AN EXTENSION TO THE UNIX MAKE COMMAND TO SUPPORT CREATING AND CHECKING MAKEFILES

By

SOU-YEN YEH

Bachelor of Science

National Central University

Taiwan, Republic of China

1977

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1986

Thesis
1986
Y43e
cop.2

AN EXTENSION TO THE UNIX MAKE COMMAND TO
SUPPORT CREATING AND CHECKING
MAKEFILES

Thesis Approved:

G. E. Hedrick
Thesis Adviser

D D Fisher

[illegible]

Norman N. Durham
Dean of the Graduate College

1251296

PREFACE

This study investigates the UNIX‡ make command and its related systems. An extended system built upon the make command and named "makec" is designed and implemented. Makec supports the functions of creating makefiles and checking makefiles. The makefiles created by makec can be used for project maintenance without any modification. Some version 7 UNIX system makefiles were checked by makec and were found to be in error. These results indicate the potential importance of makec.

I wish to express my sincere appreciation to my major adviser, Dr. G.E. Hedirck, for his guidance, patience, and encouragement throughout my graduate study. Thanks are also extended to my committee members Dr. D. D. Fisher and Dr. S. A. Thoreson for their contributions and advice. A special thank goes to my parents Mr. Lai-Shing Yeh and Mrs. Bi-Yen Chen Yeh for their understanding and financial support.

‡ UNIX is a trademark of AT&T Information System.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In a typical computer project, maintenance costs can be as high as half of the total expenditures [7]. Spending so much money for maintenance is wasteful and research has been conducted to cut the expenditure [7][14]. Automation is one of the most promising ways to ease the maintenance and to reduce its costs.

An example of automation is make, a software tool which is used to automate some part of the UNIX's software project maintenance. Make requires the project inter-file dependency information as its action basis. This information usually is kept in a file named either "makefile" or "Makefile". Whenever updates are made, based upon the information in makefile, make determines which files are outdated, then issues the necessary commands to generate the up-to-date files. Thus, make saves much routine maintenance work.

Make, however, has some limitations. Two major ones are that users must create their own makefiles, and make does not check to see if these makefiles are correct. A medium-sized project usually contains hundreds of files, and has complicated inter-file dependency relationships, so the makefile may have hundreds of file names and hundreds of description lines.

Makec - An Extension to the Make System

Designing a makefile is similar to designing a piece of software. Users must spend a large amount of time creating it, but cannot be sure of the correctness of its dependency description. Any slight mistake in a makefile may cause an error which permeates the project, resulting in bugs which are difficult to trace. In order to save manpower and to decrease the number of possible errors in makefiles, the availability of a makefile automatic generating and checking tool would be very helpful. The object of this thesis is to design a system, named makec, to do this job.

A project must have all of its programs kept in one directory including its subdirectories to use makec. Thus, given the project directory name, makec can go through the statements of each source file, find the file inclusion statements which constitute inter-file dependency relationships, then create or check this project's makefile. The user's manual for makec is in Appendix B. An on-line manual and the source programs of makec are kept in the research computer in the Computing and Information Sciences Department at Oklahoma State University.

Chapter II contains a general guide for the make system, an overview of the make source programs, and its data structure. The system, makedep, created at Stanford

University, is introduced in chapter II also [12]. The design and implementation of the makec system are discussed in Chapter III. In Chapter IV, the makec system is evaluated by creating some projects' makefiles and by checking some UNIX system indigenous makefiles. Makec's overhead is measured in this chapter as well. The final chapter is a summary of the thesis and suggestions for further work.

Appendices include makedep user's manual, makec user's manual, some sample makefiles, and the output and analyses of applying makec to check makefiles.

Review of the Literature

Make, classified as a version control tool in software maintenance [5], is implemented on the UNIX operating system. It has been used heavily since 1975 and has attracted much attention especially as a tool for software project maintenance [1][4][6]. Some people have tried to improve the make system. Erickson and Pellegrin generalized make to a new system named build [2]. At Stanford University, Theimer created a buildmake system which is a preprocessor for make [11]. Also, Theimer and Mann created the makedep system which can construct dependency lines for makefiles [12].

Build is an extension of the make system that applies a concept which is named "software view". A project may have several software views, such as the developer's view, the system user's view, and the testing team's view. This software view is specified in a makefile statement such as "VPATH = dirl:dir2". Using the concept of software view, build permits several software developers to make independently a collection of software while sharing the same set of directories.

Buildmake is a preprocessor for make. It provides an extended syntax for makefiles. Buildmake works on buildfiles. A buildfile has the same syntax and the same semantic meaning as a makefile, with the addition of two features. The first feature is a statement of the form "#include filename" which requests the named file to be copied into the position occupied by the inclusion statement. And the second feature is a construct of "#ifdef/#else/#endif" which causes conditional inclusion and exclusion of statements between "#ifdef" and "#else", and statements between "#else" and "#endif".

Makedep is a system to construct file dependency lines for makefiles. It was programmed at Stanford University in 1984. Makedep provides a function similar to that provided by makec in creating the makefile function. Makec, however, provides another function : checking makefile's dependency correctness which makedep system does not offer. Since makedep is related to this thesis closely, it is discussed further in Chapter II.

Feldman did not implement the two functions for creating and checking makefiles in his make project due to the large overhead and potentially poor portability [3]. The increased confidence of makefile dependency correctness that can be obtained with makec appears to justify the overhead. In addition, computer hardware costs less and runs faster each year, so this overhead problem has become relatively minor. The portability problem can be reduced to a minimum with careful design. It is worthwhile to have these two extended functions of make.

CHAPTER II

MAKE AND MAKEDEP OVERVIEW

Introduction to Make

The make system supports many activities of program development and maintenance. Make uses the contents of a file which is named either "makefile" or "Makefile" for its directions. This file contains the information relating to a project's inter-file dependencies and command sequences for updating files. Figure 1 shows the makefile of a project named "proj" which has three C source files and one data file.

```
proj : x.o y.o z.o
        cc  x.o y.o z.o -lS -o proj

x.o  : defs.h  x.c
        cc -c x.c

y.o  : defs.h  y.c
        cc -c y.c

z.o  : z.c
        cc -c z.c
```

Figure 1. A Makefile for Project "proj"

In Figure 1, those statements which have a colon are dependency lines. Each colon separates a target which is at the left hand side of it from the dependents which are at the right hand side of it. Those lines begin with a tab character are shell commands which are used to update their associated target.

Table I shows the file name extension convention used on the UNIX operating system.

TABLE I

FILE NAME EXTENSION CONVENTION ON UNIX OPERATING SYSTEM

file name extension	file type
.c	C language source file
.s	Assembly language source file
.p	Pascal language source file
.f	Fortran 77 language source file
.l	Lex regular expression file
.y	Yacc grammar rule file
.o	Object file
.h	File to be included

Figure 2 is the file dependency diagram of project "proj". Object file "x.o" depends on files "x.c" and "defs.h". Object file "y.o" depends on files "y.c" and "defs.h". Object file "z.o" depends on "z.c", but does not depend on file "defs.h". Whenever file "defs.h" is updated, the object file "z.o" is still up-to-date, but object files "x.o" and "y.o" become outdated. The command sequences :

"cc -c x.c" and "cc -c y.c"

must be executed in order to get the new versions of "x.o" and "y.o", respectively. For the same reason, the target file "proj" becomes outdated when its dependents "x.o" and "y.o" are updated. So, the command sequence :

"cc x.o y.o z.o -lS -o proj"

must be executed in order to create the new version of "proj". A simple command "make" takes care of this part of routine maintenance.

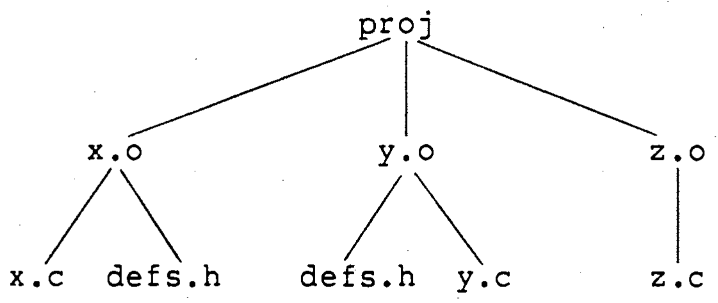


Figure 2. The File Dependency Diagram of Project "proj"

Without the make system, maintenance analysts must perform these tasks which include checking file status to find outdated files manually, reading the documentation to determine the set of commands for each outdated file, and typing in these commands. Make relieves this work from the maintenance analysts.

The make system has some built-in knowledge. For example, if a file named "x.c" is found, make assumes the object file named "x.o" depends on it and the command sequence corresponding to this dependency is :
"cc -c x.c"
These implicit rules are shown in Figure 3.

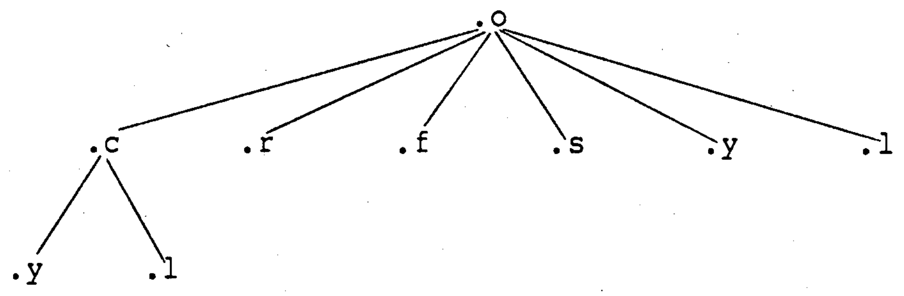


Figure 3. The Implicit Rule for the Make File Dependency Relation

In Figure 3, there is more than one route for files with the name extension ".y" or ".l". The implicit dependency route chosen is the shorter one unless an intermediate file in the longer route exists or is mentioned in the description of the makefile.

Files with extensions not mentioned in Figure 3 are not included in the implicit dependency relation in the make.

By taking advantage of these implicit rules, the simpler makefile, shown in Figure 4, for the project "proj" can work well. This makefile has exactly the same function as the longer version of makefile shown in Figure 1.

```
proj : x.o y.o z.o
        cc x.o y.o z.o -lS -o proj
x.o y.o : defs.h
```

Figure 4. A Simpler Version of Makefile for Project "proj"

The make system depends upon both the UNIX file name convention and the UNIX shell command. Make is used both with the UNIX and with the GCOS operating systems. This thesis is concerned with the UNIX system only. Feldman gives a detailed description for the make system for both UNIX and GCOS[3].

Make Source Program Overview

The make system has fifty-one source programs which are kept in seven files : main.c, dosys.c, doname.c, gram.y, files.c, misc.c and defs.h. The input-process-output diagram and the routine hierarchy diagram of the make system are shown in Figures 5 and 6, respectively.

