SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 255-265 (1979)

Make — A Program for Maintaining Computer Programs

STUART I. FELDMAN

Bell Laboratories, Murray Hill, New Jersey 07974, U. S. A

SUMMARY

Good programmers break their projects into a number of pieces, each to be processed or compiled by a different chain of programs. After a set of changes is made, the series of actions that must be taken can be quite complex, and costly errors are frequently made. This paper describes a program that can keep track of the relationships between parts of a program, and issue the commands needed to make the parts consistent after changes are made. *Make* has been in use on UNIX† systems since 1975. The underlying idea is quite simple and can be adapted to many other environments.

KEY WORDS Program maintenance Program updating

INTRODUCTION

Consider the plight of a modern programmer. Following the dictates of software engineering, he has divided his project into a number of sensible modules. He makes use of all the tools he can. Common data structure descriptions and constants are entered just once and are shared by the relevant programs. An automatically generated parser analyzes the input. The complicated logic is specified in decision tables. The output is guided by a formatter.

Now he decides to make a small change to a data structure. How does he regenerate his program? One possibility is to "recompile everything in sight". It may not be easy to put together the monster command sequence he needs. Doing all this work too often will certainly break the budget, since those marvelous tools aren't free. Alternately, he may think hard and deep, decide exactly which parts are affected by the change, figure out the sub-sequence needed to do the processing, and go to work. He has saved machine time, but if he has forgotten just one module, he now has a bug that may take days to find.

These difficulties affect projects of all sizes. Even in a simple card environment, it is hard to remember which object decks are good and which are bad. In a large project with many people working on related pieces of a product, it is very likely that parts will become inconsistent.

What is needed is a mechanism for keeping track of the status of the parts of a program, and for executing an efficient control sequence to ensure that the end product is correctly made from consistent pieces. We describe here a program called *Make* [1] that addresses the problem of keeping track of and re-creating software. *Make* provides a way to save the command sequences needed to perform transformations. It automatically executes a sufficient sub-sequence of the commands

† UNIX is a trademark of Bell Laboratories.

0038-0644/79/0409-0255$01.00

Received 1 November 1978

after checking which files have changed. *Make* was designed to simplify life for the developer and maintainer of moderately complex programs, but it has proved to be a very useful tool for a much larger range of problems than was originally expected.

The *Make* program has been in heavy use since 1975. The major implementation runs under the UNIX [2] operating system on PDP-11, VAX-11, and Interdata 8/32 computers, but a version is also available for the Honeywell 6000 GCOS. The following descriptions refer to the version running on UNIX systems, but we also discuss possible implementations in other environments. We omit many details of the program, but hope to show the power and generality of the idea.

Earlier programs have done related tasks. For example, the *Rapid Program Generator* [3] on the PDP-10 will automatically recompile if a source file is present but the associated object is either missing or older. However, *Make* is a more general and complete approach to the problem.

Basic Algorithm

We assume there is a certain set of primary source documents. These are to be transformed and combined to produce other files, including the final products. If a source file is changed, then the files that are generated from it are obsolete. The *Make* program reads a text file that describes the objects to be maintained. *Make* queries the environment to discover which files have been changed, and deduces the set of commands that must be executed to ensure that all the relevant files are consistent and up to date.

To be a little more formal, a file **B** is said to *depend* on a file **A** if a change in **B** requires **A** to be updated. The heart of the program is this rule: *To "make" a particular node* **N**, *"make" all the nodes on which it depends. If any has been modified since* **N** *was last changed, or if* **N** *does not exist, update* **N**.

This algorithm is simply a depth-first traversal of the dependency graph. *Make* must have a description of the dependencies and the ability to check that one file is up to date with respect to another. The program should also be able to issue commands. (The *Make* idea would still apply on a system where commands could not be issued. A list of control lines to be executed on a batch system would be valuable.) Thus, implementation of this algorithm places few requirements on the system.

Programs that deal with job execution seem to require many special features and options, and *Make* has proved no exception to this rule. A number of mechanisms must be introduced in the program to permit the algorithm to be applied smoothly. We describe some of these complications below, but the reader should keep in mind that the essence of the program is just this depth-first search.

