



Analysis and Design of Algorithms Laboratory

A mini project report

on

MAKE TOOL

Submitted by

1PI11CS137, Rakshitha K Bhat

1PI11CS138, Rashmi Raghunandan

1PI11CS151, Sanjana S

Guided by

Professor N S Kumar



Department of Computer Science & Engineering

PES INSTITUTE OF TECHNOLOGY

(An Autonomous Institute under VTU Belgaum)

100 Feet Road, BSK III Stage, Hosakerehalli, Bengaluru - 560 085



Analysis and Design of Algorithms Laboratory

A mini project report

on

MAKE TOOL

Submitted by

1PI11CS137, Rakshitha K Bhat

1PI11CS138, Rashmi Raghunandan

1PI11CS151, Sanjana S

Guided by

Professor N S Kumar



Department of Computer Science & Engineering

PES INSTITUTE OF TECHNOLOGY

(An Autonomous Institute under VTU Belgaum)

100 Feet Road, BSK III Stage, Hosakerehalli, Bengaluru - 560 085



PES INSTITUTE OF TECHNOLOGY

(An Autonomous Institute under VTU Belgaum)

100 Feet Road, BSK III Stage, Hosakerehalli, Bengaluru - 560 085

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the mini project entitled “**Make Tool**” has been carried out by

1PI11CS137, Rakshitha K Bhat

1PI11CS138, Rashmi Raghunandan

1PI11CS151, Sanjana S

in the partial fulfillment of fourth semester Analysis and Design of Algorithms Laboratory

[11CS255]

Signature of the
Lab Incharge

Signature of the
Head - CSE

Name & Signature of the Examiners:

Examiner 1:

Examiner 2:

ACKNOWLEDGEMENTS

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible, and whose constant guidance and encouragement helped us in completing the project successfully. We consider it a privilege to express gratitude and respect to all those who guided us throughout the course of the completion of the project.

We would like to express our heartfelt thanks to **Prof. M. R. Doreswamy**, PESIT founder, **Prof. D. Jawahar**, CEO and **Dr. K. N. Balasubramanya Murthy**, Principal, for providing us with a congenial environment for carrying out the project.

We express our gratitude to **Prof. Nitin V. Pujari**, Head of the Department, Computer Science, PESIT, whose guidance and support has been invaluable and for including the project as part of the course.

We extend our sincere thanks to our mentor **Prof. N S Kumar** for his constant guidance, encouragement, support and invaluable advice without which this project would not have been completed.

Last, but not the least, we would like to thank our friends whose invaluable feedback helped us to improve the software by leaps and bounces, and our parents for their unending encouragement and support.

ABSTRACT

TITLE: Make Tool

AIM: To create and implement a Make Tool.

DESCRIPTION OF THE UTILITY:

The man page of the Make Tool defines the tool as “a utility which determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them.”

The Make Tool uses a Make File which contains all the information about the dependencies between files and the commands to be executed when any of the files undergoes a change. When the Make Tool is run, it identifies those files which have changed and their corresponding dependent files and executes the commands associated with them.

IMPLEMENTATION:

The Make Tool is implemented in three stages which are interdependent on each other. The first stage is string tokenisation and parsing of the text in the Make File. The second stage is creating a binary tree to represent the dependencies between files. The third stage is checking the timestamps of the files against a default value to identify which files have changed and execute the appropriate command associated with the file.

Implementation of these three stages is done using various algorithms for parsing, tree creation and traversal.

SOFTWARE REQUIREMENTS: ANSI C, Ubuntu 8.04 or higher, C compiler, Windows XP or higher, Ubuntu terminal

HARDWARE REQUIREMENTS: 32-bit processor, Monitor, Mouse, keyboard

CONTENTS

Acknowledgements.....	1
Abstract.....	2
Introduction.....	4
Design Details.....	5
Implementation.....	7
Test Cases and Results.....	12
Conclusion.....	14
Bibliography.....	15

INTRODUCTION

An algorithm is a set of rules or a step-by-step procedure to be followed in problem-solving operations. Selecting the most appropriate algorithm, depending on its efficiency is an important step in developing any application.

