Institute of Technology, Carlow

B. Sc. in Software Engineering

CW228

Research Manual

C Maintenance Tool

Name: Anna-Christina Friedrich

ID: C00132716

Supervisor: Dr. Christophe Meudec

Table of Contents

1. Introduction

In the context of the 4th year project in the course 'B. Sc. Software Engineering' I will elaborate on the topic 'A C Maintenance Tool'.

According to [1], a website about programming language popularity which combines several researches, C is estimated to be the second most used programming language. Because of the fact that there is a lot C code currently in use, programmers should get support by understanding and changing those. There are various so called refactoring tools for object oriented languages, but there are merely a few tools for refactoring C. That is why this project is concerned with creating a tool for understanding legacy C code.

To go into detail a definition of the term 'refactoring' is useful. Martin Fowler gives a quite good definition of software refactoring in his book 'Refactoring-Improving the Design of existing Code':

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.[3]

In the scope of refactoring, the tool should perform at first full cross-referencing of all the identifiers appearing in a set of C files. Each identifier will be listed with its occurances and declarations. This approach leads to some problems that have to be considered.

First of all the scope of variables. The tool has to distinguish between local and global identifiers, even if they have the same name. There may be a variable in the scope of a loop with the same name as global variable, e.g. the tool has to make sure that those variables are presented as two different ones. Another main problem we have to be concerned about is the occurance of preprocessing commands. While preprocessing does not contain legal C code, there have to be made additional rules. Assuming there is a macro calling a global variable, the tool has to figure out the appearance of the variable in the macro. Therefore preprocessing commands have to be included while cross-referencing.[2]

Starting with a research about similar tools, available on the internet, and some tryouts, I will educe ideas for the cross-referencing tool from research. A slight overview of C language, C standard, scope notion and macros will be given as well. A further important point of research is the choice of the programming language, which will be argued in a further section of this paper. Since C code needs to be scanned properly, a tool for parsing Code must be applied. Thus I will talk about Parser generators like Lex and Yacc and Antlr, considering which one suites the best for the cross-referencing tool. Finally the above mentioned problems with preprocessor commands are explained in more detail.

While simulating a realistic Software development, emphasis must be on the reliability and accuracy of the tool.