Input	Process	Output and Result
makefile	rddescf : read description file	data structure for dependency and command sequence
output from rddescf	if (p flag) then printdesc () : print out the dependency information and shell commands	printout of file dependency information
---	if (METER set) then meter () : collect statistics for the utilization of make system	collection of the make utilization statistics
output from rddescf	doname () : check time status for target files and issue shell commands if it is necessary	execution of shell commands and result of execution

Figure 5. The Input-Process-Output Diagram of the Make System

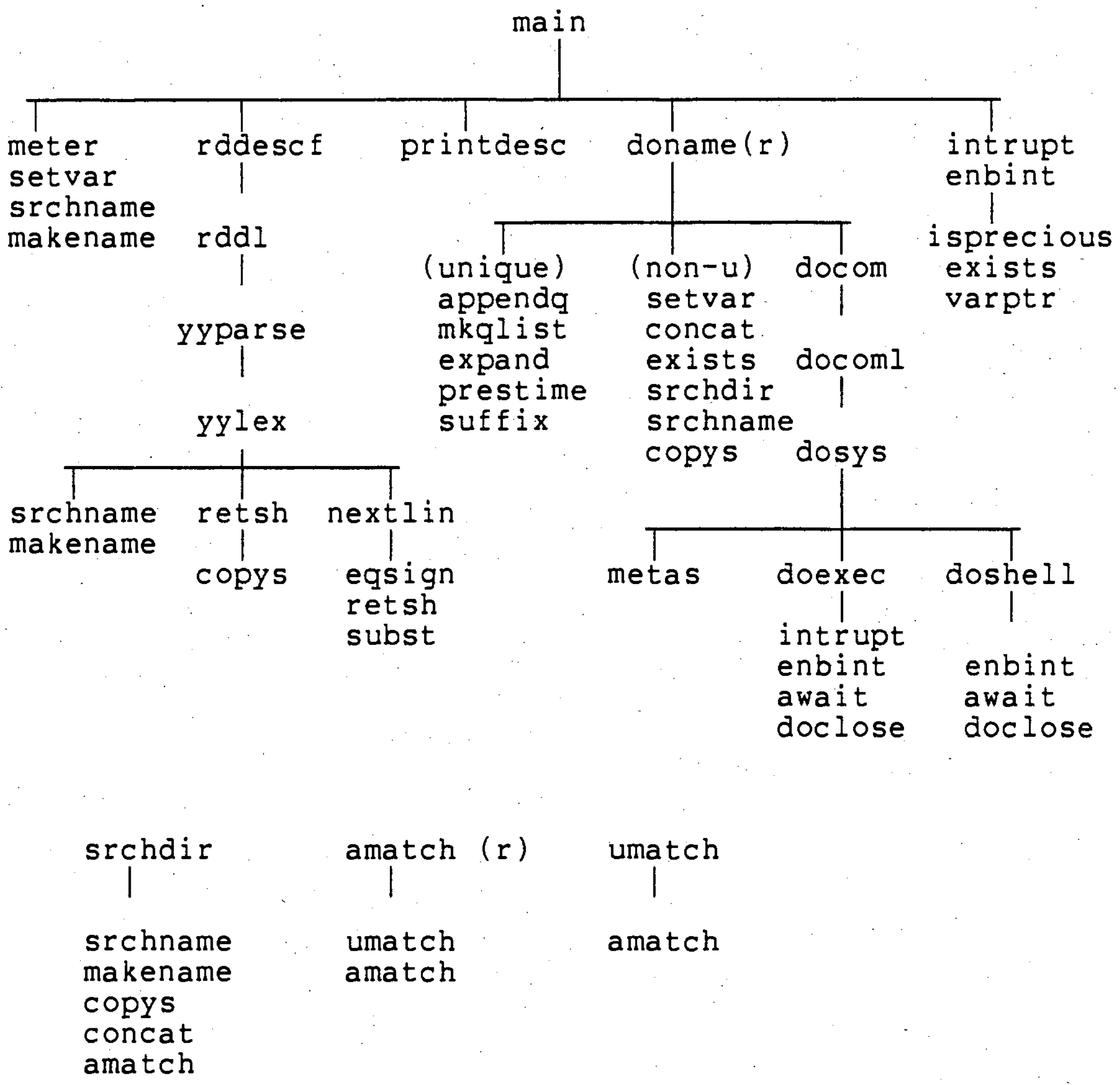


Figure 6. The Routine Hierarchy Diagram of the Make System

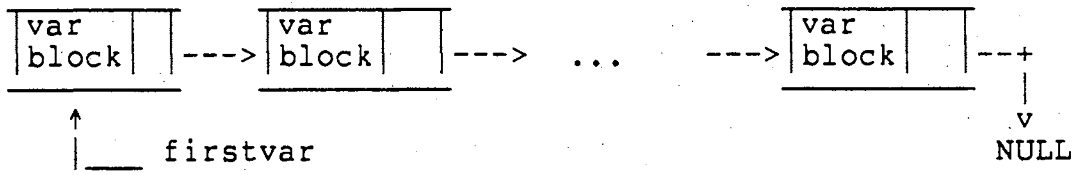
The make system's routines "yyparse", "yylex", and "nextlin" are modified and incorporated into makec system. Routine "doname" and its calling routines check the creation time of each file, find out outdated files which are older than some of its dependent files, and issue shell commands. They are not related to the makec system. Makec looks for inclusion statements in files, creates file dependency relationships based on these statements, and compares the found inter-file dependency relationship with the other source of dependency information which is kept in a makefile.

The data structures used most often in the make system are linked lists. A diagram to illustrate these data structures is shown in Figure 7. Structure variables "varblock" are used to keep macro definitions of a makefile. Variables "varblock" are dynamically allocated, then kept in a linked list with variable "firstvar" which points to the list head.

A hashing table is used to improve the file name searching efficiency. This hashing table keeps pointers pointing to variables named "nameblock". Each structure variable "nameblock" keeps relevant information of a file. Each target file's nameblock has a pointer pointing to a linked list of elements "lineblock."

A structure variable "lineblock" represents a dependency line which has a target, several dependent files, and a set of shell commands. If there is more than one

Macro definition :



Dependency and Command Information :

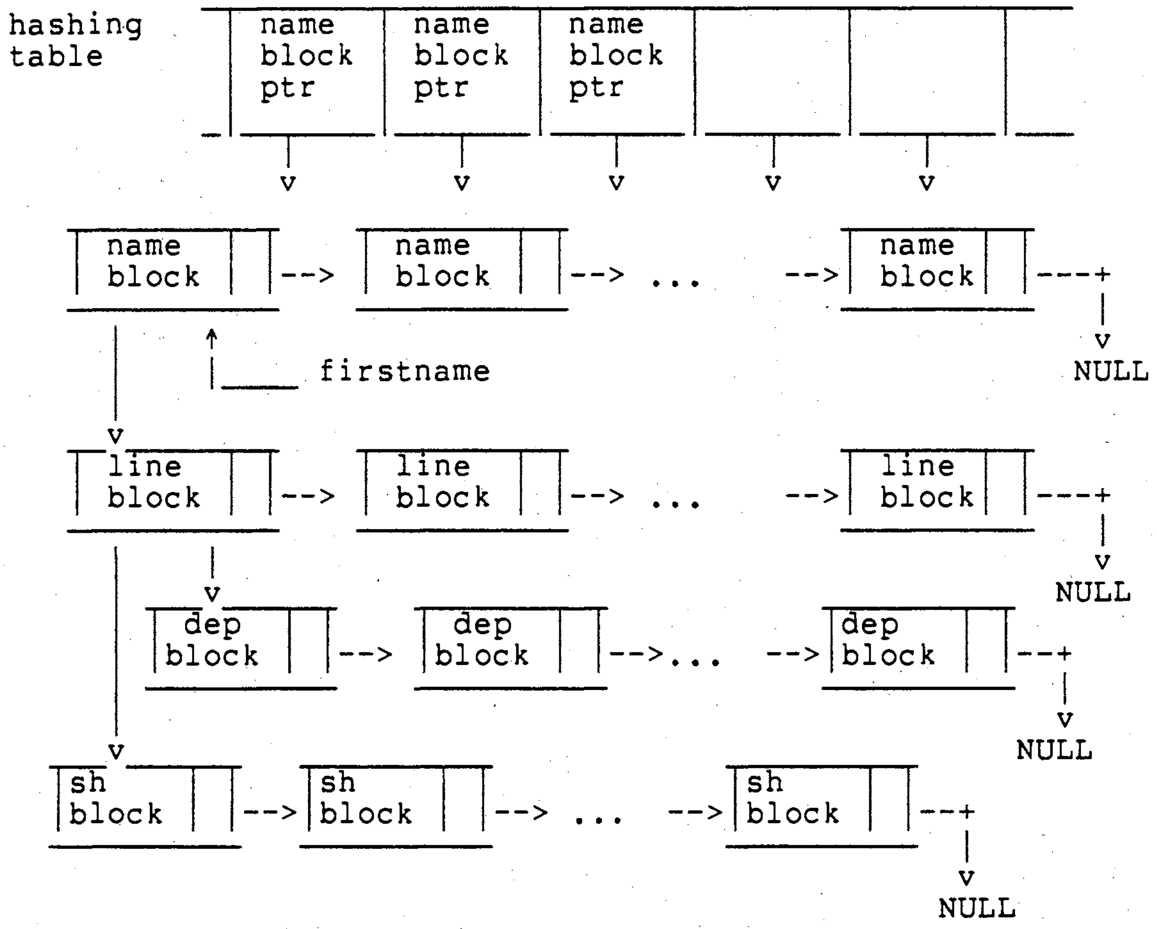


Figure 7. The Linked List Data Structure as Used in the Make System

target in a dependency line, this dependency line is expanded to several dependency lines. Each new dependency line has exactly one target, the same dependent files, and the same shell commands. Structure variable "depblock" keeps the dependent files of a target. Structure variable "shblock" keeps the shell commands of a dependency line.

Because the make system utilizes some fixed size arrays, it has some limitations on the project size and the number of dependency relationships. Table II describes all these limitations. These limitations are set for small to medium sized projects. By changing some constants defined in the data file "defs.h," make system can work for large-sized projects too.

Makedep System Overview

The makedep system was developed by Theimer and Mann in 1984. This system determines the dependencies for one or several files. This function is almost the same as the first function offered by makec system. Users can execute the makedep command on the OSU research computer by entering the command :

```
"makedep -option file ..."
```

from the keyboard. The complete on-line manual entry for makedep system is listed in Appendix A.

