Institute of Technology, Carlow

B.Sc. in Software Engineering

CW228

Project Report

C Maintenance Tool

Name: Anna-Christina Friedrich

ID: C00132716

Supervisor: Dr. Christophe Meudec

Submission Date: 16.04.2010

Table of Contents

1. Introduction

This document is the final report about my project 'CMT' required for 4th year B. Sc. In Software Engineering.

CMT stands for C Maintenance Tool. This tool is supposed to support studying unknown C source files. Above all it is useful because there are plenty of programs available written in C and those have to be maintained. CMT uses a grammar file to parse the input C file. While parsing all identifiers are stored inside a tree data structure taking the scope of each identifier into account, i.e. each identifier definition is stored with all its occurrences. This data structure is the basement for each following action. It is possible to simply create a text file including a list of all identifiers, separated into variables and functions. Furthermore there is a possibility to create interactive Html files. Those files display the source code of the C file and offer to highlight the next or previous occurrence of a chosen identifier. According to refactoring there is a renaming facility, which allows to replace all occurrences of an identifier by a new name. For the purpose of testing the possibility of renaming all identifiers automatically was implemented.

In the following sections I would like to enlarge upon the achievements during developing the tool. Of course no project works as expected and there are always problems occurring that you did not respect during the planning phase. Therefore all major challenges which all together lead to a delay of the project are elaborated. Afterwards I try to figure out what I learned by doing this project according to software engineering knowledge as well as soft skills. Finally I will describe what I would have done in another way if I had started from scratch and I try to give advice if someone faces such a project.

2. Achievements

There are mainly two of the previous documents which are concerned about difficulties of the project and its execution. The research manual includes first considerations about challenges that need to be solved. The project plan explains a distinction into versions that are supposed to be finished at certain dates.

First of all the scope problem was elaborated. Each identifier has its scope in which it is valid. If there are two different identifiers having the same name the second declaration overwrites the previous one, so that each occurrence inside the scope(and inside following scopes) of the second identifier belongs to the declaration of the second identifier. In order to store all occurrences of identifiers correctly linked to their declaration regarding the name problem, it was necessary to find a proper solution. A detailed explanation is given in sections 3.2 and 3.4. During implementation of the scope problem more and more challenges occurred which led to a first delay of the project that will be explained later on in this section. After testing several self written C files including various scopes and identifiers having the same name and some random files from the internet, I am confident that nearly each identifier is stored correctly with its occurrences. Unfortunately there is an exception according to structures. Generally if you want to insert an occurrence, its declaration is searched in each valid scope backwards, so that you can avoid the mentioned name problem. But if there is an occurrence of an attribute of a structure(always after 'dot' and 'arrow') it is not possible to use this algorithm of finding the declaration. This problem is further explained in section 3.9 and led to a further delay of the project.

The second main problem refers to the C preprocessor. Preprocessing commands can be divided into three groups: file inclusion, macro definition and conditional directives. I underestimated the issue of file inclusion as I stated in the research manual: “File inclusion isn’t as difficult as macros and conditional directives, because #include <file name> expands the scope of refactoring, i.e. a further file has to be refactored.” This issue turned out to be a bit more difficult than estimated. First of all the grammar needed to be extended, because there was originally no preprocessing rule inside the given C grammar. Thus I spent some days to figure out the right syntax to identify file inclusion. As this part was working there was no problem in referencing the included file if it is located in the same directory as the main C file. Another challenge was to implement the search for included files, if they are not located inside the current directory. The solution in my project is to search first in the current directory and all its subfolders and then going backwards and searching in previous folders step by step. In the end the tool is now able to handle file inclusion no matter where the file is located. If the tool cannot find the file an exception is caught and the user gets a “file not found” message.

Due to the previous mentioned problems which delayed the project it was not possible to insert macros and conditional directives into the grammar. Especially because you don't have to use legal C code inside macro definitions and conditional directives, therefore the given rules cannot be reused. Everything needs to be implemented specially for preprocessing. Although I was using a preprocessor grammar[1], it was not possible to include those rules into the C grammar. Unfortunately I could not consider the problems in macros mentioned in the research manual, like using a global variable inside a macro or the concatenation problem.