The Make Tool uses algorithms for binary tree creation and traversal as well as certain searching algorithms.

Basic Information about the Make Tool:

A Make Tool is a program that determines which parts of a large program need to be recompiled, and issues the commands to recompile them.

- In an application containing a very large number of files, keeping track of all the changed files, dependent files and the commands to recompile them is an arduous task for a programmer.
- This can be avoided by using a Make Tool.
- A Make File contains a list of all the files in the program as well as information about which file is dependent on which. It also contains the commands for compilation, object file creation or other system commands that are to be issued in case of a change in any of the corresponding files.
- The Make Tool uses the information in the Make File to identify the files to be recompiled when any of the files has been changed and to execute the associated commands.
- After any file has been changed, the user can run the Make Tool by using the command: `make -f <makefile name>`
- This will automatically compile all the changed programs and other system commands if necessary.

DESIGN DETAILS

The Make Tool has been designed in three stages:

Stage 1: Extracting data from the Make File (Parsing)

The Make File contains all the details of dependencies and commands to be executed. Hence, data from this file is absolutely necessary for the tool to function correctly. Data extraction involves the following steps:

- **Reading the Make File:** The file is read line by line.
- **Tokenisation:** The file names mentioned in each line are to be extracted. This is done by first setting the delimiter as “:” and next as a blank space.
- **Storing the filenames in a data structure:** The data structure is similar to a list. Each node in this structure has three pointers: **left**, **right** and **down**.
 - The **right** pointer points to a linked list of all the files that the main node is dependent on.
 - The **left** pointer points to a linked list of all the commands associated with the main node.
 - The **down** pointer points to the name of the file on the left-hand side of the colon (“:”) in the next line of the Make File.

This data structure created in this stage is then passed on to the next stage.

Stage 2: Creating the binary tree to represent dependencies and the files.

In this stage, the binary tree that holds the complete structure including all the dependencies is created using the following steps.

- The main nodes of the tree are identified from the nodes attached to the **down** pointers of the structure of the previous stage. These nodes are linearly added to the right of the root node of the binary tree.
- The files that the main nodes depend on are identified as the files in the linked list that the **right** pointer of each node in the structure of the previous stage points to. These filenames are added as the left subtree of each of the main nodes.
- Thus, a binary tree is created such that all the files that a particular file depends form the left subtree of the node that the file is represented by. All nodes to its right are independent of it.
- The commands associated with each node are added as a singly linked list to the node. This linked list is an attribute of the node structure created.

This results in the formation of a binary tree which is passed on to the next stage.

Stage 3: Comparing the timestamps to identify the changed files.

The first time the Make Tool is run, since none of the files have been executed, all the commands for compilation are executed. After this initial run, the timestamps of all the files are reset to a default value. Subsequent calls to Make follow these steps to check for the correct command to be executed:

- A pointer is assigned to the first node on the right of the root node of the binary tree passed from the previous stage. This node is marked as the **parent** node of the subtree

that it is the root of.

- The left subtree of this parent node is traversed using the postorder traversal algorithm. For each node that is visited, the timestamp of the file whose name is contained in the node is checked against the default value assigned during the first execution. The timestamp of the file is found using the **stat** function.
- If the timestamp of any of the files in the left subtree is different from the default value, then the **parent** node of that left subtree needs to be recompiled and the commands associated with it need to be executed.
- This execution of commands is done using the **system** function.
- Once this is done, the program moves to the next node to the right of the **parent** and repeats the process for the left subtree associated with that node. This is repeated till all such left subtrees have been checked for changed files.
- Once the right subtree of the whole binary tree has been checked, the left subtree of the root node is traversed to check if any of the files contained in it have changed using the same timestamp checking procedure. If there are any changes, then the commands associated with the root node of the binary tree need to be executed.