Makedep utilizes linked list data structures. Figure 8 shows the most important data structure used in makedep system. Linked list "SrcFiles" keeps all the source file

TABLE II

LIMITING CONSTANTS USED IN MAKE SYSTEM

constant name	defined value	description
HASHSIZE	509	hashing table size for the file names (including names expanded from meta characters)
NLEFTS	256	maximum number of LHS files in one specification line
NCHARS	500	(not used)
NINTS	250	(not used)
INMAX	2000	maximum number of characters in a dependency line
OUTMAX	2500	maximum number of characters in a set of command lines
QBUFMAX	1500	buffer size for $? (file names which are younger than the target file)

names mentioned in the makedep command line. These files are processed one by one to find out their dependents.

Linked list "IncDirs" keeps all the directory paths to be searched. Linked list "IList" keeps all the included files. Each file has a linked list "DepList" to keep its immediate dependents. Each element of "DepList" points to a file in the list "IList". The file, which is the object of the pointer, is an immediate dependent of the file which owns the linked list "DepList". In this way, the dependency relationships are represented. A linked list "iList" is not shown in Figure 9. It is a working list which keeps the immediate dependencies of a file. The information, kept in "iList," is eventually transferred to "IList".

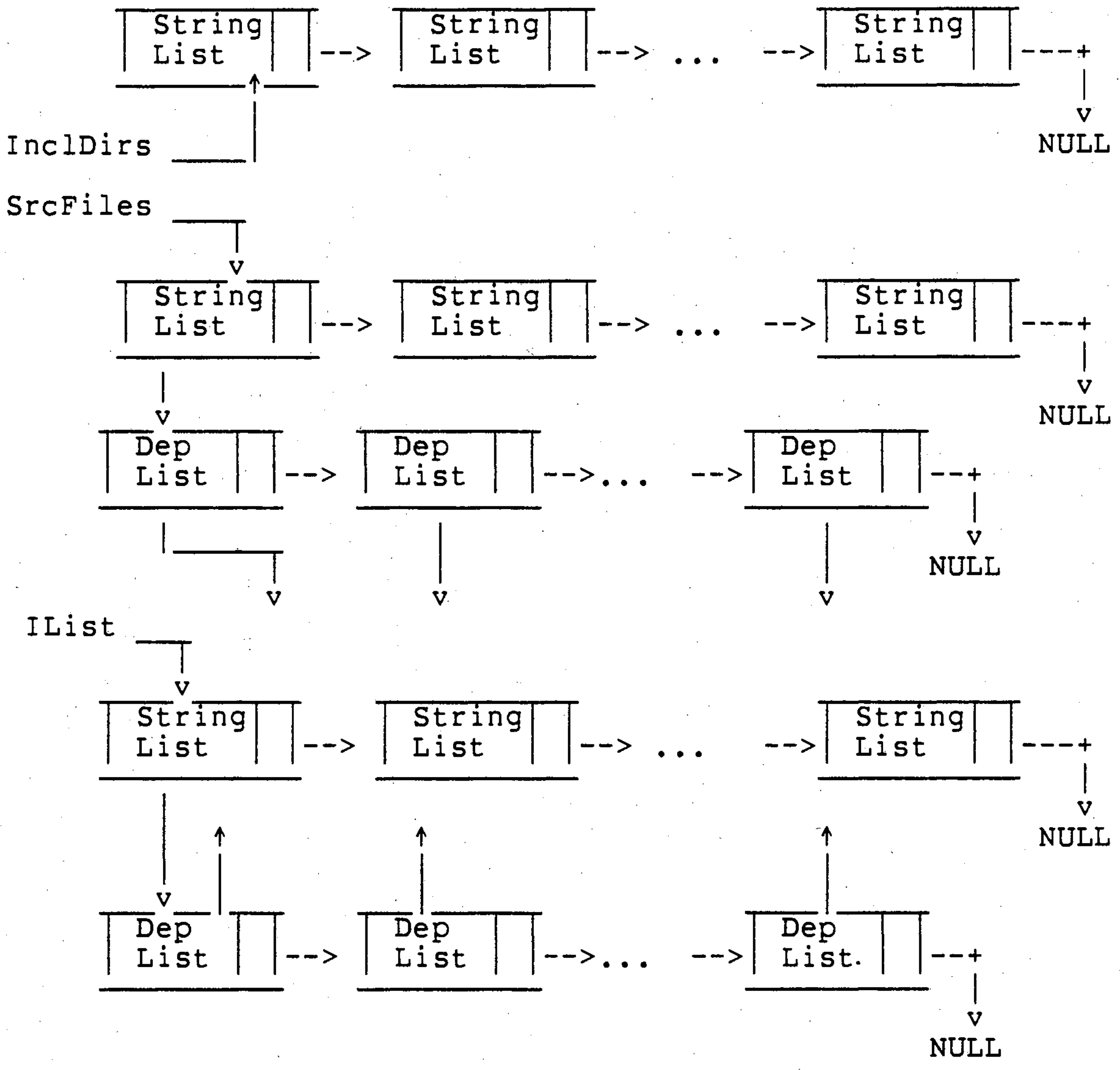


Figure 8. The Data Structure Used in the Makedep System

CHAPTER III

DESIGN AND IMPLEMENTATION OF MAKEC

This chapter discusses makec's data structure and outlines the makec system. The details of routines and some features of makec are also provided.

Introduction of Makec

The data structure of the makec system is shown in Figure 9. The variable "fl_inf[]" is a hashing array which keeps the information of a project's source files and data files. Since object file names can be deduced easily from source file names, they are not kept in this hashing array.

Each element of array "fl_inf[]" has six fields to describe a file. The first field "fl_nm" keeps a pointer which points to the file name. Together, the second field "dep_indx" and the third field "num_dep" keep the immediate dependents of the described file. For example, as shown in Figure 9, the array element "fl_inf[k]" describes a file named "any.c". This file's name is stored in a dynamically allocated storage area which is pointed to by the first field "fl_nm[k]". The contents of the second and third fields, "dep_indx[k]" and "num_dep[k]", are j and n, respectively. These two values show that the immediate dependents of file "any.c" are those files with indices kept in array elements from tdep_inf[j] to tdep_inf[j+n-1].

The fourth field "mark" is used for checking dependency information. The fifth field "dir_inf" keeps the directory path of files. The sixth field "type" is used to keep file type information.

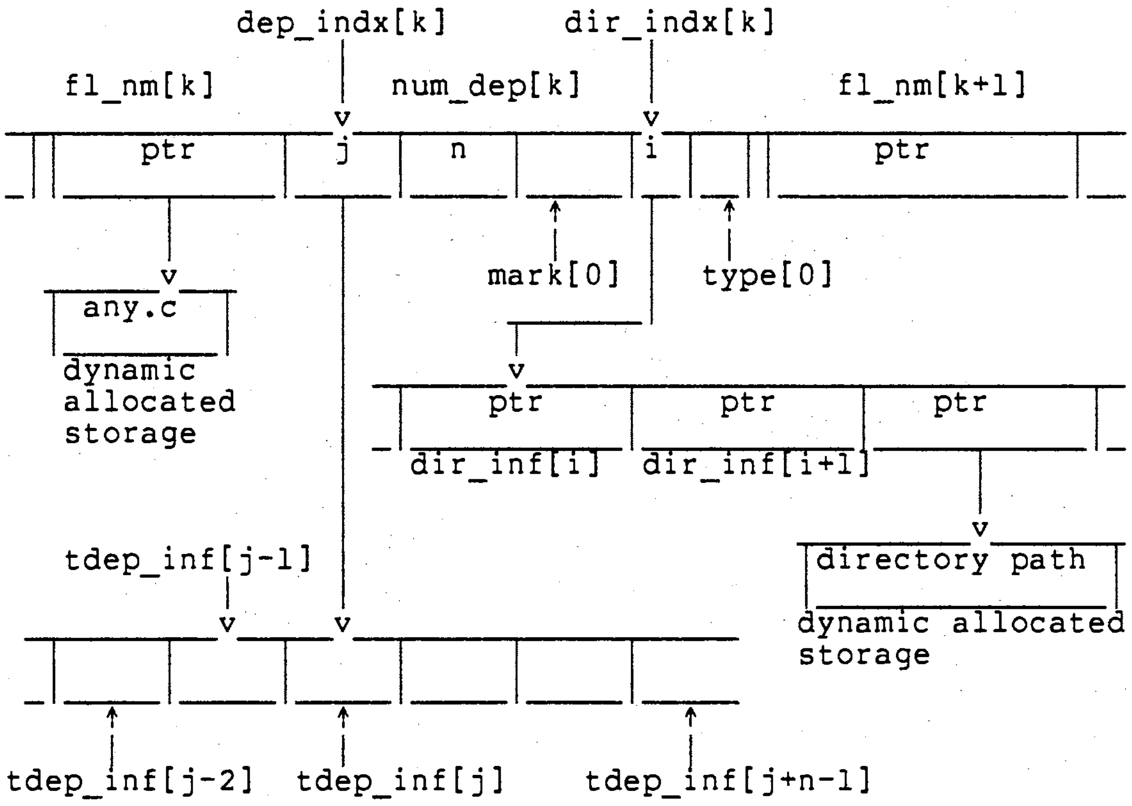


Figure 9. The Data Structure of Makec System

Since makec uses fixed length arrays to store file information, directory paths, and dependency information, there are some limitations on the project size, the number of directories in a project, and the number of file dependency relationships. Table III lists these limitations. If a project has parameters exceeding the limits shown in Table III, the constants MEDIUM or LARGE have to be defined and the command "make" must be executed. These actions generate a new executable module of makec which has larger limiting constants than that shown in Table III and can handle medium-sized or large-sized projects.

TABLE III

LIMITING CONSTANTS USED IN MAKEC SYSTEM

constant name	defined value	description
MAX_FILE	301	hashing table size for file information (total source files and data files of a project)
MAX_TDEP	600	maximum number of dependencies in a project
MAX_SDEP	60	maximum number of dependencies related to a single file
MAX_DIR	20	maximum number of directory paths referenced in a project

The input-process-output diagram of the makec system is shown in Figure 10. The straight-line characteristic of the routines makes the routine structure and the control path of the makec project very clear. To utilize this property fully, the debugging messages embedded in the makec system are arranged to show the execution result of each major routine in the control flow. The debugging messages can be triggered for any step or any combination of steps. With this arrangement, many unwanted debugging messages can be filtered out, which helps in the maintenance of the makec system. The details of controlling debugging messages for the makec system are shown in Appendix B, the makec user's manual.

Automatic Makefile Creation

The routines in the makec system are discussed following their execution sequence. Routine "proc_cmd" is makec's first routine to process the input. "proc_cmd" is an acronym for "process command". This routine processes the makec command line which includes flags, file names, and directory names.