Nonetheless I was also able to implement output in a text file and in an interactive html file using JavaScript and CSS. The html file shows the source code. By hovering over an identifier a menu pops down and the user can choose between 'show next' and 'show previous'. By clicking on those the next/previous occurrence of the certain identifier is highlighted. The layout of the html file, especially the indentation of the source code can still be improved.

According to the project plan I figured out 3 versions of the tool:

CMT- Version 1:

- analyse source file: one C-File to be cross-referenced, no preprocessing, has to handle scope, just variables to be referenced
- output just as text file
- rename all: facility of renaming all identifiers for testing functionality of the tool

CMT- Version 2:

- analyse source file: has to handle several C-Files (preprocessing: #include), also function names to be referenced
- create html: output also as html file
- rename identifier: renaming of certain identifiers

CMT- Version 3:

- final version
- analyse source file: has to handle preprocessing: macros and conditional directives, renaming of those

Taking the achievements into account I managed to create version 1 and 2 of the expected versions. Because of the fact that some problems were underestimated and took more time than expected, the last version 3 could not be finished.

3. Challenges

3.1 Action Code

Problem:

At the very beginning of using the grammar file a certain place needed to be localised where to insert additional code. This additional code, called 'action code', is indicated by curly brackets in order to insert code into the grammar using a programming language; in this case Java was used. The first approach was to handle the storing of identifiers. An identifier is defined as a token as follows:

```
IDENTIFIER
      :     LETTER(LETTER|'0'..'9')*
            ;
```

Therefore the name was stored by getting the letters of one identifier. This solution worked fine in the beginning, but as the scope was introduced the first challenge considering the grammar occurred.

Whenever there is a left curly bracket occurring in a grammar rule, a new scope is valid. Scopes are represented by integers, initialised by zero, for now. Whenever there is a right curly bracket the scope becomes invalid. Following rule shows the introduction of a block statement:

```
compound_statement
      : '{' declaration* statement_list? '}'
      ;
```

Storing each identifier with its scope number was slightly challenging. Storing the identifiers name works whereas the scope number is always zero, although the scope number should have been increased with each left curly bracket. Hence a solution needs to be found, so that both is stored correctly, the name and the current scope number.

Solution:

The first obvious difference between the mentioned 'IDENTIFIER' and 'compound statement' is that identifier is written in upper case letters and compound statement in lower case letters. This is an important difference in Antlr, thus upper case define tokens, which are recognized by the lexical analysis(Lexer) of the source code, and lower case letters define rules, which are applied during the syntactical analysis(Parser). To detect the problem, one has to understand how lexer and parser influence each other. First of all the lexer simply breaks down the source code into a stream of tokens. This token stream is now input to the parser that recognizes the grammar rules and applies it to the tokens. This means that the lexer part is strictly separated from the parsing part, therefore it is obvious why the scope number is always zero for each identifier: while tokens are identified the scope is always in its state of initialisation, i.e. zero.

This issue causes the reinvestigation of the given grammar. It is clearly necessary to put both code snippets into either lexer or parser part of the grammar. Because of the fact that there is no specific token for curly brackets, the code to store identifier names has to be moved into the parsing part. Therefore each rule referring to the token 'IDENTIFIER' needs additional code to store this identifier. The rule, which is principally concerned with the declaration of identifier expressions is called 'direct declarator' and looks at this stage of elaboration as follows, including action code:

```
direct_declarator
        :   (   id=IDENTIFIER
            {     String d = $id.getText();
                        int line=$id.getLine();
                  int col=$id.getCharPositionInLine();
                  String source=input.getSourceName();
            CMT.idTree.insert(new
Identifier(d,line,col,source,CMT.scopenumber));
      }
        |      '(' declarator ')'
      )
        declarator_suffix*
        ;
```