The output of this stage is that all the commands associated with changed files are executed and also displayed to the user.

IMPLEMENTATION

The main functions used in the program and the stage in which they are used are:

Stage 1:

void read(FILE *)

The read() method is a function which takes in FILE pointer as a parameter. The return type is void.

This function is used to read the contents of the Make File line by line using the **fopen()**, **fputs()**, **fgets()** and **fclose()** library functions. After reading each line, it calls the tokenize function which tokenizes the string using the delimiter ":".

void tokenize(char *):

The tokenize() function is takes in a character array(i.e a string). The return type is void.

After each line is read from the Make File, this function is called. The line that has been read is passed as a parameter to this function. It makes use of the **strtok()** function included in the **string.h** header file. The **strtok()** function parses a string into a sequence of tokens delimited by the ":" operator. Each of these tokens are further subjected to another round of tokenization using the delimiter " "(blank space) by calling the **tokenline()** function. The **tokenline()** function has similar implementation as the **tokenize()** function with the only difference being that the former tokenizes strings using " "(blank space) as a delimiter whereas the latter tokenizes the string using ":" as the delimiter.

void endins(char *):

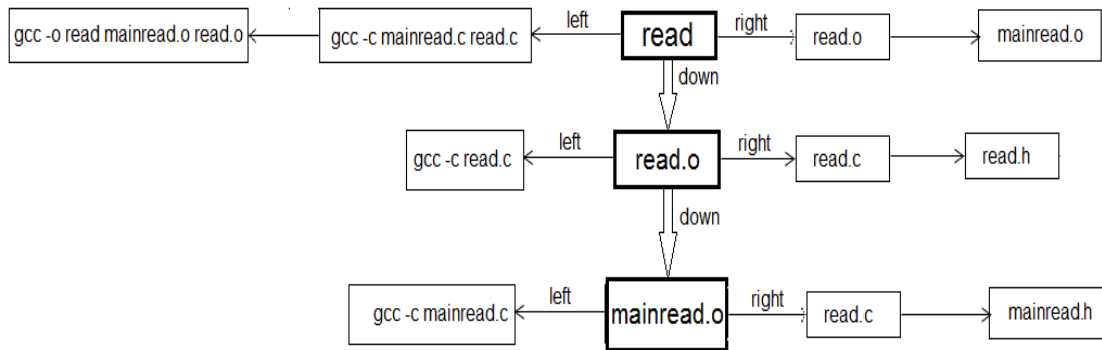
The endins() function takes in a character array (i. e. a string). The return type is void.

This function inserts the given string into a data struture which is similar to a list. The data structure used has 3 pointers left, right and down. The right pointer points to a list which contains all the names of the dependent files. The left pointer points to a list which contains all the commands to be executed when a change occurs to any of the dependent files present on its right hand side. The down pointer points to a list of file names which have dependent files associated with it.

Consider the example Makefile:

```
read: read.o mainread.o
    gcc -c mainread.c read.c
    gcc -o read mainread.o read.o
read.o: read.c read.h
    gcc -c read.c
mainread.o: mainread.c read.h
    gcc -c mainread.c
```

The data structure for this file, generated by this stage, would be:



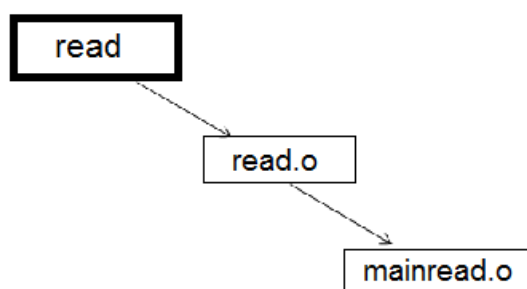
Stage 2:

void createRight():

This method takes no parameters as input. The return type is void.

This method is used to create the right part of the tree structure. The contents of the nodes of the right subtree of the binary tree are obtained from the **down** pointers of the parsed data structure from the previous stage. These form the main nodes and are called the **dependents**. They represent the files that have some dependencies.

