

K*: A FORTRAN-BASED CODE FOR PROGRAMMING AND EVALUATING INTERACTIVE SOFTWARE

D. F. Redmiles
Mathematical Analysis Division
National Bureau of Standards
Washington, D. C.

ABSTRACT

K* (read kay-star) is an experimental library of FORTRAN 77 subroutines that simplifies the design and implementation of command-driven interactive programs. K* gives program designers a way to evaluate and modify easily the user interface. Also, it relieves programmers of coding many tasks associated with user interaction.

The K* library and the programming methodology it uses are designed with "mid" to "large" sized applications in mind. This paper first describes K* and under which circumstances K* is most useful; second, it presents an example implementation of an interactive program.

KEYWORDS

Command interpreter; data structure; formal language; FORTRAN 77; human interface; LISP; operating system; relational database; software engineering; structured programming.

INTRODUCTION

A common form of interactive computer program is the command interpreter. A command interpreter is a program that prompts a user for a command, carries out the given command, and returns to the original prompting mode awaiting further instructions.

The command interpreter style occurs in programs across many disciplines and at all levels of computer programming; e.g., CAD/CAM programs, operating systems, and text editors. Builders of interactive programs face a double problem: they must both design a good user interface and implement an error-free code.

The way command interpreters interact with their human users has long been the focus of research (1). Within the past few years, the theory of formal languages has been used to develop grammars solely for implementing and evaluating the interactions between a user and a command-driven program (2).

Such research is described in the April 1983 issue

of the Communications of the ACM (3). That issue is devoted entirely to the human-computer interface.

The problem of implementing good, error-free code has received even more attention (4). Economic (5), social (6), and mathematical (7) considerations have each been applied to the problem of cost-effectively delivering working software.

K* combines the ideas of modern programming practices and of formally specifying the user interface. Unlike the grammars used in the research cited above, the specification language used by K* is designed to serve easily several purposes discussed later.

Moreover, the K* subroutines help with code implementation by providing FORTRAN subroutines that perform most of the functions needed in the human interface and by encouraging a modular code design.

DESCRIPTION

COMPONENTS

K* (K: command interpreter; *: generalized) is a package of FORTRAN 77 subroutines that support a general command interpreter. For a particular interactive program, the programmer supplies K* with a "description" file of the user interface and with application dependent subroutines. The description file gives K* a map of the available commands and other user interactions. When it executes, K* uses the description file to interpret user commands. K* calls on the application subroutines to execute the commands.

The description file is specified in a language called LEMM (Linear Encoding for Multi-Media), originally developed for representing scientific data (8) at the suggestion of Don J. Orser. Loosely, LEMM is a list encoding of a relational database record (9); its form is designed to be invariant over various computer storage media; i.e., disk, tape, printout.

The LEMM description file contains the names of commands, their arguments, prompt messages, and help messages. Some commands have subcommands with additional arguments and messages. Each command has an

associated sequence number that K* uses at run-time to find and execute user-supplied subroutines that perform the specific command.

Thus for each command, the applications programmer is expected to supply one or more subroutines. He must also provide one master subroutine to coordinate all calls.

Using the LEMM description file, K* prompts the user for a command, verifies its existence, collects its arguments on a stack, and passes the command's sequence number and stack of arguments to the master call subroutine. The master call subroutine then passes the arguments to the appropriate command subroutine.

USAGE AND BENEFITS

By design, K* is intended for implementing command interpreter programs. Hence, K* is not currently appropriate for all styles of interactive programs, just command-driven ones.

Using the K* package seems at first to incur unnecessary trouble. This point is conceded in the case of a small, isolated program. However, for large problems, the benefits of using K* begin to outweigh its overhead.

Note the use of the word "problem" as opposed to "program". The solution to one problem may call for several small programs. K* facilitates the planning and maintenance of a set of programs by imposing a uniform implementation.

Separately specifying the user interactions in the LEMM description file emphasizes the importance of the user interface. The LEMM description file serves other purposes as well.

Since the LEMM description file removes the various user interactions from a program's code, changes to the command names, prompts, and help messages may be made without recompiling the code. Such modifications can be made using an ordinary text editor; no programming effort is required.

K* has a facility for allowing program designers to walk-through the program interactions before any supporting code is written. Thus program designers can evaluate and refine a design before its implementation.

When coding begins, not all commands have to be implemented at once. If commands and subcommands are thought of as program modules (10), the LEMM description file may be viewed as a map of the programming task. The hierarchy of program modules is recorded in this map in the command/subcommand hierarchy. Programmers may implement modules as appropriate. The modular approach lessens the problems of program maintenance and debugging.