A new identifier object is created and stored inside a tree data structure. The current scope number is a static variable, which is always increased and decreased according to curly brackets. Occurrences of identifiers are managed by a different rule, called 'primary expression':

```
primary_expression
        : id=IDENTIFIER
      {     String d = $id.getText();
                  int line=$id.getLine();
                  int col=$id.getCharPositionInLine();
                  String source=input.getSourceName();
                  Node<Identifier> temp = CMT.idTree.search(
                        new Identifier(d,line,col,source,CMT.scopenumber));
                  if(temp!=null)
                        temp.content.getOccurrences().insert(
                        new Occurrence(d,line,col,source,CMT.scopenumber));
            }
        | constant
        | '(' expression ')'
        ;
```

First the tree has to be searched for an identifier declaration of the same name. If one is found, the occurrence is stored inside another data structure in the identifier object.

This improvement to the code makes it possible to store all necessary information about an identifier correctly. Issues about realization of the scope and its representation can be found in the next section.

3.2 Scope

Problem:

As mentioned in the previous section, the first solution to handle the scope in a C file was to increase an integer variable with each left curly bracket and to decrease the variable with each right curly bracket. In order to check if the program works fine, several test files were derived. During this phase a problem occurred, which changed the solution of the scope basically. Consider one scope following another scope, e.g. one if-statement followed by a while-loop:

```
int main(void){
      if(condition){          }
      while(condition){ }
}
```

According to the first attempt, the red area has the scope number 0. The next scope, displayed in green, is scope number 1. Scope number 2 is associated with the yellow scope. Now the first right curly bracket occurs and the number is decreased, so that we are in scope 1 again. Now the problem appears in the blue scope. The scope number is increased to number 3, so that the yellow and the blue scope are having the same number: 2. But obviously both scopes are absolutely separated from each other. Hence there must be a solution which lets one identify each scope properly.

Solution:

Avoiding the problem that there are two different scopes with the same scope number, this number is always increased with a left curly bracket, but is not decreased with a right curly bracket. Now the issue of scopes which are invalid, i.e. right curly bracket occurred, needs to be solved. Thus the data structure 'Stack' is useful. The idea is to push with each left curly bracket the increased scope number onto the stack and to pop with each right curly bracket from the stack. This technique makes it possible to assign each scope a unique identification number and to represent at each stage all valid scopes, simply by looking at the scopes on the stack. The stack referring to the previous example should look as follows:

SCOPE	STACK
Inside red scope	$\begin{array}{\|c\|}\hline \\ \hline \\ \hline 0 \\ \hline\end{array}$
Inside green scope	$\begin{array}{\|c\|}\hline \\ \hline 1 \\ \hline 0 \\ \hline\end{array}$
Inside yellow scope	$\begin{array}{\|c\|}\hline 2 \\ \hline 1 \\ \hline 0 \\ \hline\end{array}$
Inside blue scope	$\begin{array}{\|c\|}\hline 3 \\ \hline 1 \\ \hline 0 \\ \hline\end{array}$

3.3 Distinction between Variable- and Function Identifier

Problem:

In terms of section 3.2, where action code to store identifiers is explained, there is no distinction between variable and function identifiers. Why must function and variable identifiers be treated different at all? In the C language one has at least to declare each function before the main function. Consider following code:

```
      void function( ); // this is the declaration of the function
      int main(void){
            ...
      }
      void function( ){ // this is the definition of the function
            ...
      }
```

The problem is that both, declaration and definition, are recognized as direct declarators. Consequently those are stored as two different identifiers, however it is important, especially for the renaming part, that those are stored as the same identifier. That is why variable and function identifiers have to be stored separately.