For the example Makefile given above, the right side of the tree with **read** being the root would be:



*struct treest *createDependent(struct treest *, struct node *):*

This function takes two input parameters – a treest pointer and a node pointer. The return type is a pointer of type treest.

The nodes created in the previous method are dependent on other files. These files are attached as a tree to the left of the nodes of the previous method, thus completing the right subtree for the binary tree.

*struct treest *expandifPresent(struct treest *, struct node *, int):*

This function takes three input parameters – a treest pointer, a node pointer and an integer. The return type is a pointer of type treest.

In some cases more than one file depends on the same file. Thus, this node needs to be present in the left subtree of both the nodes. For this we make use of this function. Every time a given node is encountered, we traverse the parsed data structure to check if this node is already present. If it is, then instead of adding that node, we add the dependents of this new node to the left subtree.

struct node *findPos(struct treest *q):

This function takes a treest pointer as an input parameter. It returns a pointer to a node.

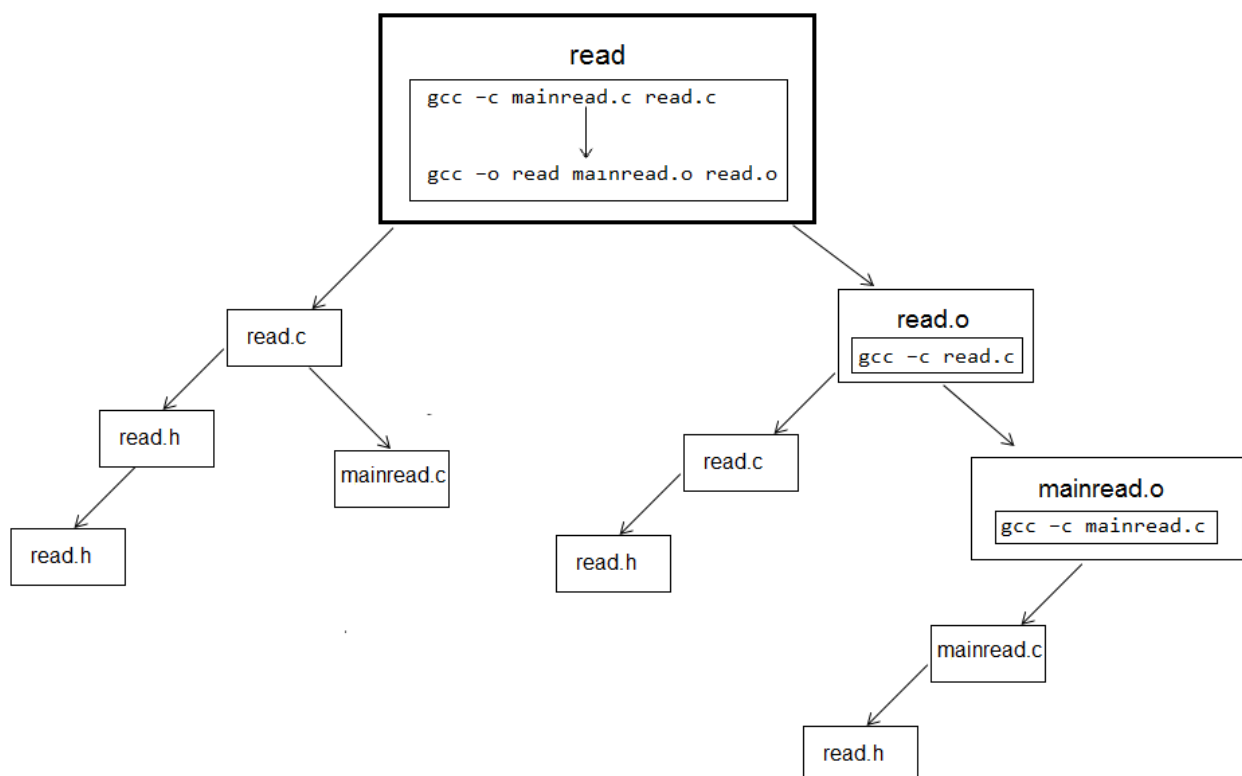
This is used to find if a given node has other dependents.

void addCommand()

This function has no input parameters. The return type is void.

