the beginning of the command. At this point, the interpreter "shifts gears" and goes into forward or recognition mode and begins back down the grammar for the language.

The same backup mechanism is also used during command specification in order to back up the parse to allow the respecification of all or part of the command. The command delete function backs out of the parse tree until the beginning of the command is reached.

The same backup mechanism may be adapted to control the partial backup required for executing commands in "repeat mode" where at least one of the alternatives are defaulted to their current values.

The process of marking some nodes in the execution path as defaulted is as yet undefined. It seems that it should be possible to identify those execution functions which need not be re-evaluated in subsequent invocations of the command. The interpreter would then be smart enough to skip over defaulted parameters when in the forward or specification phase of the command and would not invoke backup for defaulted parammeters.

APPENDIX 1: USING THE CML SYSTEM

WRITING CML PROGRAMS

Source programs for the CML compiler are free form NLS files. Comments may be used wherever a blank is permitted and the structural nesting of the source file is ignored by the compiler.

COMPILING CML PROGRAMS

CML source programs are compiled into REL files with the Output Compiler command using CML as the compiler name. The current marker (top of display area) should point to the first statement of a CML program, not the top of an NLS file.

RUNNING CML PROGRAMS

After loading the user program for the parser (<rel-nls>parser) and your rel file, you must connect your grammar to the parser. This is done by using NDDT to change the address field of the instruction at PARSER+1 to point to your grammar (whose address is contained in the symbol table entry corresponding to your subsystem name).

Example:

IF your subsystem name is "expjournal" then you could connect the parser to your grammar with the following NDDT command:

S*[how]* L*[ocation]* PARSER+1← MOVEI A1,EXPJOURNAL<CR>

After connecting your test grammar to the parser, parsing is initiated by the NLS command :

G*[o to]* P*[rograms]* E*[xecute program]* PARSER CA

FUNCTION INTERFACE PROTOCAL

The syntax of the function call in the CML meta-language is similar to that of most programming languuages: the name of the function is followed by a list of expressions enclosed in parenthesis. In the CML system however, there are some strict rules which apply to all execution functions invoked by the interpreter. These rules are enumerated below:

1) Additional actual arguments

Preceeding any actual arguments which appear in a function reference in CML, the interpreter supplies two additional actual arguments. These are:

1) a pointer to the "function state record"

2) an integer which defines a parsing mode

= parsing: normal execution mode

= backup: backup after a FALSE path is taken

= cleanup: resetting of state after completion of command

These addtional arguments must be used by all execution functions to determine what they are to do. The pointer to the "function state record" is used to return values from the function and to save state information associated with a particular invocation of the function. The length of the function state record is presently 9 words and this record may be formatted in any manner appropriate to the function.

If 9 words is not sufficient space to record all of the state associated with a particular invocation of a function, then the function must use a storage allocator to allocate the additional storage and record the handles to the allocated storage in the function state record. Note that if this additional "local state" storage is required, then it is the responsibility of the execution function to de-allocaate the local state storage when called in backup or cleanup modes.

2) Returning parse failure

All execution functions are passed a pointer to their function state record. If the function processes normally, then it returns the same pointer as its only return value. If the function decides that the parse should fail at a given point, then it returns FALSE.

3) Passing arguments by address

All of the actual arguments in a function call on an execution function are passed by address rather than by value. The values actually passed are pointers to the function state records corresponding to the actual arguments. The format of the function state records are defined by the execution functions which manipulated them, and thus the location of parameter values in these records is determined by convention, the caller and callee having previously agreed to a particular layout for the function state record. The layout of the records for the built-in interpreter functions in given elsewhere in this appendix.

4) Order of control

An execution function will always be called in parsing mode before it is called in backup or cleanup modes.

A function routine which saves state information in the function state record must initialize its state record to some consistent state before it calls any subroutines which may cause SIGNALS or otherwise cause control to abnormally pass above the execution funtion.

Format of the function state records for the built-in CML recognizers.

Each of the functions of the CML parser utilitzes the function state records in a locally defined way summarized below.

REGOGNIZER	RECORD FORMAT	# WORDS USED
keyword	word 1: address of keyword str	1
viewspecs	wordl: updated vs word 1	7
	word2: updated vs word 2	
	words 3-7: vs collection string	
levajd	wordl: level adjust count	7
	(u = +1, d = -1, etc)	
	words 2-7: vs collection string	
ssel	words 1-2: txt ptr to start of entity	4
	words 3-4: txt ptr to end of entity	
dsel	same as ssel	
lsel	same as ssel	
confirm	word 1: confirmation code	1

APPENDIX 2: SAMPLE CML PROGRAM

% the following sample program should help illustrate the use of the CML language for describing NLS commands. %

% the grammar is taken from observation of a hypothetical first grade class in the process of receiving art instruction %

% for a more exhauative example, take a look at (dornbush,syntax,) %

```
FILE sampleprogram   % CML to sample.rel %

   SUBSYSTEM sample
```

```
      objects =
           "GLUE"!L1!
         / "PASTE"!L1!
         / writingthings;
      writingthings =
          "CRAYONS"!L1!
         / "PENS"
         / "PENCILS";
      COMMAND zuse =
         "USE"!L1!  what ← writingthings
            <"to draw a pretty">
               ( whom ← "PICTURE"!L1! <"of Aunt Mary">
               / whom ← "SKETCH"!L1! <"of your dog"> )
            CONFIRM
            % call execution routine process the USE command
               *** commented out for now ***
               xuse( what, whom)
               ***    *** % ;

      COMMAND ztake =
         "TAKE"!L1! what ← objects
         <"out of your">
            where ← ("EARS"!L1! / "NOSE"!L1! / "MOUTH"!L1!)
         <"PLEASE!!"> CONFIRM;
   END.
FINISH
```

APPENDIX 3: SAMPLE INTERPRETER PARSEFUNCTION ROUTINE

Assume that in some command we want the typein of a number to appear as an alternative of some set of keywords. We can accomplish this

by defining a parsefunction (call it looknum) which looks at the next input character and succeeds if the next character is a digit and fails otherwise. If we write this function as the first alternative in some command, then control will pass from the interpreter to the parsefunction before it passes to the keyword interpreter.

Suppose our command looks like:

```
COMMAND sample =

   "INSERT"!L1!

      ( looknum() <"number">  ent ← #"NUMBER"

      / ( ent ← ( "TEXT"!L1! / "LINK"!L1!) ))

      % entity now contains an entity type ( number, text, or
      LINK ).  We now use the LSEL function to get a selection of
      this type %

         source ← LSEL( entity)

      % get a command confirmation %

         CONFIRM

      % now invoke the insert execution function passing as
      arguments the entity type and the selection of that type %

         xinsert( entity, source);
```

Now take a look at the parsefunction looknum which is called by the interpreter both when prompting the user and also during the actual parse of the command .

```
   % LOOK FOR A NUMBER %

      (looknum) PROC(

         % looknum looks at the next input character, if it is a
         digit, then a true return is taken else FALSE is returned %

         % FORMAL ARGUMENTS %

            resultptr,       % ptr to the function state record %

            parsemode,       % parsing mode for the interpreter %

            string);         % ptr to prompting string %

         REF resultptr, string;

         %------------%

         CASE parsemode OF
```

```
    = parsing:
        CASE lookc() OF
            IN ['0, '9]:
                NULL;
        ENDCASE RETURN (FALSE);
    = parsehelp:
        *string* ← "NUM:";
ENDCASE;
RETURN (&resultptr);
END.
```