Solution:

Occurrences of function and variable identifiers can be stored equally, there is no need for any change. Therefore the focus is on the 'direct declarator' rule of the grammar. The distinction can be made as follows: if the next token after an identifier declaration is a left parenthesis, it is a function identifier, otherwise it is a variable identifier. The code needs to get a condition:

```
direct_declarator
       :   (   id=IDENTIFIER
           {     String d = $id.getText();
                      int line=$id.getLine();
                 int col=$id.getCharPositionInLine();
           String source="test";
                 if(input.LT(1).getText().equals("(")){
                       Node<Function> temp = CMT.functionTree.search(
                 new
Function(d,line,col,source,CMT.scopeStack.peek().getNumber()));
                       if(temp!=null)
                             temp.content.setDefinition_line(line);
                       else CMT.functionTree.insert(
                 new
Function(d,line,col,source,CMT.scopeStack.peek().getNumber()));
                       }else {CMT.idTree.insert(
                 new
Identifier(d,line,col,source,CMT.scopeStack.peek().getNumber()));}
      }
```

```
          |        '(' declarator ')'
          )
          declarator_suffix*
     ;
```

Function identifiers are stored with their declaration line and definition line. We also have to find out whether it is a definition or a declaration of the function. Therefore we have to search for the current identifier inside the function tree. If there is no such identifier in the tree, the current identifier represents a new declaration. Otherwise, if there is already an identifier with the same name, the current identifier is the definition of the function. By handling function identifier there is also the issue of overloading function names. For now this is not respected in this tool, it does not affect the correctness of the tool.

3.4 Scope and Functions

Problem:

Occurrences of variable identifiers are stored by first searching for a definition of the identifier inside the current scope, if there is no such definition it is searched for the definition in each valid scope on the scope stack going backwards. Now consider a function definition with its parameter list inside the parenthesis. This definition of the identifier is in the default scope 0. The definition of the same identifier is in the parameter list of the function declaration in scope 0 as well. A problem occurred while finding the right definition for an occurrence of an identifier, defined in the parameter list of a function. Because of the definition of the tree data structure this occurrence was always stored in the definition of the function declaration. To clarify this problem see the following code:

```
1 void function(int i);       // declaration of function

2 int main(void){ ...   }

3 void function(int i){       // definition of function
4     i++;        // this occurrence should be stored in definition of line 3
5 }
```

The occurrence of 'i' in line 4 should be stored in the definition of line 3. Because of the implementation of the tree it is wrongfully stored in the definition of line 1. Actually this is the only circumstance where two identifiers with the same name have their definition in the same scope. A solution needs to be found that this occurrence is stored correctly.

Solution:

The solution to the above defined problem is simply to define a kind of intermediate scope, so that there is a difference between both definitions in the same scope. A new scope is pushed on the stack when a left parenthesis occurs after a direct declarator. This means that the parameter list has its own scope. If there is a semicolon after the right parenthesis, the scope is popped from the stack, so that the scope becomes invalid. If there is a left curly bracket after the right parenthesis, the scope of the parameter list stays on the stack and the scope of the function body is pushed on the stack as well.

At the end of the function body the scope of the body must be popped as well as the scope of the parameter list. This is realized as follows. First we need a scope class with two variables: the scope number and a boolean variable 'popTwice', which indicates the popping of two scopes from the stack. Whenever there is a parameter list scope, the next scope will have its boolean 'popTwice' set to true. Therefore another boolean inside the grammar is used. It is set to true when a parameter list scope is pushed. The rule 'declarator_suffix' contains the parameter list of a function:

```
declarator_suffix
        :   '[' constant_expression ']'
      |   '[' ']'
      |
      { popItTwice=true;
         CMT.scope++;
         CMT.scopeStack.push(new Scope(CMT.scope,false));
      }
      '(' parameter_type_list ')'
      {
         if(input.LT(1).getText().equals(";")){
            CMT.scopeStack.pop();
            popItTwice=false;
         }
      }
      |   '(' identifier_list ')'
      |   '(' ')'
        ;
```

The variable 'popItTwice' is set to true and a new scope is pushed on the stack. This boolean is used for the condition of the next scope(indicated by left curly bracket). We have to ask if 'popItTwice' is true. If yes the boolean variable of the scope object must be set to true. See the following modified rule of the grammar:

```
compound_statement
        : { CMT.scope++;
                  if(popItTwice){
                        CMT.scopeStack.push(new Scope(CMT.scope,true));
                        popItTwice=false;
                  }
                  else CMT.scopeStack.push(new Scope(CMT.scope,false));
          }

            '{' declaration* statement_list? '}'

          { if(CMT.scopeStack.peek().isPopTwice()){
                  CMT.scopeStack.pop();
                  CMT.scopeStack.pop();
                  }else CMT.scopeStack.pop();
          }
        ;
```

If the condition is true the scope object is initiated by setting the variable 'popTwice' to true. Finally we just have to ask if the 'popTwice' variable of the current scope object is set to true. If yes, the stack is popped twice, so that the scope of the function body and the scope of the parameter list is popped.