Every dependent in the make file may be followed by the commands required to execute it. For this a singly linked list of all the commands is attached to each main node of the tree. This linked list is an element in the node structure.

The final binary tree for the Makefile given as an example above is:



This binary tree is passed on to the next stage where the timestamp comparison takes place in order to find out which files have been changed.

Stage 3:

void setTime(struct treest*)

The setTime() function takes a pointer of type struct treest as the input parameter. The return type is void.

This function sets the time of last modification of all the files in the binary tree passed on from the previous stage to a default value (this program sets this value as 1000). This is done by traversing the tree using postorder algorithm and setting the timestamp using the **utimbuf** structure and the **utime()** and **stat()** functions which are all library functions. This function is called at the end of the Make Tool program.

char* postorder(struct treest* , struct treest * , int)

The postorder() function takes two pointers of type struct treest and one integer as the input parameters. The return type is a pointer to a character array. (i.e a string).

In this function, one of the pointers to the struct treest passed as an input parameter is one of the nodes to the right of the root node of the tree. This is called the parent node. The other pointer is the first node of the left subtree having this parent node as the root. The function is implemented until all such left subtrees on the right side of the main binary tree have been traversed. In each traversal, the timestamps of the files whose names are stored in each node are checked against the default value set in the above function using the **stat()** function. If any of the nodes are found to have a different timestamp, the commands associated with the parent node are executed. The function returns the name of the file contained in the parent node if any of the files in a left subtree have been changed.

int checkLeft(struct treest * , int)

This function takes a pointer of type struct treest and one integer as the input parameters. The return type is an integer.

This function checks if any of the nodes of the left subtree of the root of the main binary tree have been changed. A change in any of these files will lead to the execution of all the commands associated with the root node. The procedure for timestamp checking is the same as in the **postorder()** function. The function returns 1 if any of the files have been changed.

void toCall()

This function has no input parameters. The return type is void.

This function calls the other three functions. If any of the root nodes of the subtrees of the main binary trees needs to be recompiled, the commands associated with it are executed using the **execute()** function.

void execute(char *)

This function takes a pointer to an array of characters as the input. The return type is void.

This function finds the node with information being equal to the filename being passed as an input parameter. If the node is found and it has certain commands associated with it, those commands are executed using the **system()** function which is a library function. The tree is traversed to the right until such a node is found.

TEST CASES AND RESULTS

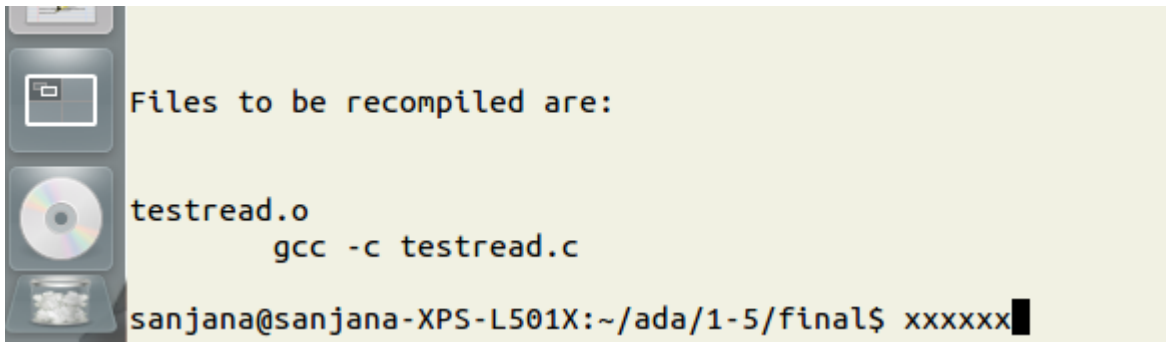
The Make File:



```
sanjana@sanjana-XPS-L501X: ~/ada/1-5/final
read: read.o mainread.o
    gcc -c mainread.c read.c
    gcc -o read mainread.o read.o
read.o: read.c read.h
    gcc -c read.c
mainread.o: mainread.c read.h
    gcc -c mainread.c
testread.o: testread.c
    gcc -c testread.c
```