Routine "proc_fl" is the second major routine in makec. Routine name "proc_fl" is an acronym for "process files". It and its supporting routines take the outputs from the "proc_cmd" as their inputs. These inputs are the command parameters which include file names, directory names, and flags. The status of every named file is checked. Each

INPUT	ROUTINE and PROCESS	OUTPUT
---	init() : initialize arrays fl_inf, dir_inf	fl_inf dir_inf
command line parameters	proc_cmd() : process the command line,	flag dir/file
from init() : fl_inf, dir_inf from proc_cmd() : flag, dir/file	proc_fl() : put all source files into fl_inf and dir_inf	fl_inf dir_inf
from proc_fl() : fl_inf dir_inf	proc_dep() : set up dependency among files, add data files into fl_inf	fl_inf dir_inf tdep_inf
from proc_dep() : fl_inf dir_inf tdep_inf	proc_put() : print out dependency information following the makefile format	printout of file dependency information
from proc_dep() : fl_inf dir_inf tdep_inf and an existing makefile	proc_chk() : compare the dependency information kept in arrays fl_inf, dir_inf, and tdep_inf with the dependency information kept in a makefile	printout of file dependency checking result for the makefile
fl_inf dir_inf tdep_inf	dump() : dump the content of important variables	dump output

Figure 10. The Input-Process-Output Diagram of the Makec System

named directory is searched, as is the status of each file in each directory.

Makec follows the UNIX file name extension convention. The makec file name extension convention is shown in Figure 11. Only those source files with name extension as ".c", ".y", or ".l" are accepted. These accepted files are C source files, yacc grammar rule files, and lex regular expression files. The assembly language files and Fortran language files are ignored to keep this project small. To extend makec to handle both of them is straightforward. The unqualified files could be object files, makefiles, assembly source files, Fortran source files, documentation files, and other special files. These files are ignored without warning messages.

file name extension	file type
.c	C source file
.y	Yacc grammar rule file
.l	Lex regular expression file
others	ignored unless is included

Figure 11. The Makec File Name Extension Convention

The data files are ignored in "proc_fl", but are checked by the following routines. Information about these accepted files is put into the file information array "fl_inf". If a file fits this qualification, but cannot be opened, a warning message is issued. All the directory paths encountered are put into the directory information array "dir_inf".

Usually, a small project needs only one directory which makec can accommodate well. If a project has too many files to be well organized in one directory, subdirectories are needed to group its files. Makec can handle the project with subdirectories too. Routine "proc_fl" may go through several levels of subdirectories recursively and check all the files belonging to this project. By default, the checking level of subdirectories is one (no recursion) which may be changed by applying "-l" option.

The number of directory paths allowed in a project is limited to 20, which is large enough to support small projects. If this directory path limit is too small for a large project, then changing the first line from "#define SMALL" to "#define MEDIUM" or "#define LARGE" in the data file "data.h", and running the command "make" in the makec directory can produce a new executable module of the makec system. This new module allows more directory paths.

After finishing its processing, "proc_fl" passes variables "fl_inf", "dir_inf", and "mkfl_dir" to the third routine group which includes routine "proc_dep" and its supporting routines. Routine name "proc_dep" is an acronym for "processing dependency". In this routine group, makec finds and records dependency relationships among files.

Routine "proc_dep" opens each source file which is kept in the file information array "fl_inf", and checks its statements line by line. If a file inclusion statement is found, "proc_dep" checks to see whether the included file name has been encountered before. If not, its entry is created in "fl_inf". If this included file has a new directory path, an entry is created for it in "dir_inf". The dependency relationship between these two files is recorded in the dependency information array "tdep_inf".

Multiple level inclusion is allowed in the make system, meaning an included file might include some additional files. Since this multi-level dependency information is embedded in "tdep_inf", the routine "proc_dep" does not worry about them. These multi-level dependency relationships are found out through a recursive trace in routines "proc_put" and "proc_chk".

After finishing the process of "proc_dep", all the file information, directory information, and dependency information of a project are fully gathered in arrays "fl_inf", "dir_inf", and "tdep_inf". Makec can progress either to the fourth or the fifth routine group. By default, makec goes to the fourth routine group "proc_put" which prints out the file dependency information.

The fourth major routine "proc_put" means "process output". It and its supporting routines print out a well formatted file dependency description. This printout is designed to be close to the format of manually written makefiles.

If only simple forms of the commands "cc", "yacc", and "lex" are needed to get the executable module of a project, the file dependency description output of makec can be a complete makefile for this project. If a project needs some special option of "cc", "yacc", and "lex" commands, or other commands to get the executable module, the makec's file dependency description output needs some modification to be a complete makefile.

Makec's dependency description output has two different formats. By default, makec supports the simple format which applies make built-in rules. This format is short and very readable. It fits the project which applies simple commands to get its executable module. In the second format which is obtained by specifying "-m" option, the output is a makefile which applies macros. This makefile can be more feasibly modified.

Makefiles Checker

The fifth step in makec's logic pipeline is the routine group "proc_chk". This routine group has two parts. The first part is a subset of make system. It includes make system's routines in file "gram.y" and supporting routines in file "misc.c". Routines in file "gram.y" are modified, enabling makec to reference line numbers within a makefile, and to exclude shell commands which do not concern makec. The second part of the routine group "proc_chk" compares the information kept in a makefile and the information kept in arrays "fl_inf", "dir_inf", and "tdep_inf".

In each execution session of make command, a main target is made. This main target may be specified in the command parameter, or may be the first target in the makefile. For projects of more than one source file, main target dependents are either object file names, dummy targets, or executable module names.

A dummy target is a target which is not a file name. Dummy targets are used for special dependency relationships which are not specified by file inclusion statements. They could be used to propagate dependency. It is possible to propagate the dependency through dummy targets so that the main target depends on object files only. Makec utilizes this characteristic to check the main target dependency relationship.

A UNIX project makefile is presented in Figure 12. In this makefile, "all", "cp", "cmp" are dummy files. Their dependency is propagated through target "proj" to file "x.o", "y.o", and "z.o". Through propagating dummy targets, the main target which could be either "all", "cp", or "cmp" eventually depends on all object files. By the propagation of dummy targets, the checking result is the same regardless of whether the main target is "all", "cp", "cmp", or "proj".

Makec checks a makefile in two steps. The first step is to check the minimal dependency description of the main target. The second step is to check the minimal dependency description of each object file target. A target file's dependency description is complete if all of the files in which it depends are present as its dependents in the makefile description line. A dependent file in a makefile dependency description line is necessary if its removal from the description line prohibits the necessary updating of a target file. A target file's dependency description is minimal if its dependency description is complete and all of its dependents are necessary.

```
all  : proj

cp   : proj
        cp  proj  /u/someone/proj
        rm  *.o

cmp  : proj
        cmp  proj  /u/someone/proj
        rm   *.o

proj : x.o  y.o  z.o
        cc *.o  -o proj

x.o  : defs.h  x.c

y.o  : defs.h  y.c

z.o  : z.c
```

Figure 12. A Makefile Which Applies Dummy Targets

If the main target dependency description misses any object file dependent, the dependency description is said to be incomplete. If the main target description line has some dependent which is not an object file of the project, this dependency is marked as an unnecessary dependency. The main target dependency checking result is printed for any missing dependency, any unnecessary dependency, or a confirmation message if it is minimal.

In the first step of "proc_chk", it checks the main target dependency line and the dependency lines which can be reached through propagating the dummy targets from the main target. In the second step, "proc_chk" checks other dependency lines. If the target of a dependency line is not an object file, makec simply prints a message and ignores that line. These non-object file targets could be dummy targets which cannot be reached by propagation from the main target. For example, assuming the makefile shown in Figure 12 has the main target "all", targets "cp" and "cmp" are ignored by makec because they cannot be reached through propagation from main target.

These non-object file targets could be source files or data files which are generated in special ways. They are ignored because makec can handle only the dependency caused by inclusion statements and the dependency between source and object files. In this sense, makec's checking makefile is incomplete. It needs the programmers' help to complete the checking. If a project meets the following conditions, makec can check the makefile dependency description fully for completeness and necessity :

1. It generates files only using C compiler, lex, and yacc commands.

2. All the files in the project follow the file name convention.

3. The makefile applies only the dummy targets which can be reached from the main target's propagation.

The routine hierarchy diagram of makec is shown in Figure 13. The description of routines is in Appendix E. All the routines accessed by "yyparse" are from the make system. These routines are used without modification except the three routines, "yyparse", "yylex", and "nextlin".