Since K* knows which applications subroutines to call for which commands and subcommands, the run-time relation of modules is "flattened". Specifically, at execution time, no command's subroutine depends on another command's subroutine being in computer memory at the same time. Thus planning of memory linkages or even memory map overlays is simplified.

Internal to the K* code are many subroutines for manipulating list structures (11) that K* calls to read the LEMM encoding and to use it. The basic list subroutines are patterned after implementations of the LISP programming language (12, 13). The LEMM subroutines use ideas from Minsky's frames (14, 15). K* is built to allow a programmer to call these list subroutines directly from his application code. Thus K* becomes especially attractive if an application can benefit from using data structures more flexible than FORTRAN scalars and arrays.

Finally, the K* subroutines which control the interaction with the user may be enhanced to handle more elaborate styles of interaction. Such modifications could be made without having to disturb an application's code.

APPLICATION

So far, the reader has learned about the components, usage, and benefits of K*. Seven steps for applying K* to a problem are now presented.

These steps may be thought of as the K* programming methodology.

1. Specify the problem task.
2. If an interactive command interpreter is appropriate, decide on a selection of commands.
3. Write out command names, prompts, and help messages in a LEMM description file.
4. Use K*'s "dummy" program to walk-through the proposed interactions.
5. Revise the commands and repeat step 4 until the command selection and options seem satisfactory.
6. Draft a user's guide for the proposed system.
7. Add FORTRAN code segments as desired, debugging each command function as added.

EXAMPLE

PROBLEM STATEMENT

It was mentioned in the introduction that a computer's operating system exemplifies the command interpreter programming style. To demonstrate K*, the implementation of a simplified example operating system will be presented. The example operating system should provide commands for editing, printing, and enumerating files. The editor should be capable of inserting, printing, and deleting lines.

The programming will be simplified by permitting files to be limited in size and kept in memory.

COMMAND SET

The requirements of the example problem can be satisfied with operating system commands "ed", "print", and "dir" for editing, printing, and enumerating (getting a directory of) files respectively. The editor commands "input", "print", and "delete" will be for inserting, printing, and deleting lines respectively.

There should also be clean ways of terminating the operating system and editor. Hence the commands "shutdown" to stop the operating system and "exit" for leaving the editor are added.

The arguments for the operating system commands are obvious: "ed" and "print" need file names; "dir" and "shutdown" need nothing.

However, in the editor, "input", "print", and "delete" need to know where to input, what lines to print, and what lines to delete. By adding the concept of a current line pointer, any confusion can be eliminated: "input" can be assumed to follow the current line; "delete" and "print" can work with the number of lines following the current line.

LEMM DESCRIPTION FILE

The above selection of commands can now be

```

(eos
  (name: eos
   sequence: 100
   parameter:
     (prompt-message: "Who are you"
      help-message: "Give your user name, &
                    e.g. Bilbo.")
   commands:
     (prompt-message: "yes"
      help-message: "Enter an operating &
                    system command; type &
                    /"HELP/" for the &
                    selection."
     choice[1]:
       (name: dir
        sequence: 101)
     choice[2]: line-editor
     choice[3]:
       (name: print
        sequence: 102
        parameter:
          (prompt-message: "File"
           help-message: "Enter the name of &
                         the file you want to print."))
     choice[4]:
       (name: shutdown
        sequence: 103)))
)
(line-editor
  (name: ed
   sequence: 200
   parameter:
     (prompt-message: "File"
      help-message: "Give the name of the &
                    file you want to edit.")
   commands:
     (prompt-message: "do"
      help-message: "Enter an editor command;&
                    / type help for a list &
                    of available commands."
     choice[1]:
       (name: input
        sequence: 201)
     choice[2]:
       (name: go
        sequence: 202
        parameter:
          (prompt-message: "where"
           help-message: "Give the line &
                         number where you &
                         want to move."))
     choice[3]:
       (name: print
        sequence: 203
        parameter:
          (prompt-message: "how many"
           help-message: "Give the number of &
                         lines to be printed."))
     choice[4]:
       (name: delete
        sequence: 204
        parameter:
          (prompt-message: "how many"
           help-message: "Give the number of &
                         lines to be deleted."))
     choice[5]:
       (name: exit
        sequence: 205)))
)

```

Figure 1: LEMM description file for the example operating system.

specified in a LEMM description file. Command prompts and help messages will also be included along with sequence numbers that let K* associate command names with subroutines. The LEMM description file for the example operating system is listed in Figure 1. This listing shows two LEMM sentences. The first defines the object "eos" which stands for "example operating system". The second defines the object "line-editor".

Parentheses group items into lists. Sublists (lists within lists) provide greater levels of detail.

Words immediately followed by colons are attributes, the properties by which an object is defined. The remaining words and literals serve as values to these attributes.