An Example

Consider a tiny compiler that has two parts, a parser and a code generator. The code generator **codegen.source** is a hand-built program. The parser is specified by a grammar **parser.grammar** that is read by a parser generator, which generates a program source file **parser.source** that needs to be compiled. The two parts use a common set of definitions that describe the shared data structures; these definitions are combined with the source programs at compile time. The compiled programs are loaded with a **library** that is also subject to change.

Figure 1 shows the flow of data through various programs. Figure 2 shows *Make's* view of the problem. This dependency graph is just Figure 1 with the arrows reversed and the processing actions written below the nodes. There are four primary files, **parser.grammar**, **codegen.source**, **definitions**, and **library**. The parser generator transforms **parser.grammar** into **parser.source**. The two code files need to be compiled, and the results loaded in conjunction with **library**, to get a working program.

We can see what actions must be taken after making different changes: A change to **parser.grammar** requires regenerating **parser.source**, then recompiling and reloading. A

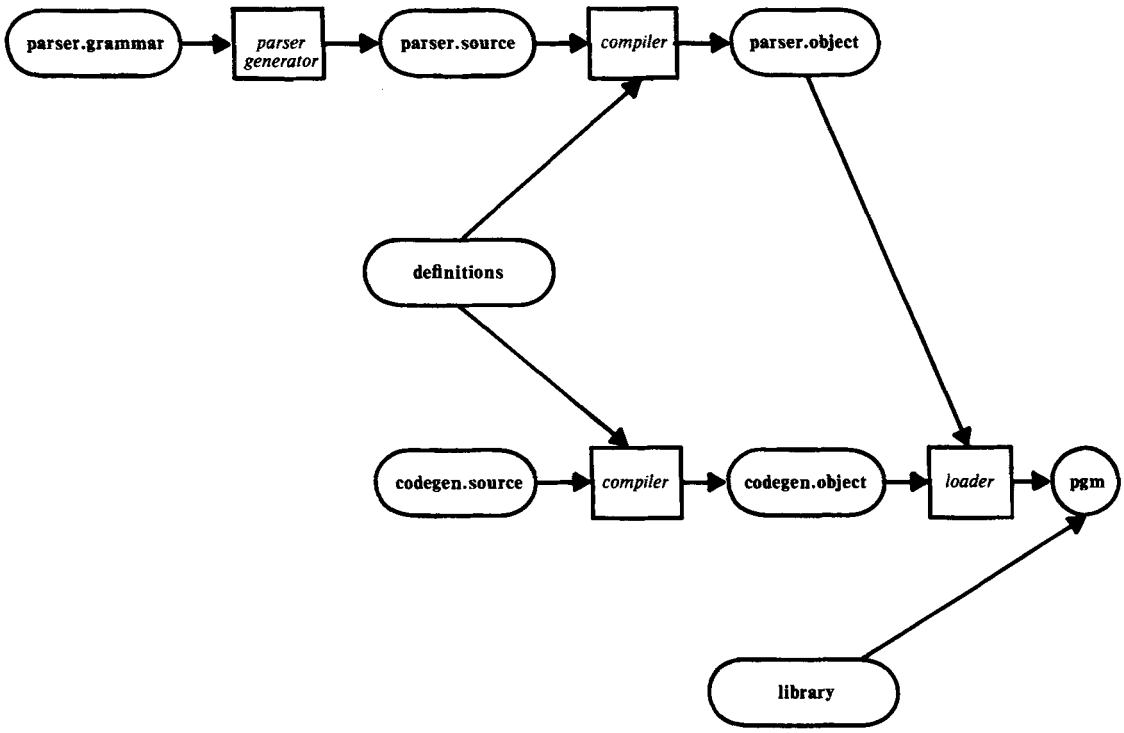

Figure 1. Data flow

change to **codegen.source** requires recompiling it and reloading. A change to **definitions** requires recompiling **parser.source** and **codegen.source**, but does not require regeneration of **parser.source**. A change to **library** requires reloading but no compiling. *Make* would issue just these command sequences in the circumstances just described.