3.5 Renaming

Problem:

This is a problem, which occurred during a test phase. For renaming one or all identifiers the C file is read line by line and for each line the tree is searched for an identifier in this certain line. If the current line equals the 'line' variable of an identifier object, each occurrence of the identifier name inside this line is replaced by a substitute by using the String method 'replace' in Java. Consider a typical line from a C program:

```
int i, index;
```

In this small example one can already see the problem. Assuming 'i' is supposed to be replaced by 'newId', then the outcome would be:

```
newIdnt newId, newIdndex;
```

Therefore a new way of replacing needs to be investigated.

Solution:

First of all the current line is split into a String array using a regular expression "\\W+", which splits whenever there is a non alphanumeric character sequence. Now each array element is compared with each identifier name, where the current line number equals the line variable in the identifier object. Afterwards the array is transformed into a String object and written into the output file. Unfortunately there is no method in Java which allows you to specify the column number, where you like to replace a substring. This would have made the renaming part much easier.

3.6 Include File Distinction

Problem:

It is possible to include system libraries and self written libraries into a C file. This is realized by the preprocessing command '#include' followed by the name of the file within angle brackets or quotes. Of course system libraries shouldn't be referenced, so that it is not possible to apply the rename facility to those. Nevertheless it is indispensable to reference all self written files as well. Imagine the user wants to rename an identifier inside a C file, but its definition is somewhere inside an included file. Hence there must be a method to distinguish between system and self written include files.

Solution:

It is common to use angle brackets for system libraries and quotes for self written files. This information is simultaneously the solution to the mentioned problem. There is a simple if-statement, which separates strings beginning with quotes from those which start with an angle bracket. This assumption requires that self written include files are in fact introduced by quotes.

3.7 Search path

Problem:

Handling file inclusion is no problem if the included files are located in the same folder as the main C file. Once there is another directory for self written h- files, one needs a kind of search in a certain environment. There is no such method in Java like '_searchenv( )' in C, that searches for a file using environment paths. Therefore an algorithm is needed to search for a particular file in a certain directory.

Solution:

After some research on the internet I've found several approaches on how to solve file searching in Java. One of those, which is used in the CMT project, is explained in the following, according to [2].

For scanning a directory there are already methods like listFiles( ) respectively list( ), referring to the Java-API. Return value of list( ) is an array of String objects including the file- and folder names inside the given folder, whereas listFiles( ) returns an array of File objects. To get all files from the directory, including all subfolders, it is useful to have an array of File objects, because the class 'File' has a method, which is necessary to scan subfolders recursively: 'isDirectory( )'. This method is true if the given File object is a directory. Hence we can search in all subfolders as well. The following code shows how to search for a file name 'find' inside a directory 'dir'. Using listFiles() there is an array of all files inside this folder. If the name of the current file equals the file name that is searched for, it is assigned to the return value. Otherwise, if the file is a directory it is searched as well by calling the method recursively.

```
File searchFile(File dir, String find) {

      File[] files = dir.listFiles();
      File matches = null;
      if (files != null) {
            for (int i = 0; i < files.length; i++) {
                  if (files[i].getName().equalsIgnoreCase(find)) {
                        matches=files[i];
                  }
                  if (matches == null && files[i].isDirectory()) {
                        matches=searchFile(files[i],find);
                  }
            }
      }
      return matches;
}
```

Inside the grammar this method is invoked. The directory of the input C file is used to search for the included file name as follows:

The directory in which the c file is located is searched first. If searchFile( ) returns null, the path of the c file is searched backwards.

```
...

      File incFile=null;
      do{
            currentDirectory = currentDirectory.getParentFile();
            incFile =searchFile(currentDirectory, includeFile);
      }while(incFile==null);

...
```

The header file that is searched for is assigned to the variable incFile and can be used in the following code. An Exception catches the case whether incFile remains null later on.