Test case 1: Changing file testread.c should generate testread.o but not execute any of the other commands since none of the other files are dependent on testread.c

Expected result: Only testread.o is generated



```
Files to be recompiled are:
testread.o
    gcc -c testread.c
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ xxxxxx
```

As expected, only testread.o is to be changed and the command associated with it is executed.

Test case 2: Changing file read.h will execute the commands associated with read.o, mainread.o and read.

Expected result: read.o, mainread.o and the executable read are generated.

```
sanjana@sanjana-XPS-L501X: ~/ada/1-5/final
sanjana@sanjana-XPS-L501X:~$ cd ada/1-5/final
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ ls
compare.c  hard.c  mainread.c  rashmain.c  read.c~  testread.o  treeform.h  treemain.c  treest.c  trial1~
compare.c~ hello.c  mainread.o  rashmain.o  read.h  trail.txt  treeform.h.gch  treest 2.c  treest.c~ trialmake.txt
compare.o  main.c  rash  read  read.o  trail.txt~ tree.h  treest2.c  treest.o  trialmake.txt~
final      main.o  rash2  read.c  testread.c  tree.c  tree.h.gch  treest2.c~ trial1  trial.mk
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ unlink read
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ unlink read.o
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ unlink mainread.o
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ vi trial.mk
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ ls
compare.c  hard.c  mainread.c  rashmain.c  testread.c  tree.c  tree.h.gch  treest2.c~ trial1  trial.mk
compare.c~ hello.c  rash  read.c  testread.o  treeform.h  treemain.c  treest.c  trial1~ trial.mk~
compare.o  main.c  rash2  read.c~ trail.txt  treeform.h.gch  treest 2.c  treest.c~ trialmake.txt  waste.c
final      main.o  rashmain.c  read.h  trail.txt~ tree.h  treest2.c  treest.o  trialmake.txt~
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$
```

Initially, it is seen that testread.o is present but read.o, mainread.o and the executable read are not.

Then, we run the Make tool after changing read.h

```
Files to be recompiled are:
read.o
gcc -c read.c
mainread.o
gcc -c mainread.c
read
gcc -c mainread.c read.c
gcc -o read mainread.o read.o
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$ ls
compare.c  hard.c  mainread.c  rashmain.c  read.c~  testread.o  treeform.h  treemain.c  treest.c  trial1~  trial.mk~
compare.c~ hello.c  mainread.o  rashmain.o  read.h  trail.txt  treeform.h.gch  treest 2.c  treest.c~ trialmake.txt  waste.c
compare.o  main.c  rash  read  read.o  trail.txt~ tree.h  treest2.c  treest.o  trialmake.txt~
final      main.o  rash2  read.c  testread.c  tree.c  tree.h.gch  treest2.c~ trial1  trial.mk
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$
```

As expected, the commands associated with read.o and mainread.o and read are executed. The executable file, **read**, is also created.

Test case 3: When no files are changed

Expected result: None of the commands are to be executed.

```
Files to be recompiled are:
sanjana@sanjana-XPS-L501X:~/ada/1-5/final$
```

As expected no commands to be executed are printed.

CONCLUSION

The Make Tool was implemented successfully. The use of algorithms is evident in the application.

BIBLIOGRAPHY

1. GNU Makefile
2. ANSI C by Ritchie and Kernighan
3. MAN pages