CPE 113

Project

Group Dayzers

Make file Generator

57070503454 Prathiphan Dubey (Lucky)

57070503483 Saket Khandelwal

Table of Contents

* Abstract
* Architecture diagram and explanation
* diagrams of major data structure
* Flowcharts for all major algorithms
* Top level flowchart
* Source Code

**Make file Generator**: Abstract

A c program or any software project may have multiple source files and multiple executable programs. In order to compile and create a main executable file, it's great to have a make file to compile and link some or all of the programs.

In our software, we have created an automated facility (Makefile generator) which will read a directory given by the user and scan all their code and generate a Makefile which will be created in the directory with their files. The user will just have to enter the program, and tell the program the directory, the program will automatically generate a Makefile and it can be found in the users given directory.

**Makefile Generator**: Architecture diagram

User Interface

main.c

Generator.c

Str\_list.c

String\_hash.c

function\_scanner.c

Binary\_search\_tree.c

src\_parser.c

* **Makefile Generator**: Data structure documentation

|  |  |
| --- | --- |
| **Component** | **Functionalities** |
| main.c | * Display user choice menu * Display user guide * Let user choose source directory to scan * Check source directory existence * Invoke generator |
| generator.c | * Scan source files in the given directory * Invoke C source parser for each source file found * Search for "main" source files * Build dependencies list of each "main" source file found * Build makefile entry for each "main" source file and its dependencies * Build the whole makefile |
| src\_parser.c | Parse C source files, tokenize them |
| function\_scanner.c | Find function references and function implementations in each source file based on its token list (token list returned from src\_parser.c) |
| str\_list.c | * Represent the linked-list of string, each element (node) in list is a zero-terminated string of characters * Allow inserting new node to the head or tail of the given linked-list * Allow freeing a whole linked-list |
| string\_hash.c | * Represent the hash table for pairs of key-value. Keys are zero-terminated string of characters, values are arbitrary data types * Allow inserting a new pair of key-value to the given hash table * Allow updating a new value to the given key entry in the given hash table * Allow searching for a specified key in he given hash table |
| binary\_search\_tree.c | * Represent the binary search tree containing pairs of key-value. Keys are zero-terminated string of characters, values are arbitrary data types. * Allow searching for the specified key in the given tree * Allow inserting a new pair of key-value to the given tree |

**String list**

String list is a linked-list of character string. The string list is implemented in the source named "str\_list.c" and has structure described below:



**Binary Search Tree (BST)**

We implement a simple BST for fast searching for a given character string. The BST data structure is implemented in the source file named "binary\_search\_tree.c". Each node in the BST has the structure described below:



Each node has a key that points to a character strings. Each node may have a "greater" node that contains a "greater" key and a "lesser" node that contains a "lesser" key. The terms "greater" and "lesser" refer to the string comparison based on the ASCII table.

**String hash table**

The string hash table is used to store pairs of key-value. Each key is a character string. The string hash table is implemented in the source file named "string\_hash.c". The hash table has 256 entries, each entry contains a linked-list of strings that have the same hash value. The data structure is described in the diagram below:



**Source file dependencies**

**Algorithm**

Let's consider the example on the next page:

We have 5 source files named: source0.c, source1.c, source2.c, source3.c and source4.c:

|  |
| --- |
| // File source0.c  // Implement the function main  int main()  {  ...  fn\_1\_1(); // Reference to function fn\_1\_1  ...  }  // Implement the function test  void test()  {  ...  } |
| // File source1.c  // Implement the function fn\_1\_1  void fn\_1\_1()  {  ...  fn\_2\_1(); // Reference to function fn\_2\_1  fn\_2\_2(); // Reference to function fn\_2\_1  fn\_3\_1(); // Reference to function fn\_3\_1  ...  } |
| // File source2.c  // Implement the function fn\_2\_1  void fn\_2\_1()  {  ...  }  // Implement the function fn\_2\_2  void fn\_2\_2()  {  ...  } |
| // File source3.c  // Implement the function fn\_3\_1  void fn\_3\_1()  {  ...  test(); // Reference to function test  ...  } |
| // File source4.c  // Implement the function fn\_4\_1  void fn\_4\_1()  {  ...  } |

The dependency relations is shown in the diagram below:



We start creating the Makefile based on the "main" source files. The "main" source file is the source file that implements the "main" function

* Step 1: we locate the "main" source file: it's source0.c We add source0.c to dependency list.
* Step 2: From source0.c, we get all functions that this source file has reference to, it's "fn\_1\_1".
* Step 3: We search all 5 source files for file that implements the function "fn\_1\_1", its source1.c. We add source1.c to dependency list.
* Step 4: We search all functions that source1.c has reference to, they are "fn\_2\_1", "fn\_2\_2" and "fn\_3\_1".
* Step 5: We search all 5 source files for files that implement functions "fn\_2\_1", "fn\_2\_2" and "fn\_3\_1". They are source2.c and source3.c. Source2.c has no function reference. We add source2.c and source3.c to dependency list.
* Step 6: Source3.c has reference to function "test", which is implemented in source0.c. Since source0.c is already in dependency list, there is no more source file to be checked.
* Then we have dependency list: source0.c, source1.c, source2.c, source3.c
* Source4.c is not in the dependency list.

**Data Structures**

We have to represent 2 relations:

* For each function, which file implements it.
* For each source file, which functions does it reference to.

The 2 relations in the example above is shown in the tables below:

|  |  |
| --- | --- |
| Function name | Implemented by source file |
| main | source0.c |
| test | source0.c |
| fn\_1\_1 | source1.c |
| fn\_2\_1 | source2.c |
| fn\_2\_2 | source2.c |
| fn\_3\_1 | source3.c |
| fn\_4\_1 | source4.c |

|  |  |
| --- | --- |
| Source file | Functions that source file has reference to |
| source0.c | fn\_1\_1 |
| source1.c | fn\_2\_1  fn\_2\_2  fn\_3\_1 |
| source2.c |  |
| source3.c | test |
| source4.c |  |

We use 2 binary search trees (BSTs) for those 2 relations. All that data are store in the generator internal data:



* fnTree: a BST representing the function names and the source files that implement them.
* srcTree: a BST representing the sources files and functions that they reference to.
* mainSrcFile: a string list representing list of main source files.

fnTree has structure described in the example below:



srcTree has structure described in the example below:



* **Makefile Generator**: Flowcharts for all major algorithms.

## Source file parsing flowchart



## Source code tokenization flowchart



## Makefile generator flowchart



## Dependencies flowchart



* **Makefile Generator**: Top Level Flowchart



**Source**

**Code**