In most cases the header file contains only function prototypes besides preprocessing commands. Therefore the defining C file for the header file must be searched as well. The problem is, that it is not guaranteed having a C file that defines a header file. If there are only macro and variable definitions inside the header file, there is no need for having an additional C file. In order to avoid an infinite loop a restriction was added to the search for defining C files.

```
...

      int folderNum=0;
      do{
             curdir = curdir.getParentFile();
             newf =searchFile(curdir, newfile);
             folderNum++;
      }while(newf==null && folderNum<2);

...
```

Now the search goes at most 2 folders backwards. This is no optimal solution, because the search can take a long time according to the system it works on. Furthermore the user is forced to put the header files with their definition files at most 2 folders prior to the directory of the main C source file.

3.8 Typedef without Name

Problem:

During the testing phase of storing structure identifiers a problem occurred several times. The C language rules for naming structs are a little extensive. Thus it is possible to define structs like

```
struct point{
   int    x;
   int    y;
};
```

or to define a struct and simultaneously defining the type name, like

```
typedef struct {
   int    x;
   int    y;
} point;
```

The second example is widely used in C programs, because it is a handier and more comfortable solution to declare a variable by saying 'point p;', rather than 'struct point p;'.

While testing a random file from the internet, the grammar wasn't able to handle the following expression:

```
typedef struct ref_item{
  struct decl_item* the_decl;
};
```

Solution:

To my mind the typedef is redundant in the previous example, because there is no alias, so no need for typedef in this case. Nonetheless it is worth to mention because it took me quite a long time to test this file and detect the bug.

3.9 Structures

Problem:

Structures are defined at the beginning of a file. There are mainly two different possibilities to define structures.

1.

define structures by using a name, define structure variables by using keyword 'struct'

```
struct point{ // definition of a structure
   int    x;
   int    y;
};

struct point p; // declaration of a struct variable
```

2.

define structures by using typedef, define structure variable by using alias name

```
typedef struct{ // definition of a structure
   int    x;
   int    y;
} point;

point p; // declaration of a struct variable
```

Access to the content of a struct is realized by using the 'point'- or 'arrow'-operator:

```
p.x=0;
```

Assume that p is a pointer to a structure of type point. We would refer to the name member as

```
p->y=0;
```

Remember the implementation of how to find the declaration of an identifier occurrence. First it is searched for the declaration in the same scope, then going backwards through each currently valid scope until there is a declaration with the same identifier name. Now consider following code:

```
1      void function(int x){
2
3          point p;
4          p.x = x;
5      }
```

Obviously for both occurrences of x in line 4 the declaration in line 1 is assumed to be the corresponding one. Whereas the first x in line 4 actually refers to the structure 'point' and its content. Therefore a solution is needed, so that identifiers which follow a point or arrow are stored in another way than all the others. A further problem is how to know to which structure the variable belongs, because there is no rule that forces the programmer to give all identifiers in two different structures different names. Therefore following code is possible and makes it even more complicated to find a solution:

```
typedef struct{
   int    x;
   int    y;
}point1;


typedef struct{
   int    x;
   int    y;
}point2;
```

Way to Solution:

There is a rule in the grammar that refers to postfix expressions including the point and arrow operator:

```
postfix_expression
        :   primary_expression
        (   '[' expression ']'
        |   '(' ')'
        |   '(' argument_expression_list ')'
        |   '.' id=IDENTIFIER  {insertOcc($id);}
        |   '->' id=IDENTIFIER {insertOcc($id);}
        |   '++'
        |   '--'
        )*
        ;
```

There is already some action code, that stores the names of the identifier, but it is the ordinary solution getting the last declaration of this identifier name.