THE DESCRIPTION FILE

Make reads a text file describing the dependency graph. The format is as follows: Blank lines are ignored. Characters from a sharp ("#") to the end of a line are ignored. A line with an embedded equals sign ("=") defines a macro (see below). A set of dependencies is declared by a line with an embedded colon; each of the names on the left depends on each of the names to the right of the colon. In the following, we call the names on the left "targets". A dependency line may be followed by one or more indented command lines. The format is thus

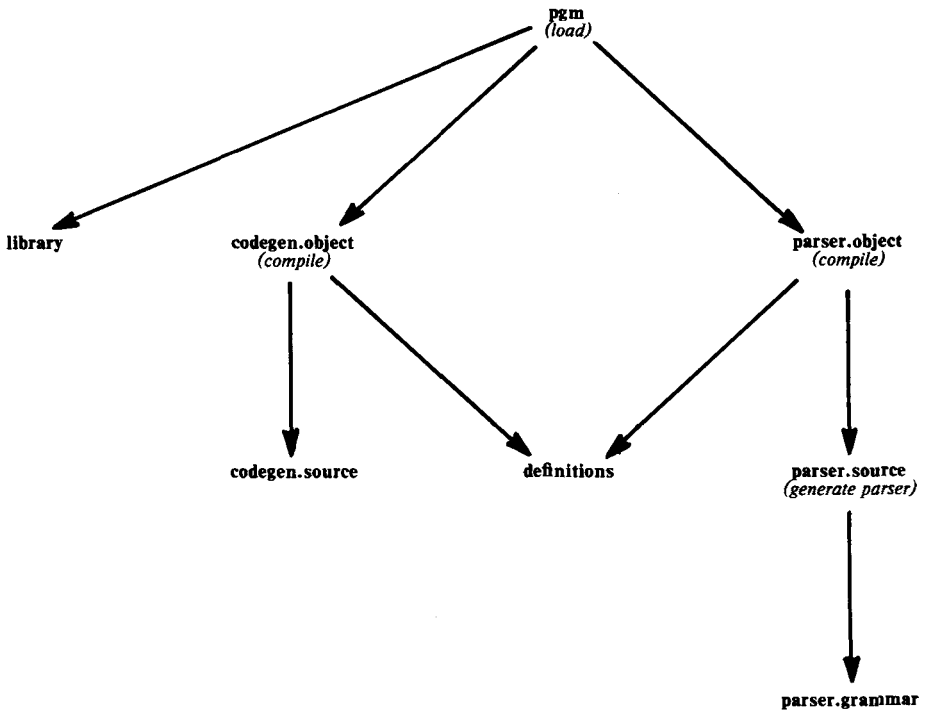

Figure 2. Dependency graph

```
targ1 targ2 : dep1 dep2      # dependency line
      command1
      command2
targ1 : dep3                 # one more dependency for targ1
another : dep
```

If a name appears on the left of more than one "colon" line, the names on the right of the colon on those lines are concatenated into a single dependency list. A command sequence may be associated with at most one of the colon lines for a particular target; in the example above, the line with **dep3** could not have a command line. It is purely a notational convenience that a target may appear on more than one colon line, but this feature is useful in very complicated description files. The associated command sequence is to be performed if any of the files on that list is out of date with respect to the target.

There is a more general dependency syntax, marked by a double-colon ("::"), which indicates that a target depends on several sets of names, and that a different command sequence may be associated with each set. If any of the files on the right side of a double-colon line is out of date with respect to the target on the left, the associated command sequence is performed.

```
target :: dep1 dep2
        command1
target :: dep3
        command2
```

Some description files are quite large: the description files for certain system libraries and other major pieces of software are over a thousand lines long. Maintenance of immense description files is clearly an error-prone process, so mechanical assistance is desirable. There is no general purpose utility that does this job, but it is sometimes possible to write a command sequence to scan the source text and accumulate references that imply dependency, then reformat this information as a description file. The details and the level of difficulty of this task depend on the system used, the language in which the programs were written, and on programming conventions that were followed. The operation of deducing the dependency graph is likely to be expensive, since the contents of a large number of files must be examined. Because of the cost and system-dependence, this facility has not been embedded within the *Make* command itself.