2. Similar Tools

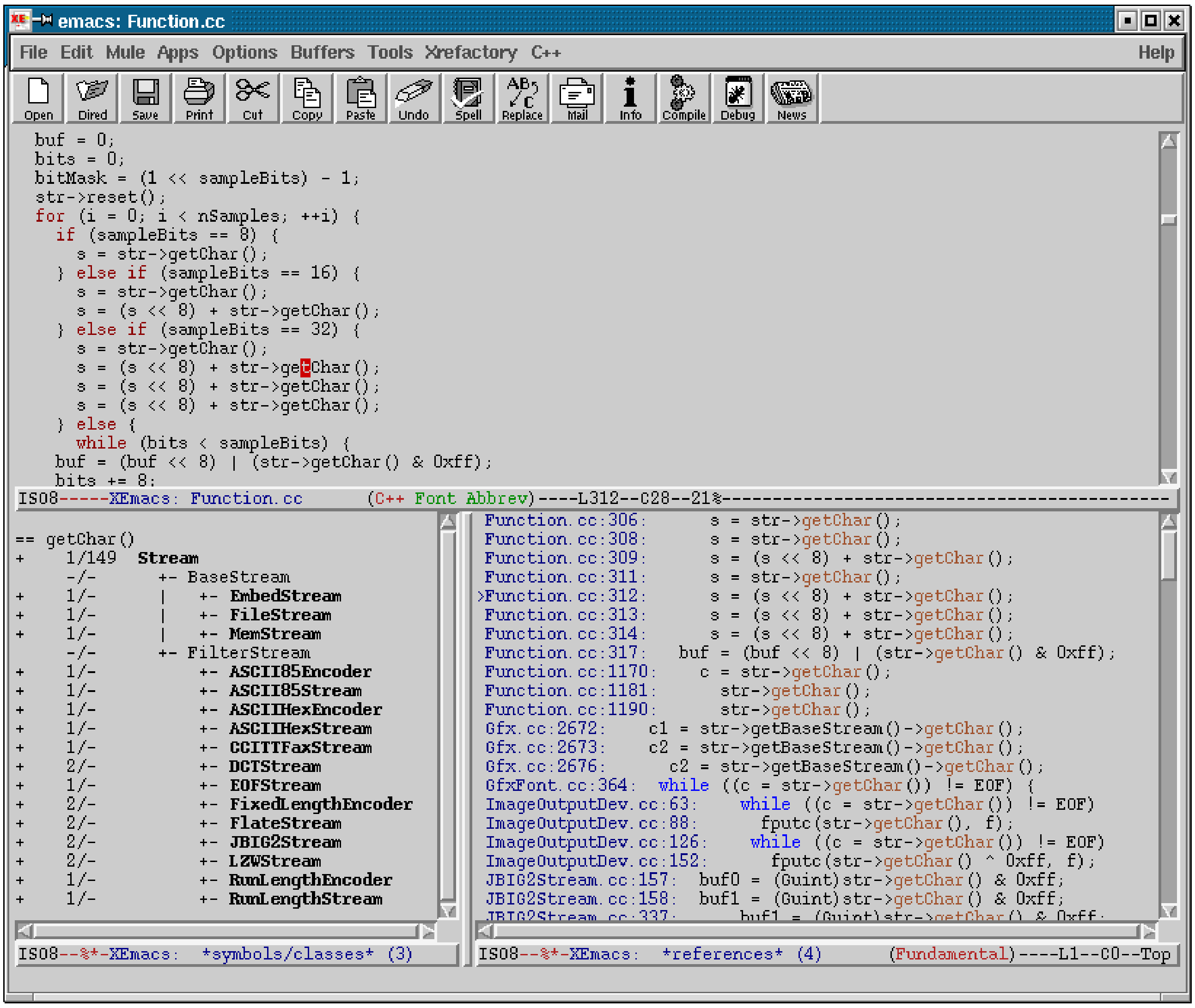
2.1 Listing

2.1.1 Xrefactory[4]

· development tool for C and Java

·Refactoring: method (function) extraction; renaming parameters, variables, fields (structure records) and methods (functions); insertion, deletion and moving of parameters, field and method moving;

·Source browsing: own tag implementation supporting multiple preprocessing passes and resolving scopes, accessibilities, overloading and polymorphism



```
  buf = 0;
  bits = 0;
  bitMask = (1 << sampleBits) - 1;
  str->reset();
  for (i = 0; i < nSamples; ++i) {
    if (sampleBits == 8) {
      s = str->getChar();
    } else if (sampleBits == 16) {
      s = str->getChar();
      s = (s << 8) + str->getChar();
    } else if (sampleBits == 32) {
      s = str->getChar();
      s = (s << 8) + str->getChar();
      s = (s << 8) + str->getChar();
      s = (s << 8) + str->getChar();
    } else {
      while (bits < sampleBits) {
    buf = (buf << 8) | (str->getChar() & 0xff);
    bits += 8;
ISO8-----XEmacs: Function.cc        (C++ Font Abbrev)----L312--C28--21%----------------------------------------
```

```
== getChar()
+    1/149   Stream
     -/-       +- BaseStream
+    1/-       |   +- EmbedStream
+    1/-       |   +- FileStream
+    1/-       |   +- MemStream
     -/-       +- FilterStream
+    1/-           +- ASCII85Encoder
+    1/-           +- ASCII85Stream
+    1/-           +- ASCIIHexEncoder
+    1/-           +- ASCIIHexStream
+    1/-           +- CCITTFaxStream
+    2/-           +- DCTStream
+    1/-           +- EOFStream
+    2/-           +- FixedLengthEncoder
+    2/-           +- FlateStream
+    2/-           +- JBIG2Stream
+    2/-           +- LZWStream
+    1/-           +- RunLengthEncoder
+    1/-           +- RunLengthStream
ISO8--%*-XEmacs:  *symbols/classes* (3)
```

```
 Function.cc:306:       s = str->getChar();
 Function.cc:308:       s = str->getChar();
 Function.cc:309:       s = (s << 8) + str->getChar();
 Function.cc:311:       s = str->getChar();
>Function.cc:312:       s = (s << 8) + str->getChar();
 Function.cc:313:       s = (s << 8) + str->getChar();
 Function.cc:314:       s = (s << 8) + str->getChar();
 Function.cc:317:    buf = (buf << 8) | (str->getChar() & 0xff);
 Function.cc:1170:     c = str->getChar();
 Function.cc:1181:       str->getChar();
 Function.cc:1190:       str->getChar();
 Gfx.cc:2672:    c1 = str->getBaseStream()->getChar();
 Gfx.cc:2673:    c2 = str->getBaseStream()->getChar();
 Gfx.cc:2676:      c2 = str->getBaseStream()->getChar();
 GfxFont.cc:364:  while ((c = str->getChar()) != EOF) {
 ImageOutputDev.cc:63:    while ((c = str->getChar()) != EOF)
 ImageOutputDev.cc:88:      fputc(str->getChar(), f);
 ImageOutputDev.cc:126:    while ((c = str->getChar()) != EOF)
 ImageOutputDev.cc:152:      fputc(str->getChar() ^ 0xff, f);
 JBIG2Stream.cc:157:  buf0 = (Guint)str->getChar() & 0xff;
 JBIG2Stream.cc:158:  buf1 = (Guint)str->getChar() & 0xff;
 JBIG2Stream.cc:337:      buf1 = (Guint)str->getChar() & 0xff;
ISO8--%*-XEmacs:  *references* (4)        (Fundamental)----L1--C0--Top
```