The reader should now have a rough understanding of the way LEMM sentences define commands.

The word "line-editor" in the "eos" object could be replaced by the actual line-editor sentence. Separating the two objects demonstrates part of LEMM's flexibility, allows the option of reusing easily the line editor subcommand in another application, and emphasizes K*'s use of an identical representation for subcommands and main programs.

This self-similar nature of K* allows it to execute subcommands to any depth. Corresponding LEMM sentences and FORTRAN modules for the commands may be prepared without undue consideration of the level at which the program may call them.

WALK-THROUGH

After the LEMM description file is entered into the computer, a program designer can "walk-through" the proposed commands to evaluate the design from the user's point of view.

```

PROGRAM EOS
  • Implicit declarations
    INCLUDE '[-.macfor]implicit.inc'
  • Master call subroutine name
    EXTERNAL Kfprog
  • Initialize K* package
    CALL Macini()
    CALL Kstini()
  • Read in the LEMM description file
    PRINT *, 'Tell me the file name of the'
    PRINT *, 'LEMM encoding of your'
    PRINT *, 'operating system.'
    READ *, cfile
  •
    OPEN (UNIT=8,FILE=cfile,STATUS='OLD')
    ilastu = Mnunt(8)
    nos = Lread(icond)
    yeffct = Lread(icond)
    ilastu = Mnunt(ilastu)
    CLOSE(ilastu)
  • activate the K* generalized command
  • interpreter, supplying it with a LEMM
  • representation of the interactions and
  • the name of a master call subroutine.
    yeffct = Kstar(nos,Kfprog)
  •
    END

```

Figure 2: main program for walking-through the eos commands.

A short program, listed in Figure 2, passes the LEMM eos object to K* with the name of a master call subroutine. K* uses the master call routine to invoke the functions corresponding to the user's request.

For a walk-through, K* supplies a dummy master call routine which satisfies the FORTRAN compiler but

executes no commands (see Figure 3). With almost no programming, a designer can evaluate messages and test the adequacy of the proposed command set.

```

      INTEGER FUNCTION Kfprog(iseq,pargsk,
+                               iccode)
*
* supposed to call the subroutines that
* actually implement the commands. This
* version of Kfprog simply returns. Its
* sole use is for walking through the
* interactions outlined in the LEMM
* representation of the command
* interpreter.
*
      INCLUDE '[-.macfor]implicit.inc'
      INCLUDE '[-.macfor]macats.inc'
      INCLUDE '[-.macfor]maccds.inc'
*
* iccode   standard condition code
* iseq     sequence number specifying a
*          command
* pargsk   stack of arguments for a command
*
*
      iccode = IOKCC
      pargsk = NIL
      Kfprog = NIL
      END

```

Figure 3: K*'s dummy master call routine.

A walk-through of the example LEMM file is shown in Figure 4. Notice some features of the K* interface: "help" is built-in, commands may be entered without format restraints, some error checking is built-in, commands and arguments may be typed on the same line, no argument is requested unless it is required and has not been supplied.

REVISION

A revision made after the original walk-through was the renaming of the editor's print command. Originally, this command was called "type" but during the walk-through, it became apparent "print" was more consistent in view of the operating system print command.

USER'S GUIDE

A user's guide could have been drafted after deciding on the command set. By waiting until after the walk-through, a program designer can take advantage of any ideas gained from becoming momentarily a user.

The user's guide for the example operating system is Appendix A.

This guide was the principal specification followed for programming the example problem. For large problems, various design strategies are available (16, 17).

FORTRAN

The example operating system has been implemented by taking advantage of the subroutines for manipulating list structures.

Selected sections of the code are listed in Appendix B. The "include" statement is the only non-standard FORTRAN 77 statement: it inserts text from another file.

The first code segment defines the global variables that maintain the state of the operating system

and of the editor. Their list representation is suggested in the included diagram.

```

$ run walk
Tell me the file name of the
LEMM encoding of your
operating system.
'lplog'
Who are you?
Joe

yes?
           {user enters a blank line}
Enter an operating system command; type "HELP"
for the selection.
HELP
commands  prompt-message  help-message
dir
ed         File           Give the name of
                    the file you want
                    to edit.
print      File           Give the name of
                    the file you want
                    to print.

shutdown

yes?
  dir

yes?
  dir

yes?
djkfkj
eos doesn't recognize djfkj

yes?
ed file-1

do?
input

do?
go top print all

do?
print
how many?
5

do?
...

```

Figure 4: dummy walk-through

The main program passes K* the LEMM file of Figure 1 and the master call routine, "Iosprg".

When K* identifies the command requested by the user and collects its arguments into a stack, it calls Iosprg.

