SLAC-PUB-844
November 1970

A COMMAND LANGUAGE META-SYSTEM†

James E. George*
Computer Science Dept., Stanford University
and
Harry J. Saal**
Stanford Linear Accelerator Center

Abstract

A meta-system for the construction of scanners for a wide variety of command-type languages has been developed. Provided with suitable sets of tables generated from command descriptions, it is capable of parsing command languages which enforce strict criteria on parameter specification or those with free field, keyword identifier construction. The description language permits recursive scanning so that list-like parameter fields may be recognized in addition to simple ones. All phases of the system construction are table driven and provide great ease and flexibility for experimentation.

1. INTRODUCTION

Our initial aim was to design an interactive text editor for a micro-programmable computer and to simulate this text editor on an IBM 360/91. In the process of trying to design the user interface and an adequate set of primitive functions, we reviewed several text editors and discovered that, as a group, we could not agree to a specific user interface; the conflict was on commands, command names and the specification of parameters to a command. We did agree on the general system objectives of:

(1) User interface should be easy to change.

(2) In a time sharing environment, each user should not be required to have his own copy of a text editor.

We realized that we were really designing a meta-system for command languages.

2. OVERALL VIEW OF THE META-SYSTEM

Figure 1 illustrates the meta-system. To define a new command language, a command description is analyzed by the table generator which generates a table suitable for use by the scanner. The inclusion of the desired semantics as subroutines in the primitive library completes the definition of the command language.

To use a specific command language, the user designates to the scanner which table is to be used. This table is then obtained and saved in the user's space. Commands can now be analyzed by the scanner using the specified table and semantics performed through activation of subroutines in the library.

This model provides the versatility desired and allows command languages to be developed or modified modularly. New or modified commands can be tested without the other users of that particular command language system being aware or affected by this testing. Further, each command language can be tailored to a user or group of users. This tailoring could provide simplified commands for less sophisticated users or could limit their actions or capabilities. These capabilities could involve items wuch as read only systems, file access restrictions, etc.

3. THE TABLE GENERATOR

The table generator is implemented in PL/I using a simple precedence syntax analyzer and semantic constructor. The syntax for command descriptions (which is an input to the table generator) is given in Figure 2.

*Supported by NSF Contract NSF-6J-687

**Supported by AEC Contract AT(043)-515

†Summary of talk to be presented at Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 12-14, 1971.

A command table consists of a set of options followed by a list of commands. The options consist of the table name to which the table generator adds the current date and time for identification, a separator for use in the tables (*PERIOD*) and a character which will surround strings to indicate type <STRING> (*QUOTES*).

The list of commands is composed of subroutines used in the commands and the keywords denoting the commands and their parameters. Commands are indicated by *KEYWORD*: *RTN* indicates the subroutine to be activated when the parameters are obtained. Both commands and subroutines (*SUBENTRY*) can have alternate names.

Normally, all special characters are treated as delimiters (all non-alphanumeric characters) by the scanner; when scanning for the next item, the scanning proceeds until a delimiter is found and then the delimiter is deleted. If a delimiter is given in *DL-EX-LIST* for a command or subroutine, it is not deleted but is returned as the following item; if the delimiter is given in *DL-SKIP*, it is treated as any other alphanumeric character.

Parameter types may be number, name, string or a table-subroutine call (<STRING>), and may be initialized. Parameters may further be restricted by using the *P* and *K* options; for *P*, no parameter before the one with this option can be filled in after this parameter; for *K*, this parameter can only be filled in after recognition of its key.

A parameter may have multiple keys of varying types. Type *VALUE* means take the next item after the key and assign it to the parameter; type *VALUE* <STRING> means take everything up to the occurrence of <STRING> and assign it to the parameter and then delete <STRING>; type *VALUESHORT* <STRING> is *VALUE* STRING but without deletion; type *SELF* <STRING> means assign <STRING> to the parameter; type *CALL* <STRING> means call table subroutine <STRING>.

4. THE SCANNER

For ease of implementation, the present scanner was written in SNOBOL4. One assembly language module was provided in order to permit SNOBOL4 to interact with our local terminal supervisor, MILTEN.

A user specified syntax table is scanned and translated into a SNOBOL 4 data structure. Commands are then entered from the terminal.

The routines SCANNER and ISTYPE (used recursively) do the bulk of the work in interpreting a user command. SCANNER first locates the command keyword and if valid, establishes a new set of delimiters based on *DL-EX-LIST* and *DL-SKIP* fields.

Using these delimiters, the next field is extracted. If it is not a keyword for the current parameter and this parameter requires a key, one looks at the next possible parameter entry. If the keyword is not required, ISTYPE is called to determine if this field (or more) is of the type specified for the current parameter. If any of the field is recognized, it is used as the parameter value; otherwise, we advance to the next parameter to see if it is a keyword or recognizable type.

Whenever a keyword is located, the action taken depends on the key-type specified. We may search for a matching delimiter (like closing quotes) where the delimiter is to be ignored once detected, or a separator character (like comma) which must be rescanned later as a keyword identifier. We may simply set a parameter value upon detecting a keyword without further scanning (like recognizing 'NOLIST') or else recursively call ISTYPE with a specified command word string preceding the remainder of the user command. This permits new data types to be recognized by using subroutines of table entries in addition to the built-in classes *NUM* (number), *NAME* (alpha followed by alphanumerics) or *STRING* (arbitrary). ISTYPE handles these three classes directly but will call SCANNER to use its table for anything else.

Whenever parameters are successfully recognized, their values are built into a tree structure which reflects the level of scanning at which a value was ascertained. This tree is then passed to the semantic routine associated with the top level command entry decoded. The semantic routines may determine if a value was explicitly determined by scanning or was an initialization value.

