Institute of Technology, Carlow
B.Sc. in Software Engineering

CW228

Design Manual

C Maintenance Tool

Name: Anna-Christina Friedrich

ID: C00132716

Supervisor: Dr. Christophe Meudec

Submission Date: 02.02.2010

Table of Contents

1. Introduction

This paper is concerned with the design of the C Code Maintenance Tool (CMT). As mentioned in the specification manual, emphasis is on creating a reliable and stable program. Therefore the design of the tool is of high importance.

At first a reference to the use cases is given and a short explanation on which use case will be focused for now.

Using command line mode the following chapter handles the syntax how to start CMT from command line and how to chose each facility.

Main action happens inside the grammar, translated into parser and lexer via Antlr[1]. Therefore the C grammar used for CMT is given as well as the beginning of inserted Java action code.

A suitable datastructure is essential. The next chapter deals with the datastructure that stores all identifiers and its occurrences. Furthermore the problem of how to handle scope needs to be solved, so that a solution using a stack is explained.

2. Use Case Diagram

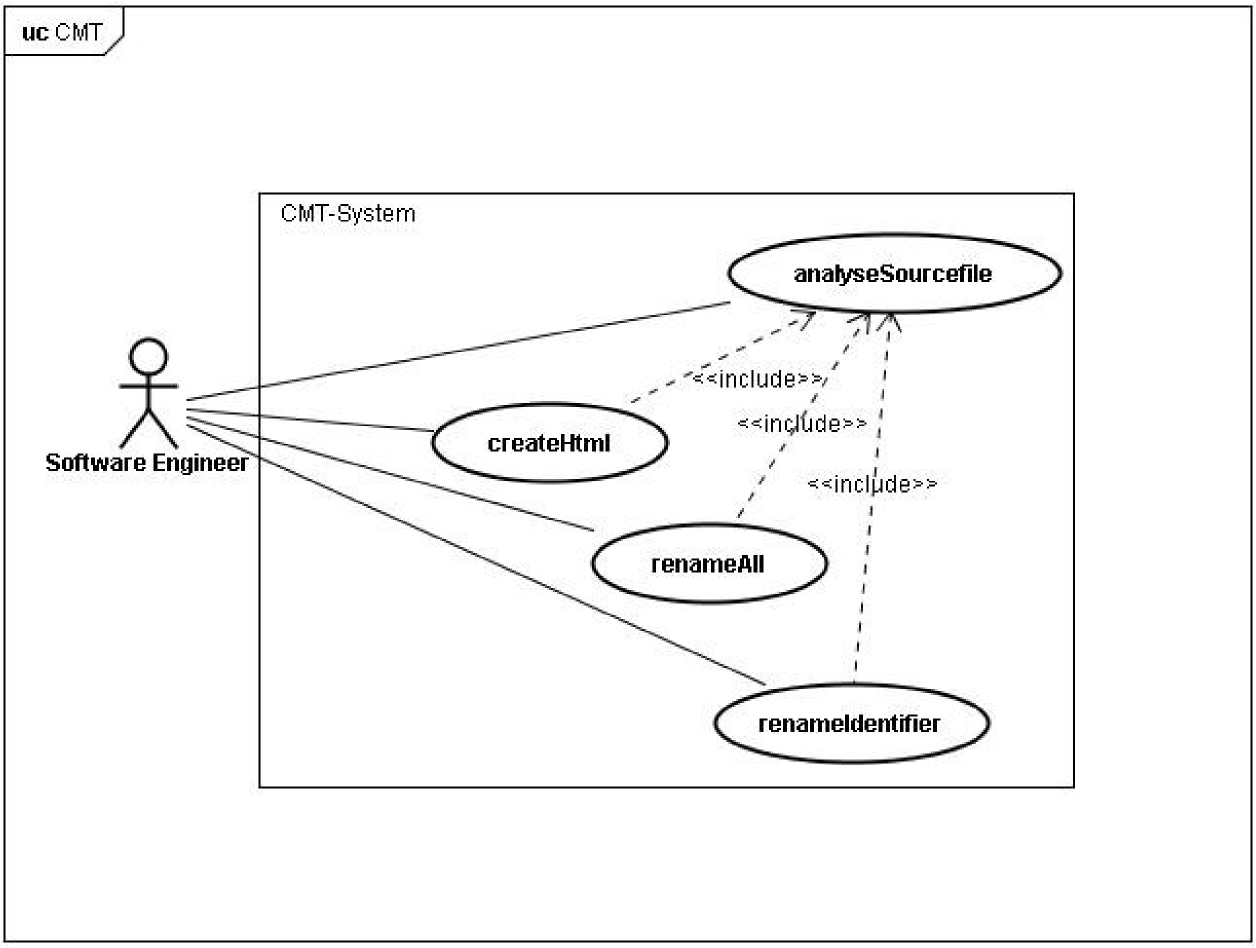

Figure 1: Use Case Diagram of CMT

3. Illustration of Use Cases

Although there are four different use cases, main focus is on 'analyseSourcefile'. This use case includes the analysis of the input C file realised by using a C grammar. Including the main action of CMT it is the most important use case. The first version of CMT is concerned to cover 'analyseSourcefile', so that identifiers of variables are properly stored so far.

4. Command Line Syntax

4.1 Starting CMT

CMT is delivered as an executable jar file. Syntax to start CMT with Windows command line:

Use `dir "name-of-folder"` to get into the folder where CMT.jar is located.

Use `java -jar cmt.jar Syntax` to start the tool using the Syntax given in the following chapter.

4.2 Syntax

There are few facilities for CMT. '~' means there needs to be a space:

- how to create html files for a set of C files
  - `createHtml ~ source ~ "path-to-C-File" ~ dest ~ "path-to-output-folder"`
- how to create a textfile for a set of C files
  - `createText ~ source ~ "path-to-C-File" ~ dest ~ "path-to-output-folder"`
- how to rename an identifier in a C file
  - first create html or textfile output to get definition of each identifier
  - the original file won't be changed, the updated file is stored in the output folder
  - `rename ~ source ~ "path-to-C-File" ~ dest ~ "path-to-output-folder" ~ id ~ "name-of-identifier" ~ scope ~ "scope-of-identifier" ~ newid ~ "new-name-for-identifier"`