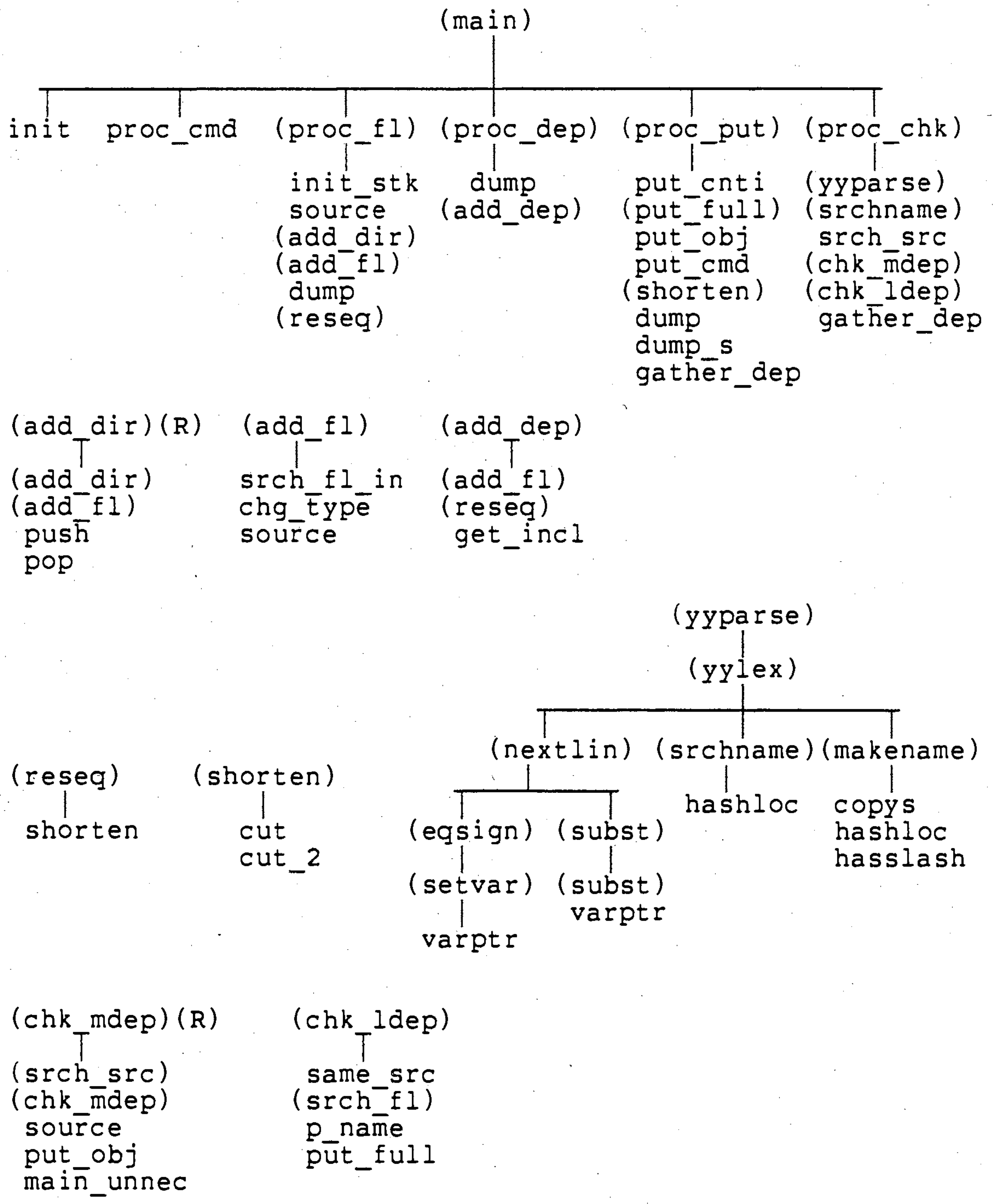


Figure 13. The Routine Hierarchy Diagram of Makec System

CHAPTER IV

APPLICATION AND EVALUATION OF MAKEC

In this chapter, the makec command is applied both to create makefiles and to check makefiles. The feasibility of these created makefiles is evaluated and the checked results are analyzed. Also, makec's overhead is measured then compared to the overhead of some other UNIX commands.

Creating Makefiles for Projects

Command "makec" is invoked for two projects "makec" and "B". Two makefiles shown in Figures 14 and 15 are created by the makec for these two projects.

Figure 14 shows the makefile created by makec for the makec project. A comparison of this makefile with the one actually used in makec project, found in Appendix C, reveals that they are similar. One difference, however, is that the makefile created by makec uses exactly one dependency line to describe the dependency of one object file, but project makec's makefile often describes the dependency of several object file targets on one dependency line. So, the makefile created by makec is longer than the makefile used in this makec project. However, both files have exactly the same function.

```
a.out    :   proc_put.o  main.o  proc_chk.o  sup_put.o\
             proc_dep.o  sup_fl.o  gram.o  proc_cmd.o\
             proc_fl.o  sup_chk.o  dump.o  sup_chk2.o\
             support.o
        cc *.o

proc_put.o      :   data.h  flag.h
main.o          :   data.h  mdata.h  flag.h
proc_chk.o      :   defs.h  data.h  flag.h  data_chk.h
proc_dep.o      :   data.h  flag.h
sup_fl.o        :   data.h  flag.h
gram.o          :   defs.h  data_chk.h
proc_cmd.o      :   flag.h
proc_fl.o       :   data.h  flag.h
sup_chk.o       :   defs.h
dump.o          :   data.h
sup_chk2.o      :   defs.h  data_chk.h
support.o       :   flag.h
```

Figure 14. The Makefile Created by Makec for Project Makec

Figure 15 shows the makefile created by makec for a sample project "B". This makefile is much different than the makefile actually used in project "B", shown in Appendix B. The makefile created by makec can only support the simple function of checking the current executable module's status and updating the executable module if it is outdated. The makefile used in project "B", however, uses many dummy targets. As well as generating the up-to-date module, this makefile offers many functions. These functions are comparing the newly generated executable module to an old executable module in the directory "/u/yeh", copying the new module to the directory "/u/yeh", cleaning all the object files, and others.

```
a.out  :  main.o  a.o  b.o  c.o  gram.o
        cc *.o

main.o       :  data.h
a.o          :  data.h
b.o          :  data.h
c.o          :  data.h
gram.o       :  data.h
```

Figure 15. The Makefile Created by Makec for Project B

Because makec can generate makefiles which offer only a simple function, if more functions are needed for makefiles, users must tailor their own makefiles or modify the makefiles which are created by makec. For more feasible modifications, makec offers a second form of makefiles. By applying the parameter "-m" in the command "makec", the output is a makefile with macros. Since the modifications can be made by using macros instead of a long list of file names, this form of makefile is easier to modify. Figure 16 shows makec's created makefile using the macro option for the makec project.

```
CC     = cc
CFLAG = -O
OBJECTS =  proc_put.o  main.o  proc_chk.o  sup_put.o\
           proc_dep.o  sup_fl.o  gram.o  proc_cmd.o\
           proc_fl.o  sup_chk.o  dump.o  sup_chk2.o\
           support.o


a.out             :  $(OBJECTS)
       $(CC) $(CFLAG)  *.o

proc_put.o        :  data.h  flag.h
main.o            :  data.h  mdata.h  flag.h
proc_chk.o        :  defs.h  data.h  flag.h  data_chk.h
proc_dep.o        :  data.h  flag.h
sup_fl.o          :  data.h  flag.h
gram.o            :  defs.h  data_chk.h
proc_cmd.o        :  flag.h
proc_fl.o         :  data.h  flag.h
sup_chk.o         :  defs.h
dump.o            :  data.h
sup_chk2.o        :  defs.h  data_chk.h
support.o         :  flag.h
```

Figure 16. The Makefile with Macros Created by Makec for Project Makec

Checking Indigenous Makefiles in UNIX

Besides creating makefiles, makec checks makefiles. For the purpose of evaluation, some makefiles indigenous to the UNIX system are checked by makec. Appendix D contains the checking output and analyses. Reviewing these analyses shows that many situations cause makec issuing main target dependency error messages. Such situations could be a project directory containing some source files which are not used in the project, the main target having some dependents which are source files or data files, and a project directory containing more than one executable module.

The output shown in Appendix D verifies that makec correctly issues messages for all the erroneous object file target dependency lines.

Situations in which makec issues error messages are described below :

1. Any file which is included by some files but is not present causes the error message "a missing file". Sometimes, this message is incorrectly issued. For example, the data file created by yacc is usually removed after completing a project because it can be generated again simply by issuing the command "yacc -d". In this case, the error message "a missing file" is still issued.

2. A directory containing some source files which are not used by the project causes the error message "main target dependency is incomplete". Makec assumes all the directory source files are members of the project and their object files are dependents of the main target. If some source files are not used by the project, their object files are not dependents of the main target in the makefile description. Thus, inconsistency occurs, and makec indicates that the makefile's main target dependency description is incomplete.

3. Source files which are not C source files, yacc grammar files, lex regular expression files, or data files are ignored. This could cause an error message of "main target has unnecessary dependencies" in case a project has some assembly source files, or Fortran source files.

4. A project directory containing more than one executable module might cause an error message. Applying a dummy main target which depends on all the executable modules can bypass the error message. Makec's checking criterion for the main target dependency completeness is that the main target must depend on all object files. Appendix C shows a makefile of the sample project "C" which has three executable modules. The dummy main target "all" is applied to depend on these three executable modules, and eventually depends on all

object files via propagation. This makefile is positive for makec's checking. A makefile of a sample project "D" is also shown in Appendix C. This makefile also has more than one executable module and is flagged by makec for "main target dependency is incomplete", because the main target does not depend on all object files.

5. File names which do not follow the UNIX file name extension convention may cause error messages, and may cause errors to be concealed. For example, a project has three C source files "e0.c", "e1.c", "e2.c", and an included data file which should have file name extension ".h". This data file may be named "e3.c", a violation of the UNIX file name extension convention. Makec treats all the files with name extension ".c" as C source files, and requests their object files be dependents of the main target. In this case, if the main target dependency description is correctly described as "Proj : e0.o e1.o e2.o", makec issues an error message "main target dependency is incomplete" for this dependency line. If the main target dependency description is wrongly described as "Proj : e0.o e1.o e2.o e3.o", makec passes this dependency line.

6. Makec catches all the dependency incompleteness and unnecessary dependents on object file target dependency lines. As shown in Appendix D, the error messages issued by makec in these cases are completely matched with the errors.

More detailed analyses of the output from makec when it checks makefiles are shown in Appendix D. From the summaries shown above and the analyses in Appendix D, three conclusions can be drawn.

1. The error messages related to the main target do not necessarily mean an error in the makefile.

2. Some dependency lines are ignored by makec. Its dependency completeness and dependency necessity cannot be checked by makec. Makec checks only those dependency lines whose targets are dependents of the main target.

3. The error messages related to the dependency line whose target is an object file most likely means errors of the checked makefile.

Execution time of Makec

The UNIX command "time" is applied to measure makec's execution time. It is compared with the commands make and makedep in Table IV. This comparison shows that the makec's execution time is close to the execution time of makedep which has fewer functions than makec. Makec's execution time is about three times of execution time of make, but make does not scan source files for the project. Thus, it appears that makec's execution time is consistent with the related commands and could be used by UNIX programmers as a program development tool.

TABLE IV

EXECUTION TIME COMPARISON AMONG SOME UNIX COMMANDS, MAKEDEP AND MAKEC

COMMAND	MEAN	MINIMUM	MAXIMUM	VARIANCE	STANDARD DEVIATION
make	1.33	1.20	1.50	0.17	0.41
makec	3.83	3.70	4.50	1.03	1.01
makec -c	4.38	4.00	5.20	2.25	1.50
makedep	4.12	3.90	5.10	2.05	1.43

Notes :

1. These execution time measurements are conducted by applying the UNIX command "time" on the PE 3230 when the system is not busy.

2. Each command execution time is measured 50 times.

3. Command "make" is applied to project "makec" which has an up-to-date executable module. Project "makec" has 18 source files, totals 70,930 characters.

4. Command "makec" is applied to project "makec".

5. Command "makec -c" is applied to project "makec".

6. Command "makedep" is applied to project "makec".

CHAPTER V

SUMMARY AND SUGGESTIONS FOR FURTHER WORK

Summary

This thesis describes an investigation of the UNIX make system and some of its related systems. A new makec system has been implemented which can create makefiles and can check makefile dependency descriptions for dependency completeness and dependency necessity.

Makefiles created by makec are shown useful for some applications. Some makefiles which have been used for a long time in UNIX system were checked by makec, and it was found that these makefiles' dependency descriptions had errors, including superfluous dependencies. The importance of the functions offered by makec thus are confirmed.

Suggested Further Work

The makec system can be improved in many ways. These suggested improvements are listed in next paragraphs one by one.

Consideration for efficiency has not been made in coding the makec programs. Should makec system become popular, efficiency improving techniques should be applied to recode the makec's most time-consuming programs to reduce its overhead.