The scanner described most closely resembles a top-down parser. However, it will try alternatives only if no additional part of a command is recognized. In general, there is almost no backing up in the scanning and inability to read further or errors in type will exit with a pointer to the offending character. This provides a very good localization of the cause of non-recognition.

5. EXAMPLE

Let the desired command be

LIST <NUM> | / <NUM> | IN <FILENAME>

where L may be substituted for LIST. Further assume that <FILENAME> can not be specified first and that <NUM> can be used in place of <NUM> /<NUM>. The command description is:

```
*QUOTES* *=* "
*PERIOD* *=* .
*TBL-NAME* *=* "EXAMPLE"
*KEYWORD* LIST *RTN* SUBROUTINE1 *DL-EX-LIST* "/"
*KEYWORD* L *RTN* SUBROUTINE1 *DL-EX-LIST* "/"
*PARM* *NUM* *INITIAL* "-1" *END*
*PARM* *NUM* *K* *P* *INITIAL* "-1"
   *KEY* / *VALUE* *END*
*PARM* *NAME* *K* *P* *INITIAL* ""
   *KEY* IN *VALUE* *END*
*END-TABLE*
```

The output table produced would be:

```
EXAMPLE          10/30/70        14:33:48.250

LIST.SUBROUTINE1./..

L.SUBROUTINE1./..

.*NUM***.-1.
.*NUM*P*K*.-1.
../.*VALUE*..
.*NAME*P*K*..
..IN.*VALUE*..
```

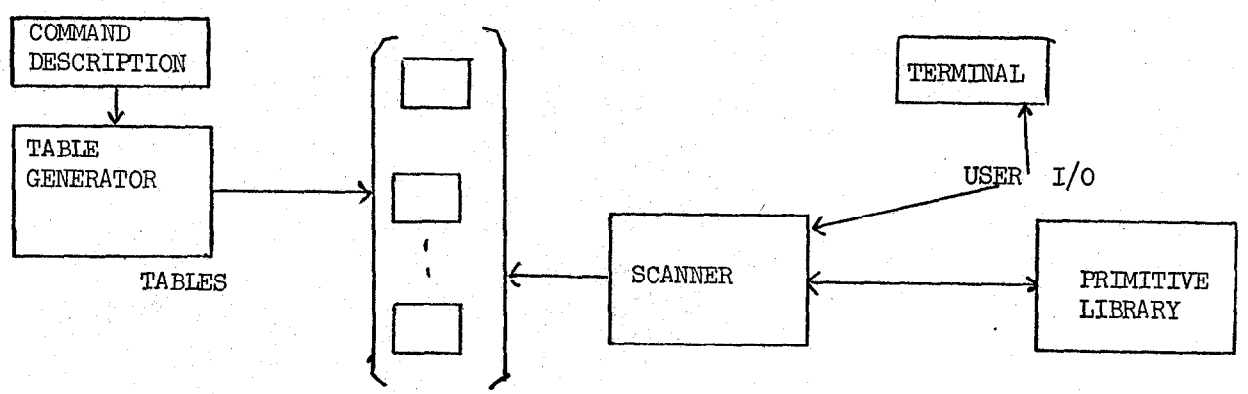

FIGURE 1

Command Language Meta-System

```
⟨COMMAND TABLE⟩   ::=  ⟨OPTIONS⟩ ⟨COMMAND LIST⟩ *END-TABLE*
⟨OPTIONS⟩         ::=  ⟨OPTION⟩|⟨OPTIONS⟩ ⟨OPTION⟩
⟨OPTION⟩          ::=  *QUOTES* *=* ⟨WORD⟩ | *PERIOD* *=* ⟨WORD⟩ |
                       *TBL-NAME* *=* ⟨WORD⟩
⟨COMMAND-LIST⟩    ::=  ⟨ID LIST⟩ ⟨PARM LIST⟩ | ⟨COMMAND-LIST⟩ ⟨ID LIST⟩ ⟨PARM LIST⟩
⟨ID LIST⟩         ::=  ⟨ID SPEC⟩ | ⟨ID LIST⟩ ⟨ID SPEC⟩
⟨ID SPEC⟩         ::=  ⟨ID⟩ { *DL-EX-LIST* ⟨STRING⟩ } { *DL-SKIP* ⟨STRING⟩ }
⟨ID⟩              ::=  *KEYWORD* ⟨WORD⟩ *RTN* ⟨WORD⟩ | *SUB-ENTRY* ⟨WORD⟩
⟨PARM LIST⟩       ::=  ⟨PARM⟩ *END* | ⟨PARM LIST⟩ ⟨PARM⟩ *END*
⟨PARM⟩            ::=  ⟨PARM ID⟩ | ⟨PARM ID⟩ ⟨KEYS⟩
⟨PARM ID⟩         ::=  *PARM* ⟨TYPE⟩ | *PARM* ⟨TYPE⟩ *INITIAL* ⟨STRING⟩
⟨TYPE⟩            ::=  ⟨V TYPE⟩ { *P* } { *K* }
⟨V TYPE⟩          ::=  *NUM* | *STRING* | *NAME* | ⟨STRING⟩
⟨KEYS⟩            ::=  *KEY* ⟨WORD⟩ ⟨TYPE KEY⟩
                  ::=  ⟨KEYS⟩ *KEY* ⟨WORD⟩ ⟨TYPE KEY⟩
⟨TYPE KEY⟩        ::=  *VALUE* | *VALUE* ⟨STRING⟩ | *SELF* ⟨STRING⟩ |
                       *VALUESHORT* ⟨STRING⟩ | *CALL* ⟨STRING⟩
```

FIGURE 2

Syntax for Command Descriptions

{} means can occur 0 or 1 time
{}{} followed by {} means each can occur 0 or 1 time in any order