5. ANTLR C-grammar

The following C grammar [2] provides a basis to analyse input C files. Additional specification to match the needs of CMT are explained in chapter 7.

```
/** ANSI C ANTLR v3 grammar

Translated from Jutta Degener's 1995 ANSI C yacc grammar by Terence Parr
July 2006.  The lexical rules were taken from the Java grammar.

Jutta says: "In 1985, Jeff Lee published his Yacc grammar (which
is accompanied by a matching Lex specification) for the April 30, 1985 draft
version of the ANSI C standard.  Tom Stockfisch reposted it to net.sources in
1987; that original, as mentioned in the answer to question 17.25 of the
comp.lang.c FAQ, can be ftp'ed from ftp.uu.net,
   file usenet/net.sources/ansi.c.grammar.Z.
I intend to keep this version as close to the current C Standard grammar as
possible; please let me know if you discover discrepancies. Jutta Degener, 1995"

Generally speaking, you need symbol table info to parse C; typedefs
define types and then IDENTIFIERS are either types or plain IDs.  I'm doing
the min necessary here tracking only type names.  This is a good example
of the global scope (called Symbols).  Every rule that declares its usage
of Symbols pushes a new copy on the stack effectively creating a new
symbol scope.  Also note rule declaration declares a rule scope that
lets any invoked rule see isTypedef boolean.  It's much easier than
passing that info down as parameters.  Very clean.  Rule
direct_declarator can then easily determine whether the IDENTIFIER
should be declared as a type name.

I have only tested this on a single file, though it is 3500 lines.

This grammar requires ANTLR v3.0.1 or higher.

Terence Parr
July 2006
*/
grammar C;
options {
    backtrack=true;
    memoize=true;
    k=2;
}

scope Symbols {
        Set types; // only track types in order to get parser working
}

@header {
import java.util.Set;
import java.util.HashSet;
}

@members {
        boolean isTypeName(String name) {
                for (int i = Symbols_stack.size()-1; i>=0; i--) {
                        Symbols_scope scope =
(Symbols_scope)Symbols_stack.get(i);
                        if ( scope.types.contains(name) ) {
                                return true;
                        }
                }
                return false;
        }
}

translation_unit
scope Symbols; // entire file is a scope
@init {
  $Symbols::types = new HashSet();
}
        : external_declaration+
        ;

/** Either a function definition or any other kind of C decl/def.
 *  The LL(*) analysis algorithm fails to deal with this due to
 *  recursion in the declarator rules.  I'm putting in a
 *  manual predicate here so that we don't backtrack over
 *  the entire function.  Further, you get a better error
 *  as errors within the function itself don't make it fail
 *  to predict that it's a function.  Weird errors previously.
 *  Remember: the goal is to avoid backtrack like the plague
 *  because it makes debugging, actions, and errors harder.
 *
 *  Note that k=1 results in a much smaller predictor for the
 *  fixed lookahead; k=2 made a few extra thousand lines. ;)
 *  I'll have to optimize that in the future.
 */
external_declaration
options {k=1;}
        : ( declaration_specifiers? declarator declaration* '{' )=>
function_definition
        | declaration
        ;

function_definition
scope Symbols; // put parameters and locals into same scope for now
@init {
  $Symbols::types = new HashSet();
}
        :       declaration_specifiers? declarator
                (       declaration+ compound_statement
                |       compound_statement
                )
        ;

declaration
scope {
  boolean isTypedef;
}
@init {
  $declaration::isTypedef = false;
}
        : 'typedef' declaration_specifiers? {$declaration::isTypedef=true;}
          init_declarator_list ';' // special case, looking for typedef
        | declaration_specifiers init_declarator_list? ';'
        ;
```

```
declaration_specifiers
        :   (   storage_class_specifier
                |   type_specifier
        |   type_qualifier
        )+
        ;

init_declarator_list