Makec accepts C source files, yacc grammar files, lex regular expression files, and data files, but ignores FORTRAN source files and assembly language files. Handling these two source language files is suggested as further work.

In order to improve the efficiency of makec, it is helpful to assume a makefile has correct file dependency description by its last modification time. This assumption can largely reduce the overhead of makec in some cases. It can be set as an option of makec's makefile dependency checking function. In this option, since the makefile is assumed to have correct dependency description by its last modification time, only those files whose modification time is newer than makefile's have to be checked through. Thus, if a makefile is believed to be correct by its last modification time, adopting this option can save much file checking time and get the same checking result. Should makec become popular, this option is suggested as an enhancement.

Makec tries to open every source file and issues error messages for any file which cannot be opened. A yacc generated data file which is named as "y.tab.h" by default could cause trouble [8]. This data file is usually removed after completing a project because it can be generated again by issuing command "yacc -d". Makec does not have information to recognize this file. So, makec wrongly issues error messages whenever a yacc generated data file is included in some files but is not present. Although yacc data files are not used very often and this defect of makec is not a large problem, special handling of this data file is a suggested improvement of makec.

Makec system utilizes some routines and most of the data structures of the make system. There is data redundancy on makec's checking option. In this checking option, makec keeps two sets of data dependency information separately. One set of information represents the project's makefile, the other set of information is gathered by checking all source files of this project. Thus, every source file name which appears in the makefile is stored twice, once in data "fl_inf" and stored in data "nameblock" again. Another example is that there are two hash arrays. Removing this data redundancy is an improvement of makec. This improvement can save storage and can reduce a little overhead for makec system. The overhead would not be reduced much by this improvement because only a little overhead of makec is caused by the routines which execute the checking function.

In some cases, users need to add some directory paths in the included file search list. This option is offered in makedep system but neither the make nor the makec system. Makec applies only default searching directory paths. Having this additional option can be an improvement to makec also.

Makedep cannot recognize that different names may be associated with one file. For example, in the UNIX system "../proj/a.c", "/u/someone/proj/a.c", "a.c", and "./a.c" could mean the same file. Makec can recognize some different names of a file, but not all of them. Recognizing all different names of a file is another improvement which could be made to makec.

SCCS and RCS are used to keep more than one version of source text. They can save space and can keep track of revisions of source text. Each of them can be used together with make system under some restrictions. Accepting SCCS files and RCS files is another further improvement for the makec system.

REFERENCES

[1] Becker, R. A., J.M. Chambers, "Design of the S System for Data Analysis." Communications of the ACM, 27, 5 (1984), 486-495.

[2] Erickson, V. B., J. F. Pellegrin, "Build - A software Construction Tool." AT&T Bell Laboratories Technical Journal, 63, 6 (1984), 1049-1059.

[3] Feldman, S. I., "Make - A Program for Maintaining Computer Programs." Software - Practice and Experience, 9, 4 (1979), 255-265.

[4] Gehani, N. H., "An Electronic Form System - An Experience in Prototyping." Software - Practice and Experience, 13, 6 (1983), 479-486.

[5] Glass, R. L., R. A. Noiseux, Software Maintenance Guidebook. Englewood Cliffs, N.J. : Prentice-Hall, 1981.

[6] Griswold, R. E., "A Tool to Aid in the Installation of Complex Software System." Software - Practice and Experience, 12, 3 (1982), 251-267.

[7] Guimaraes, T., "Managing Application Program Maintenance Expenditures." Communications of the ACM, 26, 10 (1983), 739-746.

[8] Johnson, S. C., "YACC : Yet Another Compiler-Compiler." Unix Programmer's Manual, 7th Edition, 1979.

[9] Kernighan, B. W., D. M. Ritchie, The C Programming Language. Englewood Cliffs, N.J. : Prentice-Hall, 1978.

[10] Rochkind, M. J., "The Source Code Control System." IEEE Transactions on Software Engineering, SE-1, 4 (1975), 364-370.

[11] Theimer, M., "Buildmake - Preprocessor to provide Extended Syntax for Makefiles." (Unpublished programs presented in CSNET network, 1984) Stanford University, 1983.

[12] Theimer, M., and T. Mann, "Makedep - Construct

Dependency Lines for Makefiles." (Unpublished programs presented in CSNET network, 1984) Stanford University, 1984.

[13] Tichy, W. F., "Design, Implementation, and Evaluation of a Revision Control System." Sixth International Conference on Software Engineering. Proceedings, Tokyo, Japan. September, 1982.

[14] Vessey, I., and R. Weber, "Some Factors Affecting Program Repair Maintenance : An Empirical Study." Communications of the ACM, 26, 2 (1983), 128-134.

APPENDIXES

APPENDIX A

MAKEDEP USER'S MANUAL
(Provided by Stanford University)

NAME

makedep - construct dependency lines for makefiles

SYNOPSIS

makedep [options] [source files]

DESCRIPTION

Makedep constructs a makefile-style dependency list showing which header files the object files constructed from the given source files depend upon. The dependency of the object file upon the source file is not indicated in the output; this dependency can normally be inferred by the make program.

Makedep handles nested includes properly, propagating dependencies of one header file upon another back to each object file whose source file includes the dependent header file.

The following options are accepted. In options that take an argument, the space between the option letter and the argument is optional.

-o file	Output file name. The default is "dependencies". The name "-" indicates standard output.
-I dir	Add dir to the include file search list. Multiple -I options accumulate, building the search list from left to right, with the system include directories added at the end. Directory names are interpreted relative to the directory from which makedep is invoked.
-U	Use the standard Unix header directories as the system search list. Equivalent to specifying -I/usr/include after all other -I options.
-V	Use the standard V-System header directories as the system search list. Equivalent to specifying the options -I/usr/sun/include -I/usr/local/include -I/usr/include after all other -I options.
-xV	Use the experimental V-System header directories as the system search list. Equivalent to specifying the options

-I/usr/sun/xinclude -I/usr/sun/include -I/usr/local/include -I/usr/include after all other -I options.

-N	Use no system search list. Suppresses the warning message ordinarily printed when a header file cannot be found. This option is useful when you are not interested in dependencies on system include files.
-e ext	Object files have extension ".ext". Defaults to .b if -V or -xV is specified, .o otherwise.
-d	Turn on debug output. Useful only to the maintainers.

If the source files depend on any header files in standard system include directories, one of the options -U, -V, -xV, or -N should normally be specified. These four options are mutually exclusive. If none of these options is given, only the directories specified in -I options are included in the search list (as with the -N flag), but warning messages are still printed for any header files that cannot be found.

SEE ALSO

make(1)

DIAGNOSTICS

A warning is printed for each included file that cannot be found. Other errors are fatal; the messages should be self-explanatory.

BUGS

Pathnames that are excessively long may be silently truncated or cause crashes.

Makedep does not know that the same file can have two different names, for example "bar.h" and "foo/../bar.h". This means it will fail to detect loops in the dependency graph if the pathnames grow in this way while it is following the loop. The loop will eventually terminate due to the previous bug, and garbage output will result.

AUTHORS

Marvin Theimer and Tim Mann, Stanford.

APPENDIX B

MAKEC USER'S MANUAL

NAME
    makec - create and check makefiles

SYSOPSIS
    makec [option] ... file|directory ...

DESCRIPTION

Makec takes the argument files/directories as components of a project, finds the dependency relationships among them, creates or checks a makefile for the project.

In option "-c", Makec checks an existing makefile for minimal dependency. If "-c" option is not present, creating a makefile is the assumed option. If the file/directory argument is not present, the working directory is taken as the project's directory. Dependency relationships concerned by makec correspond to the file inclusion statements.

Makec takes all the files and directories in the argument list as components of a project, and finds file dependency relationships among them. In the checking option, makec checks dependency completeness and dependency necessity for the project, and then checks the same conditions for each file. A project is assumed to depend on all of the files and all of the directories in the argument list, and is assumed to depend on nothing else. A directory name in the argument list requests all the source files in this directory or in this directory's subdirectories as components of the project. Subdirectories of the argument directory are searched, and all the source files found in these subdirectories are taken as components of the project. The default searching level of subdirectories is one.

An object file depends on its source file, and all the data files included by its source file. In the case of multi-level inclusion, all the included data files are dependents of the object file regardless of the number of inclusion level.

Other options:

-s System files included by < > are counted for dependency. By default, their dependency is ignored.

-l[number]
Number of searching levels of subdirectories is specified. By default, only one level of subdirectories is searched.

-c Checking an existing makefile based on the dependency information gathered by makec. Without this option, a makefile is created on standard output.

-m Macros are used in the created makefile. By default, the output is a makefile of simple format.

-q Quick checking is applied. This option is ignored if -c option is not present. The makefile is assumed to have correct dependency description by its last modification time.

-f[name]
Name the makefile. This option is ignored if -c option is not present. By default, the makefile name is "makefile", or "Makefile". If the compared makefile has another name, it must be specified in this option.

-t[name]
Name the main target. This option is ignored if -c option is not present. By default, the main target is the first target of a makefile. This option offers a second choice.

-I[directory]
'#include' files whose names do not begin with '/' are always sought first in the directory of the file argument, then in the directories named in -I options, then in directories on a standard list.

-d[x]
Debug and dump.
Many debugging options can be chosen depending on the value of 'x' :

null,	data dump option is set.
i,	debugging option for the procedures "init" and "proc_cmd" is set.
f,	debugging option for the procedure "proc_fl" and its supporting routines is set.
d,	debugging option for the procedure "proc_dep" and its supporting routines is set.
p,	debugging option for the procedure "proc_put" and its supporting routines is set.
c,	debugging option for the procedure "proc_chk" and its supporting routines is set.
s,	debugging option for common supporting

routines is set.
a, debugging option for all routines is set.
More than one option may be chosen.

EXAMPLES

Create a makefile for the project which contains all the source files in the working directory :

```
makec
```

Check a project's makefile for minimal dependency. This project is assumed containing all the source files in the working directory :

```
makec -c
```

Create a makefile with macros for the project whose source files are kept in the directory "dir" :

```
makec -m dir
```

FILES

makefile, Makefile

SEE ALSO

make(1)
S. I. Feldman, _Make - A Program for Maintaining Computer Programs_

BUGS

If the data file "y.tab.h" which is created by yacc is not present, a "missing file" error message is issued.

Fortran source files and assembly source files are not accepted. Only C source files, yacc grammar files, lex regular expression files, and data files are accepted as project components.

Makec recognizes some multiple names of a file, but not all of them.

AUTHOR

Sou-Yen Yeh, Oklahoma State University.