Figure 3 is a description file equivalent to the diagram in Figure 2. The file name suffixes have been changed to conform to actual UNIX system usage. The dependency (colon) lines are just an encoding of the arcs in Figure 2.

```
pgm: codegen.o parser.o library
        cc codegen.o parser.o library -o pgm   # load

codegen.o: codegen.c definitions
        cc -c codegen.c                 # compile

parser.o: parser.c definitions
        cc -c parser.c                  # compile

parser.c: parser.y
        yacc parser.y                   # generate parser into file y.tab.c
        mv y.tab.c parser.c             # change name of output
```

Figure 3. Complete description file

UNIX System Conventions

(In the rest of this paper we present actual examples from the UNIX system implementation of *Make*. To understand them, it is necessary to know a few facts. The file system imposes no structure on its elements, but by convention the suffix on a name identifies its content. A file whose name ends in ".o" is an object module; a suffix ".c" implies a C[4] language source file; a suffix of ".y" implies a **yacc** [5] parser-generator input. It is possible to issue a command, then wait for its completion. When a command finishes, it returns a code which, by convention, is nonzero if an error befell. We use just a few common commands in our examples. Command arguments are separated by blanks. Option and flag arguments usually begin with a minus sign ("−"). The command **mv a b** changes the name of file **a** to **b**. The command **cc −c x.c** compiles the C program **x.c** and puts the object in file **x.o**. The command **cc f1.o f2.o lib −o pgm** (no **−c** flag) loads the object modules **f1.o** and **f2.o** with the library **lib** and puts the runnable program in file

pgm. The command **yacc x** transforms the grammar **x** into a C procedure in a file with the fixed name **y.tab.c**).

Implicit Rules

The description file in Figure 3 is easy to create using a text editor, and it only needs to be typed once. To generate the first copy of the program, it would be necessary to type all of the command sequences that appear in that file, so the only extra work is in entering the dependency information.

However, *Make* is capable of working from much less information by making use of built-in transformations. Most of the command sequences in Figure 3 are stylized, and can be deduced from the types of the files involved. Figure 4 is a much shorter, but still sufficient, description file.

```
pgm: codegen.o parser.o library
        cc codegen.o parser.o library −o pgm    # load

codegen.o parser.o: definitions
```

Figure 4. Abbreviated description file

Even less typing is required to create this file than to run through the sequence of commands a single time.

How is this possible? *Make* takes advantage of the UNIX system suffix conventions. *Make* has a list of interesting suffixes, and a list of rules for certain pairs of suffixes. If *Make* encounters a node whose name has an interesting ending, it examines the description file and queries the file system to see whether there are any other files with the same stem and an interesting suffix. If it finds such a name and if it also has a transformation rule for that pair of suffixes, it applies it. The user may provide his own list of suffixes and rules, but *Make* has a default set that is usually adequate. (The suffix list is the dependency list for the name **.SUFFIXES**. The transformation rule for a pair of suffixes is the command sequence associated with the name formed by concatenating the two suffixes. An example is presented later in the paper. On UNIX systems, all suffixes begin with a period, but few file names do, so there rarely is conflict.)

COMMAND USAGE AND SPECIAL OPTIONS

The UNIX file system is a directed acyclic graph. At any time, a user has a "current directory"; any file whose name does not begin with a the character "/" is assumed to be in the part of the file system subordinate to the current directory. Directories can be created freely, and the easiest way to organize a project is to set aside a directory tree for the work. By default the *Make* command looks for a description in a file named **Makefile** in the current directory. (It is also possible to state the name of the description file explicitly on the command line.)

By default, *Make* updates the first name mentioned on a dependency line in a description file. Thus, if Figure 3 has been stored in the file **Makefile**, the simple command line

```
make
```

causes **pgm** to be re-created if any of the primary files has changed since the last version of the compiler was made. If some targets other than the first name in the description file are to be made, their names are given as arguments to the command; thus, the command line

```
make parser.o
```

ensures that the object module for the parser is up to date, recompiling or regenerating the files it depends on as required.

A number of command flags permit special uses of *Make's* capabilities. *Make* normally prints, then executes, the necessary commands a line at a time. One *Make* option causes it to print but not to execute the commands; this mode is useful for checking on the current state, or for saving a command sequence that might need to be executed again after *Make* is through. Another option causes *Make* to issue the commands silently. It is also possible to suppress printing of individual command lines; this is a convenience for long uninteresting commands.

Make normally stops if a command it executes returns an error indication. This mode of operations reduces the amount of work done and prevents compounding of errors. However, it is possible to tell *Make* to ignore errors on certain command lines (some commands return erroneous or peculiar error codes), or on all command lines.

If the *Make* command is used interactively and is interrupted by the user, it normally removes the target file that was being updated in order to guarantee a clean state for restarting the *Make* process. There is a notation for declaring that certain files should not be removed in this circumstance.