        : init_declarator (',' init_declarator)*
        ;

init_declarator
        : declarator ('=' initializer)?
        ;

storage_class_specifier
        : 'extern'
        | 'static'
        | 'auto'
        | 'register'
        ;

type_specifier
        : 'void'
        | 'char'
        | 'short'
        | 'int'
        | 'long'
        | 'float'
        | 'double'
        | 'signed'
        | 'unsigned'
        | struct_or_union_specifier
        | enum_specifier
        | type_id
        ;

type_id
    :   {isTypeName(input.LT(1).getText())}? IDENTIFIER
    ;

struct_or_union_specifier
options {k=3;}
scope Symbols;
@init {
  $Symbols::types = new HashSet();
}
        : struct_or_union IDENTIFIER? '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;

struct_or_union
        : 'struct'
        | 'union'
        ;

struct_declaration_list
        : struct_declaration+
        ;
```

```
struct_declaration
        : specifier_qualifier_list struct_declarator_list ';'
        ;

specifier_qualifier_list
        : ( type_qualifier | type_specifier )+
        ;

struct_declarator_list
        : struct_declarator (',' struct_declarator)*
        ;

struct_declarator
        : declarator (':' constant_expression)?
        | ':' constant_expression
        ;

enum_specifier
options {k=3;}
        : 'enum' '{' enumerator_list '}'
        | 'enum' IDENTIFIER '{' enumerator_list '}'
        | 'enum' IDENTIFIER
        ;

enumerator_list
        : enumerator (',' enumerator)*
        ;

enumerator
        : IDENTIFIER ('=' constant_expression)?
        ;

type_qualifier
        : 'const'
        | 'volatile'
        ;

declarator
        : pointer? direct_declarator
        | pointer
        ;

direct_declarator
        :   (   IDENTIFIER
                |       '(' declarator ')'
                )
        declarator_suffix*
        ;

declarator_suffix
        :   '[' constant_expression ']'
    |   '[' ']'
    |   '(' parameter_type_list ')'
    |   '(' identifier_list ')'
    |   '(' ')'
        ;

pointer
        : '*' type_qualifier+ pointer?
        | '*' pointer
        | '*'
        ;

parameter_type_list
        : parameter_list (',' '...')?
        ;

parameter_list
        : parameter_declaration (',' parameter_declaration)*
        ;

parameter_declaration
        : declaration_specifiers (declarator|abstract_declarator)*
        ;

identifier_list
        : IDENTIFIER (',' IDENTIFIER)*
        ;

type_name
        : specifier_qualifier_list abstract_declarator?
        ;

abstract_declarator
        : pointer direct_abstract_declarator?
        | direct_abstract_declarator
        ;

direct_abstract_declarator
        :       ( '(' abstract_declarator ')' | abstract_declarator_suffix )
abstract_declarator_suffix*
        ;

abstract_declarator_suffix
        :       '[' ']'
        |       '[' constant_expression ']'
        |       '(' ')'
        |       '(' parameter_type_list ')'
        ;

initializer
        : assignment_expression
        | '{' initializer_list ','? '}'
        ;

initializer_list
        : initializer (',' initializer)*
        ;

// E x p r e s s i o n s

argument_expression_list
        :   assignment_expression (',' assignment_expression)*
        ;

additive_expression
        : (multiplicative_expression) ('+' multiplicative_expression | '-'
multiplicative_expression)*
        ;

multiplicative_expression
        : (cast_expression) ('*' cast_expression | '/' cast_expression | '%'
cast_expression)*
        ;

cast_expression
        : '(' type_name ')' cast_expression
        | unary_expression
        ;

unary_expression
        : postfix_expression
        | '++' unary_expression
        | '--' unary_expression
        | unary_operator cast_expression
        | 'sizeof' unary_expression
        | 'sizeof' '(' type_name ')'
        ;

postfix_expression
        :   primary_expression
        (   '[' expression ']'
        |   '(' ')'
        |   '(' argument_expression_list ')'
        |   '.' IDENTIFIER
        |   '->' IDENTIFIER
        |   '++'
        |   '--'
        )*
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