APPENDIX C

MAKEFILES OF PROJECTS MAKEC, B, C, AND D

A. Makefile for Project Makec

```
a.out : main.o  proc_cmd.o  proc_fl.o  proc_dep.o  \
    proc_chk.o   proc_put.o  support.o  sup_fl.o \
    sup_chk.o    sup_put.o   gram.o     dump.o \
    sup_chk2.o
         cc *.o

main.o                     :  mdata.h  data.h  flag.h

support.o proc_cmd.o       :  flag.h

proc_fl.o proc_dep.o       :  data.h  flag.h

proc_put.o sup_fl.o        :  data.h  flag.h

proc_chk.o                :  data.h  defs.h  data_chk.h  flag.h

gram.o  sup_chk2.o        :  defs.h  data_chk.h

dump.o                    :  data.h

sup_chk.o                 :  defs.h
```

B. Makefile for Project B

```
FILES   = main.c  a.c  b.c  c.c  gram.y  data.h\
          makefile
OBJECTS = main.o  a.o  b.o  c.o  gram.o

all:    a.out

cmp:    a.out
        cmp a.out  /u/yeh/projb
        rm *.o gram.c a.out

cp:     a.out
        cp a.out  /u/yeh/projb
        rm *.o gram.c a.out

a.out:  $(OBJECTS)
        $(CC)  $(OBJECTS)  -o a.out

$(OBJECTS):  data.h

clean:
        -rm *.o gram.c

install:
        cp  a.out  /u/yeh/projb

lint :  main.c  a.c  b.c  c.c  gram.c
        lint  main.c  a.c  b.c  c.c  gram.c
        rm gram.c
```

C. Makefile for Project C

```
OBJ0  =  any00.o  any01.o  any02.o  any03.o  any04.o\
         any05.o

OBJ1  =  any10.o  any11.o  any12.o  any13.o  any14.o\
         any15.o  any16.o

OBJ2  =  any20.o  any21.o  any22.o  any23.o


all   :  any0  any1  any2

any0 :   $(OBJ0)
         cc  -o  any0  $(OBJ0)

any1 :   $(OBJ1)
         cc  -o any1  $(OBJ1)

any2 :   $(OBJ2)
         cc -o any2  $(OBJ2)


$(OBJ0) : any0.h

$(OBJ1) : any1.h

$(OBJ2) : any2.h

clear :
         rm *.o
```

D. Makefile for Project D

```
all  :  da

da :  da1.o  da2.o  da3.o
        cc   da?.o  -o da

db :  db1.o  db2.o  db3.o
        cc   db?.o  -o db
```

APPENDIX D

OUTPUT AND ANALYSES OF APPLYING MAKEC TO CHECK SOME UNIX INDIGENOUS MAKEFILES

The output of the makec's checking option has four parts.

Part I.
Part I prints message for missing included files. If there is not any missing file, no message is printed at all.

Part II.
Part II is printed under the heading 'MAIN TARGET dependency necessity checking ::::::::::'. This part prints messages for any unnecessary dependent of the main target. Message 'OK' means no unnecessary dependent.

Part III.
Part III is printed under the heading 'MAIN TARGET dependency completeness checking :::::::'. This part prints messages for any missing dependent of the main target. Message 'OK' means no missing dependent.

Part IV.
Part IV is printed under the heading 'OBJECT FILE TARGET's dependency checking :::::::::'. This part prints messages for any missing dependent and any unnecessary dependent of the dependency lines which have object file targets, and messages for ignored dependency lines.

Output of Applying Makec to Check "adb/makefile"

```
MAIN TARGET dependency necessity checking ::::::::::
        OK

MAIN TARGET dependency completeness checking :::::::
        OK

OBJECT FILE TARGETS' dependency checking  ::::::::::
(          cmp # 13 ) ignored, dummy or non-object target
(           cp # 17 ) ignored, dummy or non-object target
(      objects # 24 ) ignored, dummy or non-object target

(access.o #  26 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(command.o #  28 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(expr.o #  32 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(findfn.o #  34 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(format.o #  36 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(input.o #  38 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(main.o #  40 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(message.o #  42 ) missing dep on :  adb/machine.h

(opset.o #  44 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(output.o #  46 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(pcs.o #  48 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(print.o #  50 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h

(runpcs.o #  52 ) missing dep on :  adb/mac.h  adb/mode.h
                  adb/machine.h
```

```
(setup.o #  54 ) missing dep on :  adb/mac.h  adb/mode.h
                adb/machine.h

(sym.o #  56 ) missing dep on :  adb/mac.h  adb/mode.h
                adb/machine.h
```

Analysis and Conclusions of the Proceeding
Makec Checking Output

EVENT : Fifteen messages show that there are missing dependencies for fifteen object file targets.

CAUSE : Makefile "adb/makefile" has incomplete file dependency description.

DESCRIPTION : Most source files in the directory "adb" include data files "mac.h", "mode.h", and "machine.h", but the makefile "adb/makefile" does not specify these dependency relationships.

Output of Applying Makec to Check "as/makefile"

```
MAIN TARGET dependency necessity checking ::::::::::
        OK

MAIN TARGET dependency completeness checking :::::::
        OK

OBJECT FILE TARGETS' dependency checking  ::::::::::
(      cmp # 10 ) ignored, dummy or non-object target
(       cp # 14 ) ignored, dummy or non-object target
        No error on object file targets dependency
```

Analysis and Conclusions of the Proceeding
Makec Checking Output

Makefile "as/makefile" has correct file dependency description.

Output of Applying Makec to Check "awk/makefile"

```
ERROR(add_dep)-awk.h included in file(token.c) is missing

ERROR(add_dep)-awk.h included in file(awk.lx.l) is missing

ERROR(add_dep)-awk.h included in file(proc.c) is missing

ERROR(add_dep)-awk.h included in file(parse.c) is missing

ERROR(add_dep)-awk.h included in file(main.c) is missing

ERROR(add_dep)-awk.h included in file(lib.c) is missing

ERROR(add_dep)-awk.h included in file(tran.c) is missing

ERROR(add_dep)-awk.h included in file(b.c) is missing

ERROR(add_dep)-awk.h included in file(run.c) is missing


MAIN TARGET dependency necessity checking ::::::::::
        TARGET (awk  # 21) has unnecessary dependent :
                proctab.o

MAIN TARGET dependency completeness checking :::::::

        TARGET(           all #  6 )
        TARGET(           awk # 21 )
        miss dependent :  proc.o

OBJECT FILE TARGETS' dependency checking  ::::::::::
(        cp #  8 ) ignored, dummy or non-object target
(       cmp # 12 ) ignored, dummy or non-object target
(   y.tab.h # 24 ) ignored, dummy or non-object target
(     awk.h # 26 ) ignored, dummy or non-object target
( proctab.o # 29 ) ignored, dummy or non-object target

(token.o) has unnecessary dep on :    (# 29 awk.def)
(   token.c # 31 ) ignored, dummy or non-object target
(       src # 35 ) ignored, dummy or non-object target
(   profile # 39 ) ignored, dummy or non-object target
(      find # 42 ) ignored, dummy or non-object target
(      list # 45 ) ignored, dummy or non-object target
(      lint # 48 ) ignored, dummy or non-object target
( proctab.c # 52 ) ignored, dummy or non-object target
(      proc # 54 ) ignored, dummy or non-object target
```

Analysis and Conclusions of the Proceeding Makec Checking Output

(1) EVENT : Messages show that "awk.h" is missing.

CAUSE : This is a deficiency of makec.

DESCRIPTION : Makec issues an error message for any file which is included by some files and is not present.
File "awk.h" is a data file which is included in some source files, and is not present. This file is created by applying the command "yacc -d", and is removed after the project "awk" is completed because it can be created again easily.

IMPROVEMENT : Setting up a file name rule can resolve this deficiency.
This rule requires a data file named "xxx.h" to be the data file generated by applying the command "yacc -d" to a yacc grammar file "xxx.y" which bears the same file name and with file name extension ".y".

(2) EVENT : Some messages show that the main target has an unnecessary dependent "proctab.o" and has a missing dependent "proc.o".

CAUSE : This is a deficiency of makec.

DESCRIPTION : File "proc.c" is used to generate the executable module "proc", and the executable module "proc" is used to generate "proctab.c". Makec does not check command statements in makefiles and cannot handle this situation.

IMPROVEMENT : It cannot be improved.

(3) EVENT : A message shows that the file "token.o" has an unnecessary dependent "awk.def".

CAUSE : This is an error of the makefile "awk/makefile".

DESCRIPTION : The source file "token.c" does not include the data file "awk.def", so the file "token.o" should not depend on file "awk.def".

Output of Applying Makec to check "c/makefile"

```
MAIN TARGET dependency necessity checking ::::::::::
        TARGET (cl  # 40) has unnecessary dependent :
                table.o

MAIN TARGET dependency completeness checking :::::::

        TARGET(            all #  9 )
        TARGET(             c0 # 29 )
        TARGET(             cl # 40 )
        miss dependent :   cvopt.o

OBJECT FILE TARGETS' dependency checking  ::::::::::
(         cmp # 11 ) ignored, dummy or non-object target
(          cp # 17 ) ignored, dummy or non-object target
(       clean # 25 ) ignored, dummy or non-object target

(clt.o) has unnecessary dep on :    (# 43 cl.h)
(     table.o # 50 ) ignored, dummy or non-object target
(       cvopt # 55 ) ignored, dummy or non-object target
(       print # 58 ) ignored, dummy or non-object target
```

Analysis and Conclusions of the Proceeding
Makec Checking Output

(1) EVENT : A message shows that the target "cl" has an unnecessary dependent "table.o".

CAUSE : A deficiency of makec.

DESCRIPTION : Project "c" has an assembly source file "table.s".
Makec ignores all the assembly files which have the name extension ".s", and wrongly issues this message.

IMPROVEMENT : Let makec handles assembly source files can correct this deficiency.

(2) EVENT : A message shows that the main target has a missing dependent "cvopt.o".

CAUSE : There are more than one executable module in the project directory "c".

DESCRIPTION : File "cvopt.c" is used to generate an executable file named "cvopt". Makec does not check command statements in makefiles and cannot handle this situation.

IMPROVEMENT : It cannot be improved.

(3) EVENT : A message shows that the file "clt.o" has an unnecessary dependent "cl.h".

CAUSE : An error of the makefile "c/makefile"

DESCRIPTION : The source file "clt.c" does not include the data file "cl.h", so the object file "clt.o" should not depend on "cl.h".

Output of Applying Makec to Check "cpp/makefile"

```
MAIN TARGET dependency necessity checking ::::::::::
        OK

MAIN TARGET dependency completeness checking :::::::

        TARGET(            all #  5 )
        TARGET(            cpp # 15 )
        miss dependent :   yylex.o

OBJECT FILE TARGETS' dependency checking  ::::::::::
(     cp #  7 ) ignored, dummy or non-object target
(    cmp # 11 ) ignored, dummy or non-object target
        No error on object file targets dependency
```

Analysis and Conclusions of the Proceeding
Makec Checking Output

EVENT : A message shows that the main target misses
a dependent named "yylex.o".