Occasionally, a set of changes will be known to be benign, but might cause a great deal of *Make* activity. For example, if a new item is added to a definitions file which many source files use, none of the compilations will be affected by the additional declaration, but *Make* will assume that all the object files are out of date. In order to prevent unnecessary activity after a safe change, *Make* can be invoked with a special option. In this mode it only touches (updates the apparent time without changing the content) the files in the right order, instead of issuing the usual command sequences. This operation changes the file system's time information for the files, so it must be used with great restraint.

Other Facilities of *Make*

Make is useful for things other than managing recompilations. It is necessary to describe some of the other capabilities of the *Make* program before discussing these applications.

Make has a rudimentary macro substitution mechanism. A macro invocation is denoted by a warning character ("$") followed by a parenthesized name; parentheses may be omitted for one-character names. User macros may be defined either in the description file or on the *Make* command line. Command line values override the description file values, so one can change actions in special circumstances. With the description file fragment

```
TESTER=stdtest
test : x
        $(TESTER) x
```

the command line

```
make test
```

executes the command

```
stdtest x
```

The command line

```
make TESTER=newtest test
```

executes the command

```
newtest x
```

User-defined macros are static — their values are computed when *Make* reads its command arguments and description files. *Make* has a few macros of its own, and these are the only ones whose values change as it operates. *Make* thus makes available the name of the target to be updated, the list of files on which the target depended that were younger than the target, and the stem of the name that induced an implicit transformation by the suffix rules. Without these facilities the implicit rule mechanism would be useless, since the transformation rules would have to refer to specific files. The default rule for transforming a C source (".c") file to an object (".o") file is

```
.c.o:
    $(CC) $(CFLAGS) -c $<.c
```

CC is a static macro that has as default value the name of the C compiler. **CFLAGS** is initially null, but can be changed to provide special compilation flags. **$<** is the name of a dynamic macro that is the prefix of the name used to invoke this rule. Thus, if *Make* needs to compile **xyz.c,** it issues the command

```
cc -c xyz
```

If it is desired to optimize all C compilations (signified by the **-O** flag on the **cc** command), it suffices to

```
make CFLAGS=-O
```

Other Uses

It is occasionally convenient to maintain a file just for its time-last-modified, in order to provide *Make* with a time marker. (Remember, *Make* maintains no permanent data of its own — all time information must come from the environment). The UNIX command **touch** updates the apparent time-last-modified of a file without actually changing its contents. The following description file entry can be used to maintain up-to-date printed listings:

```
print: f1 f2 ··· fn
    pr $          # print just the files that were changed
                  # since the last update of print
    touch print   # update the apparent time of print
```

$? is the dynamic macro variable whose value is the list of those f_i that had changed since the last change to the file **print.** The **pr** command prints those files. The **touch** command updates the apparent time on the (zero-length) file **print.** Thus, this sequence prints all the f_i that have changed since the last listing was made. If none of the files has changed since the last

```
make print
```

was executed, the **pr** command is not invoked at all, since the file **print** is younger than all the files it depends on.

If the name of a nonexistent file is encountered during the graph search, *Make* executes the associated command sequence. *Make* assumes that the file has then been created, but it does not check. Thus, the first execution of

make print

prints all of the f_i; **touch** then creates an actual file. If the command sequence does not create a target file, *Make* executes the command sequence every time it is invoked to "make" that target. A typical entry in a description file is

```
install : result
        cp result /comdir
```

which copies **result** into a command directory. If there is no actual file named **install,** every

make install

command performs the copy, after creating an up-to-date version of **result.**

It is sometimes possible to generate description files automatically. Since *Make* reads the entire description before starting work, the description can be replaced during the execution of the command:

```
Makefile : source1 source 2 . . .
        [commands to re-create the description]
```

Make is also useful for maintaining library-like files that are made up of a number of sub-files. As an example, assume we have a library file **lib** and a number of source files **s1.c**, **s2.c**, ... If we are willing to keep all the object files in the file system, the fragment

```
lib : s1.o s2.o . . .
        ar r lib $?
```

will suffice. (The UNIX command **ar r lib f1 f2** replaces entries **f1** and **f2** in **lib** with the new version.) If we do not wish to keep the extra object files, we can use the double-colon syntax discussed in the Description File section:

```
lib :: s1.c
        cc -c s1.c ; ar r lib s1.o ; rm s1.o
lib :: s2.c
        cc -c s2.c ; ar r lib s2.o ; rm s2.o
. . .
```

For each *source* file that has changed since the object library was last updated, that source file will be compiled, then the compiler output will be archived and removed.