primary_expression
        : IDENTIFIER
        | constant
        | '(' expression ')'
        ;

constant
    :   HEX_LITERAL
    |   OCTAL_LITERAL
    |   DECIMAL_LITERAL
    |   CHARACTER_LITERAL
        |       STRING_LITERAL
    |   FLOATING_POINT_LITERAL
    ;

/////

expression
        : assignment_expression (',' assignment_expression)*
        ;

constant_expression
        : conditional_expression
        ;

assignment_expression
        : lvalue assignment_operator assignment_expression
        | conditional_expression
        ;

lvalue
        :       unary_expression
        ;

assignment_operator
        : '='
        | '*='
        | '/='
        | '%='
        | '+='
        | '-='
        | '<<='
        | '>>='
        | '&='
        | '^='
        | '|='
        ;

conditional_expression
        : logical_or_expression ('?' expression ':' conditional_expression)?
        ;

logical_or_expression
        : logical_and_expression ('||' logical_and_expression)*
        ;

logical_and_expression
        : inclusive_or_expression ('&&' inclusive_or_expression)*
        ;

inclusive_or_expression
        : exclusive_or_expression ('|' exclusive_or_expression)*
        ;

exclusive_or_expression
        : and_expression ('^' and_expression)*
        ;

and_expression
        : equality_expression ('&' equality_expression)*
        ;
equality_expression
        : relational_expression (('=='|'!=') relational_expression)*
        ;

relational_expression
        : shift_expression (('<'|'>'|'<='|'>=') shift_expression)*
        ;

shift_expression
        : additive_expression (('<<'|'>>') additive_expression)*
        ;

// S t a t e m e n t s
```

```
statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

labeled_statement
        : IDENTIFIER ':' statement
        | 'case' constant_expression ':' statement
        | 'default' ':' statement
        ;

compound_statement
scope Symbols; // blocks have a scope of symbols
@init {
  $Symbols::types = new HashSet();
}
        : '{' declaration* statement_list? '}'
        ;

statement_list
        : statement+
        ;

expression_statement
        : ';'
        | expression ';'
        ;

selection_statement
        : 'if' '(' expression ')' statement (options {k=1;
backtrack=false;}:'else' statement)?
        | 'switch' '(' expression ')' statement
        ;

iteration_statement
        : 'while' '(' expression ')' statement
        | 'do' statement 'while' '(' expression ')' ';'
        | 'for' '(' expression_statement expression_statement expression? ')'
statement
        ;

jump_statement
        : 'goto' IDENTIFIER ';'
        | 'continue' ';'
        | 'break' ';'
        | 'return' ';'
        | 'return' expression ';'
        ;

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

fragment
LETTER
        :       '$'
        |       'A'..'Z'
        |       'a'..'z'
        |       '_'
        ;

CHARACTER_LITERAL
    :   '\'' ( EscapeSequence | ~('\''|'\\') ) '\''
    ;

STRING_LITERAL
    :  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;

HEX_LITERAL : '0' ('x'|'X') HexDigit+ IntegerTypeSuffix? ;

DECIMAL_LITERAL : ('0' | '1'..'9' '0'..'9'*) IntegerTypeSuffix? ;

OCTAL_LITERAL : '0' ('0'..'7')+ IntegerTypeSuffix? ;

fragment
HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
IntegerTypeSuffix
        :       ('u'|'U')? ('l'|'L')
        |       ('u'|'U')  ('l'|'L')?
        ;

FLOATING_POINT_LITERAL
    :   ('0'..'9')+ '.' ('0'..'9')* Exponent? FloatTypeSuffix?
    |   '.' ('0'..'9')+ Exponent? FloatTypeSuffix?
    |   ('0'..'9')+ Exponent FloatTypeSuffix?
    |   ('0'..'9')+ Exponent? FloatTypeSuffix
        ;

fragment
Exponent : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
FloatTypeSuffix : ('f'|'F'|'d'|'D') ;

fragment
EscapeSequence
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   OctalEscape
    ;

fragment
OctalEscape
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
    ;
```

```
COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;

// ignore #line info for now
LINE_COMMAND
    : '#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;
```

