NASA Contractor Report 172560

ICASE REPORT NO. 85-16

ICASE

CONSTRUCTION OF A MENU-BASED SYSTEM

Robert E. Noonan

W. Robert Collins

Contract No. NAS1-17070

February 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

CONSTRUCTION OF A MENU-BASED SYSTEM

Robert E. Noonan [1]

College of William and Mary
and
Institute for Computer Applications in Science and Engineering

W. Robert Collins [2]

College of William and Mary

SUMMARY

In this paper we discuss the development of the user interface to a software code management system. The user interface was specified using a grammar and implemented using an LR parser generator. This was found to be an effective method for the rapid prototyping of a menu-based system.

[1] Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-17070 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

[2] Research was supported by NASA Langley Research Center under NASA Grant NAG-1-534.

INTRODUCTION

As interactive computing has displaced batch processing, the design of the user interface has become increasingly important. Unfortunately, in most systems the user interface is not explicitly designed, resulting in confusing and unfriendly systems. Command languages, such as UNIX (Tm) and IBM's TSO, are notorious examples of unfriendly systems.

Recently, the authors were part of a team to design and implement a software code management system, named SCMS, to run on a UNIX-based workstation. Previous experience with other software tools had convinced management that an unfriendly user interface could ruin an otherwise satisfactory system. The authors were assigned the task of designing and implementing the command language for this system, based on a set of requirements and a list of sample commands.

The user interface was to be implemented as rapidly as possible, long before the rest of the system was completed. This was to provide the capability of being able to "play" with the user interface as early as possible and to request changes in the interface. These changes were to impact the ongoing implementation as little as possible.

Based on experience with the design, implementation, and use of other interactive systems, it was determined that the user interface should be specified and implemented using a BNF grammar. Since the terminals to be used were ordinary alphanumeric terminals (of varying

kinds) using an RS-232 port, a mouse-based command system was not possible. The alternatives considered were a conventional command language and a menu-based system. While conventional command languages are commonly implemented based on a grammar, menu systems seldom are. However, the inherent friendliness and ease of use of menu systems made them the logical choice.

In the remainder of the paper the guidelines used for constructing the grammar are discussed. Next, we discuss the implementation of the system using an LR parser. We do not, however, discuss the functionality of the system; readers interested in this aspect should consult [Rochkind 1975]. Finally, the advantages of this approach are enumerated.

DESIGN OF A MENU SYSTEM

Based on experience with using a variety of menu systems, it was decided that menus would be presented on a single line, with the first character of each menu item being used to select that item. Thus, a typical menu might appear as:

```
SCMS: C(reate-lib D(estroy-lib U(se-lib Q(uit ->
```

In this case, "SCMS" is the name of the menu being displayed, and valid responses consist of "C", "D", "U", and "Q" (or their lower case equivalents).

Other menu styles are possible. This style was chosen because it was felt that brief menus would not interfere with use of the system by experts (all users are professional programmers). Also, some users may be using low bandwidth ports to the computer. However, the style chosen has no real impact on the implementation.

The movement from one menu to another was perceived as movement within a transition diagram. This leads to the use of right recursion in the specifying grammar. Thus, the grammar rules for the above menu might appear as:

```
<scms_menu> ::= c <create_lib> <scms_menu>
<scms_menu> ::= d <destroy_lib> <scms_menu>
<scms_menu> ::= u <use_lib> <use_menu>
<scms_menu> ::= q
```

Both the create and destroy commands leave the user in the SCMS menu, while the use command gets the library name and then invokes another menu. The quit command exits the system.

Four alternatives were considered for implementing the grammar. The first alternative was to handcraft a program that implemented the grammar, following the technique known as recursive descent [Aho 1977]. However, extensive changes in the menus due to experimentation would cause the program to define the interface, and not the grammar as desired. Also, this alternative violated our desire to quickly produce a running prototype.

Another alternative was to implement the grammar as a finite state machine. However, this would capture only those parts of the grammar dealing with menus. Some nonterminals occurred in the right-hand side of more than one rule. Also, we had no automated tools available for converting a grammar or part of a grammar to a running implementation.

The third alternative was to use an LL parser generator and an LL parser [Aho 1977]. There were two problems with this choice. As written the grammar was not LL(1), although this was a relatively minor problem. The major problem was that we did not have an LL parser generator available to us.

The last alternative was to use an LR parser generator and an LR parser [Aho 1977]. We had such a parser generator available to us [Collins 1980] and had extensive experience with it. More importantly, the source code of the parser generator was both available and familiar to us, a consideration that, in retrospect, was irrelevant. The grammar was SLR(1), a subset of LR. The only major problem was that the extensive use of right recursion in the grammar necessitated a potentially infinite runtime parse stack.

Since a running implementation could be produced directly from the grammar very quickly, it was decided to use an LR parser. The parser stack problem was temporarily finessed by declaring the stack to be very large (a thousand entries). In fact, the ease of implementation allowed the exploration of issues using a subset of the grammar and the experimentation with stylistic issues such as the appearance of menus,

etc,

IMPLEMENTATION

Over the course of several years, we have developed a number of distinct parser programs targeted at various application areas. One of these, named QUERY, has been used for developing traditional, interactive applications. This program was used as a starting point in developing the parser needed for this application.

QUERY has a traditional parser-based program structure, consisting of three phases [Aho 1977]: scanner, parser, and semantics phase. The first two depend only on the grammar and are otherwise application independent. The semantics phase consists of a procedure containing a giant case statement, with one case per grammar rule. As the end of a rule is recognized, the parser invokes this routine, passing as an argument the number of the grammar rule. Hence, all of the application dependent code is placed in the semantics routine and the procedures it invokes.