EXAMPLE

Figure 5 is a realistic example, the actual description file for the *Make* command itself. For the author's convenience, the code is broken up into a number of C source files. In addition, there is a grammar that must be processed by the parser generator **yacc** before it can be compiled. To do a simple test, the installed and the new versions of *Make* are run, using an option that prints details of the execution; the UNIX command **diff** prints the differences between these two files. This description file encapsulates most of the commands needed to maintain *Make.*

```
# Description file for the Make command

FILES = Makefile ident.c defs main.c doname.c misc.c files.c dosys.c gram.y gcos.c
OBJECTS = ident.o main.o doname.o misc.o files.o dosys.o gram.o
CFLAGS = -O

make: $(OBJECTS)
	$(CC) -n -s $(CFLAGS) $(OBJECTS) -o make
	size make

# Definitions file is needed for all compilations
$(OBJECTS): defs

cleanup:    # remove intermediate files
	rm *.o gram.c                    # remove object files and parser file
install:    # copy new command into system command directory
	size make /bin/make              # print out size of previous and new version
	cp make /bin/make                # copy into command directory
print: $(FILES)                      # print recently changed files
	pr $?
	touch print

test: newout oldout
	diff newout oldout               # compare the two outputs
	rm newout oldout                 # remove junk files
newout:     # run new Make in this directory, save detailed output
	make -dp >newout
oldout:     # run old Make in this directory, save detailed output
	/bin/make -dp >oldout
src:        # copy all the source into the system source directory
	cp $(FILES) /usr/src/cmd/make
```

Figure 5. Description file for Make

EXPERIENCE

The *Make* program was designed for use on moderate-sized projects. It turns out to be useful even for small programs, as soon as it becomes convenient to break them into several pieces. Many commands, and even a version of the UNIX system itself, are maintained using *Make.*

Using *Make* relieves a great deal of tedium from the programming process. The cost of using *Make* is small. There are three main components of the execution: reading the description, getting the file statuses from the system, and executing the commands to do the update. Typically, it takes a fraction of a second to read a description file. More time is needed to get the file modification times; a moderately complicated graph may entail a delay of several seconds before the first command is issued. Since the typical compilation takes far more time than that, there have been few complaints about the burden *Make* itself places on the system or about the delay before work seems to begin.

The program has grown with time. Various facilities have been added as the need or usefulness has become apparent. The current UNIX system version is about 2000 lines of C and YACC input. Perhaps half of this code consists of special cases and optimizations that are relevant to the UNIX system, but are not strictly necessary for the operation of the program.

POSSIBLE EXTENSIONS AND MODIFICATIONS OF *MAKE*

The algorithm underlying *Make* is independent of the system, but some of the details of the implementation would have to be changed for other environments. The implicit rule mechanism depends very closely on UNIX system conventions. Any other mechanism for deducing chains of dependencies would be suitable. For example, in a system that maintained information on the content of a file, but permitted multiple files under a single name, a check for forms with different types would accomplish the goal directly.

At the heart of *Make* is the comparison of two files for currency. Another relation could be substituted for it, either because a system does not maintain a sufficiently fine record of the time a file was last modified, or because some other basis for comparison (e.g. version number) is more appropriate.

Make maintains no permanent data of its own. This approach has a number of small advantages: *Make* may be interrupted and restarted without fear, the program is independent of the file environment in which it is running, the description file may be replaced during execution, and the *Make* may be invoked recursively — a command in a description file may invoke another copy of *Make*. without complications. If it did indeed keep track of its actions in a file, rather than count on the file system's records, it would be possible to handle archives, remote collections of data, and other entities for which a single time value is insufficient or inappropriate.

REFERENCES

1. S. I. Feldman, 'Make — A Program for Maintaining Computer Programs', *Bell Laboratories Computing Science Technical Report #57* (1977).
2. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual, (Sixth Edition)*, Bell Telephone Laboratories, 1975.
3. Ralph E. Gorin, 'RPG Rapid Program Generator', *Stanford Artificial Intelligence Laboratory Operating Note 51.1* (1973).
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.
5. S. C. Johnson, 'Yacc — Yet Another Compiler-Compiler', *Bell Laboratories Computing Science Technical Report #32* (1975).