6. Data structure

6.1 Storing Identifier

C programs can consist of thousands of lines of code, cosequently there are maybe hundreds of identifier and their occurrences. That is why CMT needs to have an appropriate data structure to store each single identifier with each occurrence. There are some data structures, like List, Stack and Tree. Using a Stack doesn't make sense to store identifier. Because of the large amount of items to be stored, the data structure has to be very efficient.

Assuming that a List is used adding a definition of an identifier would not be very time consuming. But each time there is an occurrence of an identifier, the list has to be iterated to find the definition. This happens with a complexity of O(n), where n is the length of the list.

According to the speed in searching, using a binary tree is more efficient. The worst case is that the definition of all identifiers is ordered alphabetically, which turns the tree into a list, so that the complexity is O(n) as well, where n is the depth of the tree. But in the best case the tree is balanced, so that each parent node has two children. This case makes the search faster; the complexity is O(log n). But the most convincing criterion for the decision is that the average case ia complexity of O(log n) as well.

According to the efficiency of datastructures a binary tree will be used to store identifiers and their occurrences.

6.2 Handling Scope

As elaborated in previous manuals, the scope of identifier is one of the main problems. Assuming there is a variable called index, defined in the main()-method, and there is as well a variable called index, defined in a loop. If index is called inside the loop it has to refer to the definition inside the loop. If index is called outside the loop the occurrence has to be added to the first definition in main(). The problem of handling scope in CMT is, so far, solved with an Integer variable 'scope'. According to the progress of the tool, the variable 'scope' will probably need more information, so that a helpful data structure will be used: the stack.

First of all the classes 'Identifier' and 'Occurrence' get an Integer attribute called 'scope'. Obviously a new scope starts with left curly bracket and ends with rigth curly bracket. The Integer (Stack) is initialised with 0. Whenever there is a left curly bracket, 'scope' is increased (and pushed on the stack). The whole purpose is that each new instance of 'Identifier' and 'Occurrence' are initialised with the current scope number. If there is a right curly bracket, 'scope' is decreased by 1 (popped from the stack). Therefore each occurrence can be allocated to its correct definition, because each occurrence of an identifier will be allocated to its definition having the same or lower number of scope.

The following code shows the use of the scope stack.

```
1  #include <stdio.h>                           SCOPE = 0
2
3  int main(void){                              SCOPE = 1
4
5    int i=0, index;
6
7    for( index = 0; index <=3; index++){       SCOPE = 2
8
9         int i = 2;
10        if(i%2==0){                           SCOPE = 3
11          i += index;
12        }                                     SCOPE = 2
13   }                                          SCOPE = 1
14
15   return 0;
16 }                                            SCOPE = 0
```

An extract of the output would be:

name: i, definition line: 5, scope: 1
occurrences: -

name: i, definition line: 9, scope: 2
occurrences: [line: 10, scope:2;
line: 11, scope: 3]

name: index, definition line: 5, scope: 1
occurrences: [line: 7, scope: 1;
line: 11, scope: 3]

Of course 'index' in line 7 will have three occurrences with different column numbers, but I left it out, because the example is focusing on scope.