First thoughts led to the idea that the search has to take place in all scopes, not just in all valid scopes. That is ,because the scope of a structure ends with the occurrence of the right curly bracket of the structure body. To avoid this problem we simply neglect to pop the scope of the structure from the scope stack, so that all structure scopes are valid throughout the file.

Just to search for the first declaration of an identifier name is not accurate enough considering the last mentioned problem of having two (or more) different structures with the same identifier names inside of the structure. Furthermore you don't know in which file the variable is declared.

This is a very severe problem because of the fact that nearly each C project contains structures, but a suitable approach on how to solve it isn't available at the moment, although I spent lots of time to think about it and try solutions.

Therefore I try to explain an approach which unfortunately did not work. As I mentioned above there is a rule inside the grammar which handles postfix expressions. We have to intervene at this rule. Somehow we need to know to which structure the expression belongs, i.e. Which structure is before the 'dot'- or 'arrow'-operator. Furthermore we have to know which identifiers are generally declared inside a structure and in which structure. Thus the Identifier class gets three more attributes: a String 'belongsTo', a String called 'type' and a boolean attribute whether it is declared inside a structure called 'structure'. Now we can search for a declaration after those operators: the identifier has to have the same name as usual, furthermore 'structure' needs to be true and the 'belongsTo' attribute of the searched identifier must be the same as the 'type' attribute of the identifier before the 'dot'- and 'arrow'-operator. The implementation of this attempt does not work. Either there is an error in reasoning or a bug in my implementation, but consequently I am not able to handle structures until now.

3.10 Html Output

Problem:

To make the html output interactive CSS and JavaScript are used. The user can see the next or previous occurrence of a certain identifier by hovering over the name. A menu drops down and the user can choose between previous and next occurrence. In the background each identifier and each occurrence needs a unique name, so that it can be localised. Furthermore the number of occurrences must be known for each identifier in order to detect whether it is the last/ first occurrence. Hence a data structure is needed to store all identifiers with their number of occurrences. Originally JavaScript only offers arrays to build up a data structure. Therefore the Javascript library prototype.js [5] is used which includes the definition of a Hash object. Libraries are included by using its path, but exactly this is the problem. If the prototype file is situated inside the executable jar, we do not know where the user stores the jar file on the computer. Supposing we know the absolute path to the prototype file, what happens if the user moves the jar file to another location? Consequently the path is not valid any more and the html file doesn't work as expected.

Solution:

As I don't come to a suitable solution using absolute or relative pathnames, I decided to copy the prototype file into each folder where an html file is created. Furthermore the user has to be advised of not deleting the prototype file and just move html output as a couple with the prototype file. This solution is merely provisional as long as no other solution can be found.

4. Learning Success

Apart from the fact that my mother language is not English, this project was difficult in many ways. One reason for difficulties was simply because it was the first software project I have actually created on my own from scratch. But just this circumstances made me learn a lot during the project. First of all I could not believe that research phase takes at least 2 months, but during this phase I figured out how to do the research effectively. I improved my knowledge on how to use different media like internet, books and papers properly and I also discovered how to divide a huge amount of work into small pieces, so that you can solve problems step by step. I also learned the hard way that project plans are actually guidelines, because it is hard to be on schedule all the time, especially if you don't have lots of experiences how long which task may last. The importance of having someone to talk to turned out to be very important as well. Not only the supervisor was able to help with problems, but also the exchange with other students, who had to solve their problems. Different views and team work are useful to go on and collect new ideas.

Besides the mentioned soft skills, I learned a lot of software engineering skills. The project 'C Maintenance Tool' is built up on a C grammar. Therefore I had to get to know parsing of source code in general and the syntax of grammar files including regular expressions. The challenge was not to write a grammar, but understanding and using a given grammar correctly. After weeks of studying the grammar and trying to change and test I feel comfortable with grammar files now. By looking at an unknown grammar, there are obvious parts that can be differentiated, like rule definitions and token definitions. Although the parsing generator Antlr[3] was used in this project, it is possible to understand a grammar for e.g. Lex/Yacc[4], because grammar files are basically similar.