CAUSE : A deficiency of makec. The project directory
"cpp" contains a source file "yylex.c" which
is used as an included file.

DESCRIPTION : File "yylex.c" is a source file which is
included in file "cpy.y".
Makec requires that object files of all the
source files are dependents of the main target.
So, makec issues this missing dependency message.

IMPROVEMENT : Let makec accepts source files be included
in other files can resolve this problem.

Output of Applying Makec to Check "dc/makefile"

```
MAIN TARGET dependency necessity checking :::::::::
        TARGET(dc #13) has unnecessary dependent : dc.c
        TARGET(dc #13) has unnecessary dependent : dc.h

MAIN TARGET dependency completeness checking :::::::

        TARGET(            all #  3 )
        TARGET(             dc # 13 )
        miss dependent :   dc.o

OBJECT FILE TARGETS' dependency checking  :::::::::
(    cmp #  5 ) ignored, dummy or non-object target
(     cp #  9 ) ignored, dummy or non-object target
        No error on object file targets dependency
```

Analysis and Conclusions of the Proceeding
Makec Checking Output

EVENT : A message shows that the main target misses a dependent "dc.o".

CAUSE : A deficiency of makec.

DESCRIPTION : Makec requires that a project should depend on object files, but not depend on source files or data files.
Since project "dc" has only one source file, it is reasonable that its makefile's main target depends on the source file and the included data file.

IMPROVEMENT(1) : If this makefile be changed to the following statements, makec would accept it.

```
all  :  dc.o
        cc -n -s -O dc.c -o dc
dc.o :  dc.c  dc.h
```

IMPROVEMENT(2) : Makec can be modified to handle this kind of special project which has only one source file.

Output of Applying Makec to Check "eqn/makefile"

```
ERROR - e.def included in file(lex.c) is missing

ERROR - e.def included in file(text.c) is missing

ERROR - e.def included in file(shift.c) is missing

ERROR - e.def included in file(move.c) is missing

ERROR - e.def included in file(diacrit.c) is missing

ERROR - e.def included in file(integral.c) is missing

ERROR - e.def included in file(lookup.c) is missing

ERROR - e.def included in file(funny.c) is missing


MAIN TARGET dependency necessity checking :::::::::
        OK

MAIN TARGET dependency completeness checking :::::::
        OK

OBJECT FILE TARGETS' dependency checking  :::::::::
(       cp #  8 ) ignored, dummy or non-object target
(      cmp # 12 ) ignored, dummy or non-object target
(      e.c # 27 ) ignored, dummy or non-object target
(    e.def # 29 ) ignored, dummy or non-object target
(     list # 36 ) ignored, dummy or non-object target
(     gcos # 39 ) ignored, dummy or non-object target
(      src # 42 ) ignored, dummy or non-object target
(     lint # 46 ) ignored, dummy or non-object target
        No error on object file targets dependency
```

Analysis and Conclusions of the Proceeding Makec Checking Output

EVENT : Some messages show that the file named "e.def" is missing.

CAUSE(a) : A deficiency of makec. (same as event (1) in the output analysis of applying makec to check "awk/makefile")

CAUSE(b) : File name "e.def" does not follow the file name convention of make/makec systems.

IMPROVEMENT : A modification of makec (same as event(1) in the output analysis of applying makec to check "awk/makefile") and renaming file "e.def" as "e.d" can resolve this problem.

Output of Applying Makec to Check "f77/makefile"

```
ERROR - tokdefs included in file(lex.c) is missing


MAIN TARGET dependency necessity checking ::::::::::
        TARGET (f0  # 41) has unnecessary dependent
                : gram.o

MAIN TARGET dependency completeness checking :::::::

        TARGET(            all #  8 )
        TARGET(            f77 # 37 )
        TARGET(             f0 # 41 )
        TARGET(       fixasf77 # 46 )
        miss dependent :   malloc.o

OBJECT FILE TARGETS' dependency checking  ::::::::::
(      cp # 10 ) ignored, dummy or non-object target
(     cmp # 16 ) ignored, dummy or non-object target
(compiler # 31 #34) ignored, dummy or non-object target
(  gram.c # 49 ) ignored, dummy or non-object target
( tokdefs # 56 ) ignored, dummy or non-object target

(lex.o # 59 # 60 ) missing dep on :  f77/732.h

(732x.o) has unnecessary dep on : (# 60 defs.h)
                (# 60 ftypes.h)

(732x.o #  60 ) missing dep on :  f77/732.h

(732.o # 60  # 63 ) missing dep on :  f77/scj.h
               f77/732.h

(putdmr.o # 60  # 63 ) missing dep on :  f77/732.h

(put.o # 60 # 63 ) missing dep on :  f77/scj.h  f77/732.h

(error.o #  60 ) missing dep on :  f77/732.h

(misc.o #  60 ) missing dep on :  f77/732.h

(io.o #  60 #  64 ) missing dep on :  f77/732.h

(intr.o #  60 ) missing dep on :  f77/732.h

(exec.o #  60 ) missing dep on :  f77/732.h

(expr.o #  60 ) missing dep on :  f77/732.h

(data.o #  60 ) missing dep on :  f77/732.h
```

```
(equiv.o #  60 ) missing dep on :  f77/732.h

(proc.o #  60 ) missing dep on :  f77/732.h
(gram.o #  60 ) ignored, dummy or non-object target

(init.o #  60 ) missing dep on :  f77/732.h

(driver.o) has unnecessary dep on :    (# 61 defs.h)

(driver.o #  61 ) missing dep on :  f77/732.h
(    lint # 66 ) ignored, dummy or non-object target
( cleanup # 74 ) ignored, dummy or non-object target
```

Analysis and Conclusions on the Proceeding Makec Checking Output

(1) EVENT : A message shows that the file "tokdef" is misssing.

CAUSE : File "tokdef" is generated by special method.

DESCRIPTION : File "tokdef" is created from file "tokens" by applying commands "grep" and "sed". Since makec ignores command statements in the makefile, this situation cannot be handled.

IMPROVEMENT : It cannot be improved.

(2) EVENT : A message shows that the main target has an unnecessary dependent "gram.o".

CAUSE : File "gram.c" is generated by special method.

DESCRIPTION : File "gram.c" is created from five grammar files by applying commands "sed" and "cat". Makec ignores shell commands in the makefile, and cannot handle this situation.

IMPROVEMENT : It cannot be improved.

(3) EVENT : A message shows that the main target misses a dependent "malloc.o".

CAUSE : File "malloc.c" does not belong to project "f77" but is kept in the project directory "f77".

DESCRIPTION : File "malloc.c" is not used by project "f77". Makec requires that every source file in the project directory must be a member of the project. So, makec issues the missing dependency message.

(4) EVENT : Several messages say that there are missing dependencies and unnecessary dependencies on some object files.

CAUSE : Errors of makefile "f77/makefile".

DESCRIPTION : File "f77/makefile" describes several unnecessary file dependency relationships, and misses some file dependencies too.
For example, files "732.o" and "put.o" should depend on files "scj.h" and "732.h", but the makefile "f77/makefile" does not mark these dependencies.
The makefile misses other 12 files' dependency on file "732.h".
Besides, the file "732x.o" should not depend on files "defs.h" and "ftypes.h", and should depend on file "732.h".
File "driver.o" should not depend on file "defs.h" and should depend on "732.h".

APPENDIX E

MAKEC ROUTINES DESCRIPTION

add_dir	- Add all the source file names of a directory into fl_inf.
add_fl	- Add a file into fl_inf.
chg_type	- Change a file name from type '.y' or '.l' to '.c'.
chk_ldep	- Check the dependents of one file.
chk_mdep	- Check the dependents of the main target.
copys	- Copy the input parameter to an allocated space.
cut	- Cut specified lengthed characters from a directory path.
cut_2	- Cut two levels in a directory path.
dump	- Dump the data structures.
dump_s	- Dump the dependents of a single file.
eqsign	- If the input parameter has an equal sign.
gather_dep	- Gather all the dependents of a file.
get_incl	- Get inclusion statement from a file.
hashloc	- A hashing function.
hasslash	- If the input parameter has slash.
init	- Initialize variables.
init_stk	- Initialize the stack which is used to keep directory names.
main_unnec	- Print some message for main target unnecessary dependents.
makename	- Make a nameblock, and creat a pointer from hashing table to it.
nextlin	- Get in a complete line.
p_name	- Print out a name and keep it indented in fields.
pop	- Pop a directory name from the stack.
proc_chk	- Check the file dependency description of a makefile.
proc_cmd	- Process the command line.
proc_dep	- Process dependencies. Find out dependencies among files.
proc_fl	- Process source files. Put them into fl_inf.
proc_put	- Print out file dependencies following the makefile format.
push	- Push a directory name into the stack.
put_cmd	- Print out the required command to compile the input parameter.
put_cnti	- Print out the continuation character, and indent on the next line.
put_full	- Print out the full path name.
put_obj	- Print out the object file name of the input parameter.
reseq	- Resequence the input parameters.
setvar	- Set up varblock for a macro definition.
shorten	- Shorten the directory path to make it neat.
source	- Check if the input parameter is a source file name.
srch_fl_in	- Search fl_inf for a file using index.
srch_src	- Search fl_inf for a file whose object file name is the parameter.

srchname	-	Search the parameter in make hashing table.
subst	-	Substitute macros in the first parameter and keep the result in the second parameter.
varptr	-	If the input parameter has not been kept in varblock link list, add it in.
yyparse	-	Parse a makefile.

2

VITA

Sou-Yen Yeh

Candidate for the Degree of

Master of Science

Thesis: AN EXTENSION TO THE UNIX MAKE COMMAND TO SUPPORT CREATING AND CHECKING MAKEFILES

Major Field: Computing and Infomation Science

Biographical:

Personal Data: Born in Taiwan, R.O.C., March 26, 1954, The son of Lai-Shing and Bi-Yen Yeh

Education: Graduated from National Normal University High School, Taiwan, R.O.C., in May, 1972; received the Bachelor of Science degree in Physics from National Central University, Taiwan, R.O.C., in May, 1977; completed requirements for the Master of Science degree at Oklahoma State University in May, 1986.

Professional Experience: Teaching assistant at Chinese Maritime College, Taiwan, R.O.C., September 1979-July 1980; patent engineer at Tai-E Patent and Law Office, Taiwan, R.O.C., August 1980-April 1981; graduate assistant at Oklahoma State University, Computing and Information Sciences Department, August 1983-June 1985.