7. Adjusting Grammar

The given C grammar needs to be provided with action code. As the programming language of CMT is Java this code has to be written in Java. Action code can be inserted after the colon following a rule name by using curly brackets. Because of the fact that lexical analysis happens before syntactical analysis it is important to consider whether putting action code into lexer or parser part of the grammar. Considering the scope variable always refering to expressions inside curly brackets, it is reasonable to use only the parser part of the grammar. This assures that each scope is identified correctly. According to the first version of CMT it will just handle one C file. Therefore the variable 'source' is always initalised by 'test' as long as it is important for the tool.

7.1 Handling Scope

The rule to handle statements in the given C grammar is called 'compound_statement'.

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

Scope variable is a static variable defined in class CMT. Whenever there is a left curly bracket a new scope starts, so that the variable has to be increased by 1. Right curly bracket indicates the end of the scope. Therefore the scope variable has to be decreased by 1. Following code shows the rule with inserted action code:

```
compound_statement
      : '{' {CMT.scope+=1;}declaration* statement_list? {CMT.scope-=1;}'}'
      ;
```

7.2 Handling Definition of Identifier

The rule to identify definition/declaration of identifier is implemented by the following code.

```
direct_declarator
       :    (    IDENTIFIER
                 |       '(' declarator ')'
                 )
       declarator_suffix*
       ;
```

To refer to the identifier it is more comfortable to assign it to a variable, here called 'id'. Inside the action code 'id' can be used by putting the $ sign in front of it. Name, line and column of the identifier are used to create a new instance of the class Identifier. This object is stored inside the binary tree 'idTree', a static variable of CMT as well. Following code shows the rule with inserted action code:

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

7.3 Handling Occurrences of Identifier

To handle occurrences of identifier the code to be included is more comprehensive. The rule to be extended is called 'primary_expression'.

```
primary_expression
        : IDENTIFIER
        | constant
        | '(' expression ')'
        ;
```

Similar to definition handling a variable is needed to refer to the identifier. But the intention is now to store an occurrence. Therefore it has to be searched for a definition. The class Identifier implements the method 'compareTo()', which compares name and scope number while comparing identifier with each other. If there is a definition in the same scope, the occurrence is added to this definition. If there is no definition of the identifier in the same scope, it is searched for a definition with the next smaller scope number.

```
primary_expression
       : id=IDENTIFIER

       {      String d = $id.getText();
              int line=$id.getLine();
              int col=$id.getCharPositionInLine();
              String source="test";

              Node<Identifier> temp = CMT.idTree.search(new
              Identifier(d,line,col,source,CMT.scope));
              if(temp!=null)
                     temp.content.getOccurrences().insert(new
                     Occurrence(d,line,col,source,CMT.scope));
              else{
                     for(int index=CMT.scope-1;index>=0;index--){
                            Node<Identifier> temp2 = CMT.idTree.search(new
                            Identifier(d,line,col,source,index));
                            if(temp!=null){
                                   temp2.content.getOccurrences().insert(new
                                   Occurrence(d,line,col,source,CMT.scope));
                                   index=-1;
                            }
                     }
              }
       }

              | constant
              | '(' expression ')'
        ;
```

8. Conclusion

The design of software basically affects the efficiency of a program. Therefore it is very important to adjust the design of certain software to its specific needs.

CMT is a tool that has to handle huge datastructures. According to its efficiency the datastructure 'binary tree' is used. Secondly the basis of CMT is the educed C grammar. First action code snippets were added to the grammar, so that scope of variables of one C file can be considered.

Prospectively, it is possible that the variable 'scope' will turn into a class, which leads to the use of a further datastructure Stack. The code inserted in the grammar is changing as well while progress. Rules for handling further C files, macros and conditional directives must be applied, while distinction between system libraries and self written .h-files is a big problem. Therefore this paper will be updated during the whole project.

9. Bibliography

[1] Parr, T., '*ANTLR v3*', retrieved on 20-01-2010 from http://www.antlr.org/

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