There are several problems with this structure. All of the input is read in the scanner and passed in a logical stream to the parser. As soon as the parser has consumed a token or symbol, the scanner is called to produce a new token. In particular, the scanner is required to produce a token before the parser begins executing!

A second problem is the coordination of the prompt menus with the input. The former would have to be output by the semantics routine, while the latter is handled by the scanner. This results from the fact that the scanner has no knowledge of the state of the parser. Coordinating the two was clearly a problem.

The third problem had to do with incorrect input. In this case, the user has to be notified and a new input obtained. With input in one routine and output in another, coordination was again going to be a problem.

One possible solution was to eliminate the scanner entirely and have all the output of menus, input, and legality checking done in the semantics routine. The major problem with this solution is that the code to perform this function had to be replicated in as many places as there were menus. Also, this solution decreased the importance of the grammar in specifying the user interface.

In a very real sense, the coordination of parser input with the output of menus is similar to the functions performed when a syntax error is discovered in the input [Graham 1975]. The error recovery routine first determines what tokens are legal given the current state of the parser, so that a legal token can be either inserted ahead of or exchanged for the illegal token. This notion was combined with the concept of lazy input [Kaye 1980], in which input is not requested from a terminal until the program requests it via a "read" statement. However, for this scheme to be effective the parser must use "default"

reductions [Anderson 1973] in those states in which the input is irrelevant. Fortunately, this is a space saving technique commonly employed by parser generators.

In our implementation, there is no scanner. No input is done until the parser enters a state in which a token is required. At this point, the "error recovery" routine enumerates on the terminal the set of legal tokens (menu choices). A character is read from the terminal and checked against the first character of each token. If a match occurs, the "error recovery" routine returns with the "correct" token. Otherwise, a bell character is output and the user reprompted for input.

This approach has a number of advantages. First, the menus themselves appear explicitly in the grammar:

```
<use_menu> ::= C(heck <check_menu>
<use_menu> ::= D(efine <define_menu>
<use_menu> ::= R(emove <remove_menu>
<use_menu> ::= E(xit <scms_menu>
```

Thus, addition of new menu choices or even of entire new menus is accomplished by modifying the grammar and regenerating the parser. The latter process takes only a few wall clock minutes on our supermini computer.

Having menu presentation and legality checking of input done in one

place has a number of advantages. First, there are enormous savings in the amount of code that must be written. Stylistic changes in the presentation of menus is easily accomplished. Finally, the code is fixed and uses automatically generated tables.

The only remaining problem was the potentially infinite parser stack. However, because of the design of the grammar, menu states only appear on the stack more than once through the use of right recursion. Since this use is merely to mimic transitions from one menu to another, the duplication of a menu state on the parser stack is unnecessary. Through the use of a simple and efficient mechanism, the stack is cut back to the previous occurrence whenever such a duplication occurs. Thus, the parser stack cannot grow to be larger than the number of parser states.

Table 1 gives some statistics on the size of the specifying grammar.

	SCMS	Pascal
Grammar Rules	114	202
Nonterminals	72	92
Terminal Symbols	33	64
Parser States	86	194
Menus	12	--

Table 1 : SCMS vs. Pascal Grammars

As can be seen from the comparison with Pascal, the language implemented was nontrivial. The design of the menu items themselves took considerably longer than the implementation.

One of the unexpected benefits of this approach was that it uncovered problems in the requirements document for the system. These discrepancies were easily fixed because they were discovered early enough not to impact the implementation of the functionality of the system.

CONCLUSIONS

Using a grammar to formally specify the user interface to an application usually results in a cleaner, more consistent interface. An added benefit is that a parser generator can be used to generate the user command language directly from the grammar. Thus, a running prototype can be generated very quickly.

We have found that even menu-based applications benefit from this approach. The problems encountered in using an LR parser were easily solved. The resulting prototype enabled early experimentation with the system and uncovered discrepancies in the requirements document.

NOTES

UNIX is a trademark of AT&T Bell Laboratories.

REFERENCES

1. Anderson, T., Eve, J., and Horning, J. J. Efficient LR(1) parsers. Acta Informatica, 2 (1973), 12-39.

2. Aho, Alfred V., and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley, 1977.

3. Collins, W. Robert, Knight, John C., and Noonan, Robert E. A translator writing system for micro-computer high-level languages and assemblers. NASA-AIAA Workshop on Aerospace Applications of Microcomputers, (November 1980), 179-186.

4. Graham, Susan L., and Rhodes, S. P. Practical syntactic error recovery. CACM, 18 (November 1975), 639-650.

5. Kaye, Douglas R. Interactive Pascal input. SIGPLAN Notices, 15 (January 1980), 66-68.

6. Rochkind, Marc J. The Source Code Control System. IEEE Trans. on Soft. Engr., SE-1, (December 1975), 364-370.

1. Report No. NASA CR-172560 ICASE Report No. 85-16	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle CONSTRUCTION OF A MENU-BASED SYSTEM		5. Report Date February 1985 6. Performing Organization Code
7. Author(s) Robert E. Noonan and W. Robert Collins		8. Performing Organization Report No. 85-16 10. Work Unit No.
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		11. Contract or Grant No. NAS1-17070, NAG-1-534 13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South, Jr.
Final Report
Submitted to Software Practice & Experience.

16. Abstract

In this paper we discuss the development of the user interface to a software code management system. The user interface was specified using a grammar and implemented using a LR parser generator. This was found to be an effective method for the rapid prototyping of a menu-based system.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
grammars parser generator menu-based system	61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	12	A02