According to the chosen programming language, this project improved my knowledge about Java, albeit I have used it before. Especially creating, searching and writing files using the File, PrintStream and BufferedReader/Writer class from the package java.io played a major role in the CMT project.

Furthermore I was able to apply my knowledge about Html and Javascript while implementing the html output. Hence I could improve my overall programming knowledge.

5. Consequence and Advice

Now, as the project is ending, there were some problems that could have been avoided with a different approach or different time management.

I started studying and using the grammar file at the beginning of December 2009. In order to understand the syntax I started with a tutorial creating a simple calculator. This first contact with computer language grammars helped me to gain first insight. Nonetheless it was really difficult and time consuming to get to know the existing grammar for the C programming language. It is always difficult to become familiar with existing source code, but it is slightly more difficult to become familiar with an unknown source code in an unknown programming language. Therefore it took me a lot of time to find the correct location for the action code, whereas the inserted code looks pretty simple. I also have to admit that I am not able to understand and explain everything of the grammar until now. Therefore I should have started using the C grammar file and trying to add some action code earlier, already at the end of the research phase.

This problem implies a further consideration. You have to be very careful while choosing the tools you will use throughout the project time. While choosing an adequate parser generator I was torn between two different ones: the “old” Lex and Yacc[4] which is widely known on the internet, which implies that there are lots of elaborations and help, on the other hand the “new” parsing tool Antlr[3], which is more flexible in supporting programming languages and which is getting more and more popular on the internet. Now there is Lex/Yacc versus Antlr, where Antlr is more challenging. Looking backwards it would have been more comfortable and easier to use Lex/Yacc and maybe I would have been able to complete my project as I planed in the beginning. If you have to work on unknown things, like grammar files in my case, it is definitely better to use those tools that have a better support and more discussion boards on the internet. Nonetheless I would have chosen Antlr again if I have had to start from scratch. I think grammar files are basically similar regarding the language,furthermore software is a fast moving part of science where old is always facing the new. Therefore you have to chose for yourself at which state it is right to remain or step up.

It is necessary to consider the chosen programming language as well. Has it been a good choice to use Java in this project? Yes and no referring to different parts of the project. I am familiar with Java and it is very useful to create data structures and built up objects for identifiers. Nonetheless it is a bit hard to implement reading and writing to files,especially in the rename all identifiers part, which is by the way the most time consuming part of the CMT tool. You have to read and write line by line, therefore you have to go through each identifier and its occurrences for each line, so that you do not forget to rename an occurrence. Additionally you cannot just name the column where the word is located, there is no such method in Java. Hence one solution is to break each line into a String array where each element contains a word from the line, consequently you have another loop which downgrades the speed of the tool. Maybe C or C++ would have been faster in file writing and reading, but those are more error-prone and what is more important, the renaming all identifiers function is actually implemented to test the tool.

6. Conclusion

As I explained in section 2 two versions of the planned three versions are realised. Furthermore there are some problems that have not been solved yet. Nonetheless I am proud of what I achieved although I am not totally satisfied with the tool. The structure problem is really a very severe problem because actually each major C project uses structures. It would have been nice to reference macros as well, but that is not as bad. Although the project is not finished I learned a lot that will help me starting following projects.

7. Bibliography

[1] Parr, T., '*ANSI C ANTLR v3 grammar*', retrieved on 22-11-2009 from http://www.antlr.org/grammar/1153358328744/C.g

[2] '*Verzeichnisse durchsuchen/bearbeiten/auslesen'* , retrieved on 05-04-2010 from http://www.java-forum.org/allgemeines/33129-verzeichnisse-durchsuchen-bearbeiten-auslesen.html

[3] Parr, T., '*Why Use ANTLR?*', retrieved on 16-11-2009 from http://www.antlr.org/

[4] Levine, J.R., Mason, T., Brown, D., 1995, '*lex & yacc*', O'Reilly, ISBN 1-56592-000-7

[5] Prototype Core Team, *'Prototype – JavaScript Framework'* , retrieved on 15-03-2010 from http://www.prototypejs.org/