Iosprg splits into two subroutines, "Ioscom" for calling operating system commands and "Jedcom" for calling editor commands. These routines invoke a particular command subroutine as indicated by the sequence number.

Finally, two command subroutines are listed: "Ishut" for the operating system command, "shutdown"; and "Igo" for the editor command, "go". Ishut illustrates the use of the condition code for signaling K* to finish a command level, in this case, the main program. Igo illustrates heavy use of the list subroutines and the global variables.

An execution of the complete system is listed in Appendix C. Note how closely the walk-through of Figure 4 simulates the actual session.

SUMMARY

K* is a library of FORTRAN 77 subroutines for programming and evaluating command-driven, interactive programs.

The K* subroutines support programming by automating the user interface and supplying routines for manipulating list structures. In addition, K* prescribes a programming methodology based on modern programming practices.

K* requires separation of the user interactions from the code. This separation encourages careful consideration and adjustment of the user interface, especially with the aid of walk-throughs before coding begins. Also, the interactions file enhances the programming specifications.

The K* user interface subroutines may be modified without disturbing an application.

K* allows applications programmers to implement command-driven programs in a uniform fashion, without the burden and hazard of writing code for the user communication.

ACKNOWLEDGEMENTS

I thank Anthony P. Datel for his help in designing and programming K*, and Martin Knapp-Cordes for his technical assistance.

REFERENCES

1. Foley, J. D. and V. L. Wallace, "The Art of Natural Graphic Man-Machine Conversation", Proceedings of the IEEE, vol. 62, no. 4, April 1974, pp. 462 - 471.
2. Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems", International Journal of Man-Machine Studies, vol. 15, no. 1, July 1981, pp. 3 - 50.
3. Jacob, Robert J. K., "Using Formal Specifications in the Design of a Human-Computer Interface", Communications of the ACM, vol. 26, no. 4, April 1983, pp. 259 - 264.
4. Boehm, B. W., "Seven Basic Principles of Software Engineering", The Journal of Systems and Software, vol. 3, no. 1, March 1983, pp. 3 - 24.
5. Boehm, B. W., "Software Engineering Economics", IEEE Transactions on Software Engineering, vol. SE-10, no. 1, January 1984, pp. 4 - 21.
6. Baker, F. T., "Chief programmer team management of production programming", reprinted in Tutorial on Structured Programming: Integrated Practices, edited by Victor R. Basili and F. Terry Baker, IEEE Computer Society, Los Alamitos, 1981, pp. 255 - 273.
7. Mills, Harlan D., "Mathematical Foundations for Structured Programming", reprinted in Tutorial on Structured Programming: Integrated Practices, edited by Victor R. Basili and F. Terry Baker, IEEE Computer Society, Los Alamitos, 1981, pp. 42 - 107.
8. Bhansali, Kirit J., et. al., "Database Development under the ASM/NBS Program on Alloy Phase Diagrams", Proceedings of the 29th

- National SAMPE Symposium and Exhibition, Reno, April 1984, pp. 1450 - 1464.
9. Ullman, Jeffrey D., Principles of Database Systems, Computer Science Press, Rockville, 1982, p. 19.
10. Shooman, Martin L., Software Engineering, McGraw-Hill Book Company, New York, 1983, p. 107.
11. Standish, Thomas A., Data Structure Techniques, Addison-Wesley Publishing Company, Reading, 1980, p. 185.
12. Meehan, James R., The New UCI LISP Manual, Lawrence Erlbaum Associates, Hillsdale, 1979.
13. McCarthy, John, et. al., LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, 1965.
14. Minsky, Marvin, "A Framework for Representing Knowledge", in The Psychology of Computer Vision, edited by Patrick Henry Winston, McGraw Hill Book Company, New York, 1975, p. 211.
15. Winston, Patrick Henry and Berthold Klaus Paul Horn, LISP, Addison-Wesley Publishing Company, Reading, 1981, pp. 291 - 314.
16. Basili, Victor R., "Structured Program Design", reprinted in Tutorial on Structured Programming: Integrated Practices, edited by Victor R. Basili and F. Terry Baker, IEEE Computer Society, Los Alamitos, 1981, pp. 111 - 120.
17. Griffiths, S. N., "Design Methodologies -- A Comparison", reprinted in Tutorial: Software Design Strategies, edited by Glenn D. Bergland and Ronald D. Gordon, IEEE Computer Society, Long Beach, 1979, pp. 189 - 213.

 *
 * The names of computer hardware and *
 * software mentioned in this paper are *
 * intended only for communication *
 * purposes. Such mention should not *
 * be construed as endorsement by *
 * either the author or his institution. *
 *