Figure 1: browsing sourcecode for 'getChar'

2.1.2 Cscope[5]

· developer's tool for browsing source code

·Allows searching code for:

o all references to a symbol

o global definitions

o functions called by a function

o functions calling a function

o text string

o regular expression pattern

o a file

o files including a file

· information database is generated for faster searches and later reference

· command line mode for inclusion in scripts or as a backend to a GUI/frontend

· runs on UNIX

2.1.3 C-Free[6]

· C/C++ integrated development environment

· Code finding utility: first choose scope then symbol

· Jump to declaration/definition

· List of all symbols in the program

2.1.4 Pelles C[7]

·development kit for Windows and Windows Mobile

·code editor with call tips and symbol browsing

2.1.5 Cxref[8]

·produce documentation (in LaTeX, HTML, RTF or SGML) including cross-references from C program source code

·works with ANSI C

·Files

·The files that the current file is included in (even when included via other files).

·#includes

·Files included in the current file.

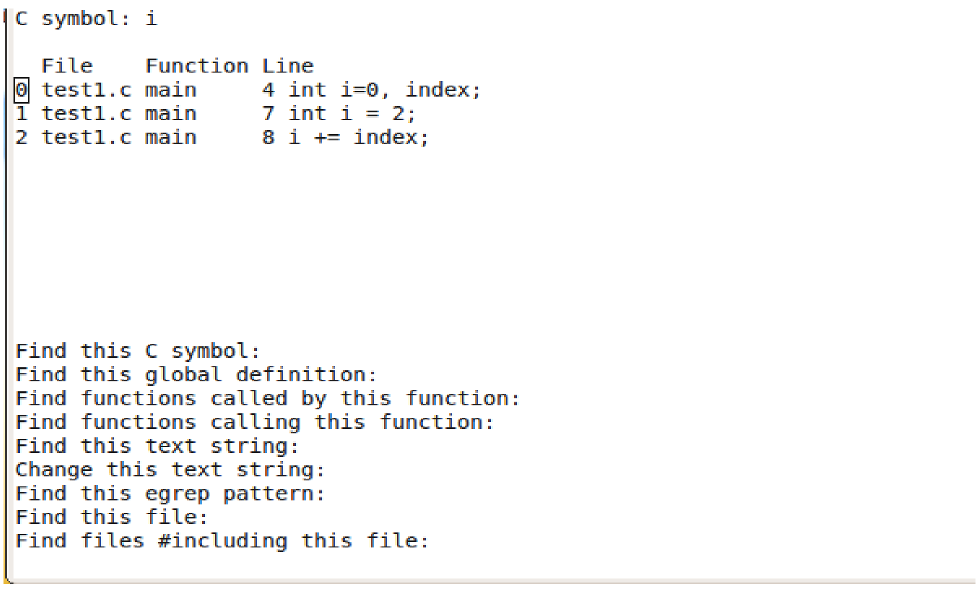
    - ·Files included by these files etc.
- ·Variables
    - ·The location of the definition of external variables.
    - ·The files that have visibility of global variables.
    - ·The files / functions that use the variable.
- ·Functions
    - ·The file that the function is prototyped in.
    - ·The functions that the function calls.
    - ·The functions that call the function.
    - ·The files and functions that reference the function.
    - ·The variables that are used in the function.

2.2 Try out

2.2.1 Cscope

Installing Cscope[5] was a large amount of work. First I tried to install it on Windows, because I am more used to it. Therefore the tool 'Cygwin' has to be installed to run Unix/Linux programs on Windows. But then it turned out into downloading each single package used by Cscope. Finally I stoped trying it on Windows and moved to Linux-Ubuntu. First I had to get to know how installing on Linux works. That was easy, because there are lots of sources in the internet. A further problem occured while doing the make-file. There was no bison and flex installed on my computer. Again I researched how to install those and finally Cscope was running. If you are a 'professional' using Linux there is no problem in installing it, but if you are new to it, it lasts a long time since you have finished install.

I tested Cscope by executing with Prog2(see 4.3.2.1). There are two different identifiers having the same name 'i'. As you can see in the following extract the identifiers are correctly listed, but there is no certain distinction between the two of them. I am missing a statement, which groups identifiers that belong to each other.Like: 'i' from line 4 is different from line 7 and 'i' from line 7 is the same as in line 8.



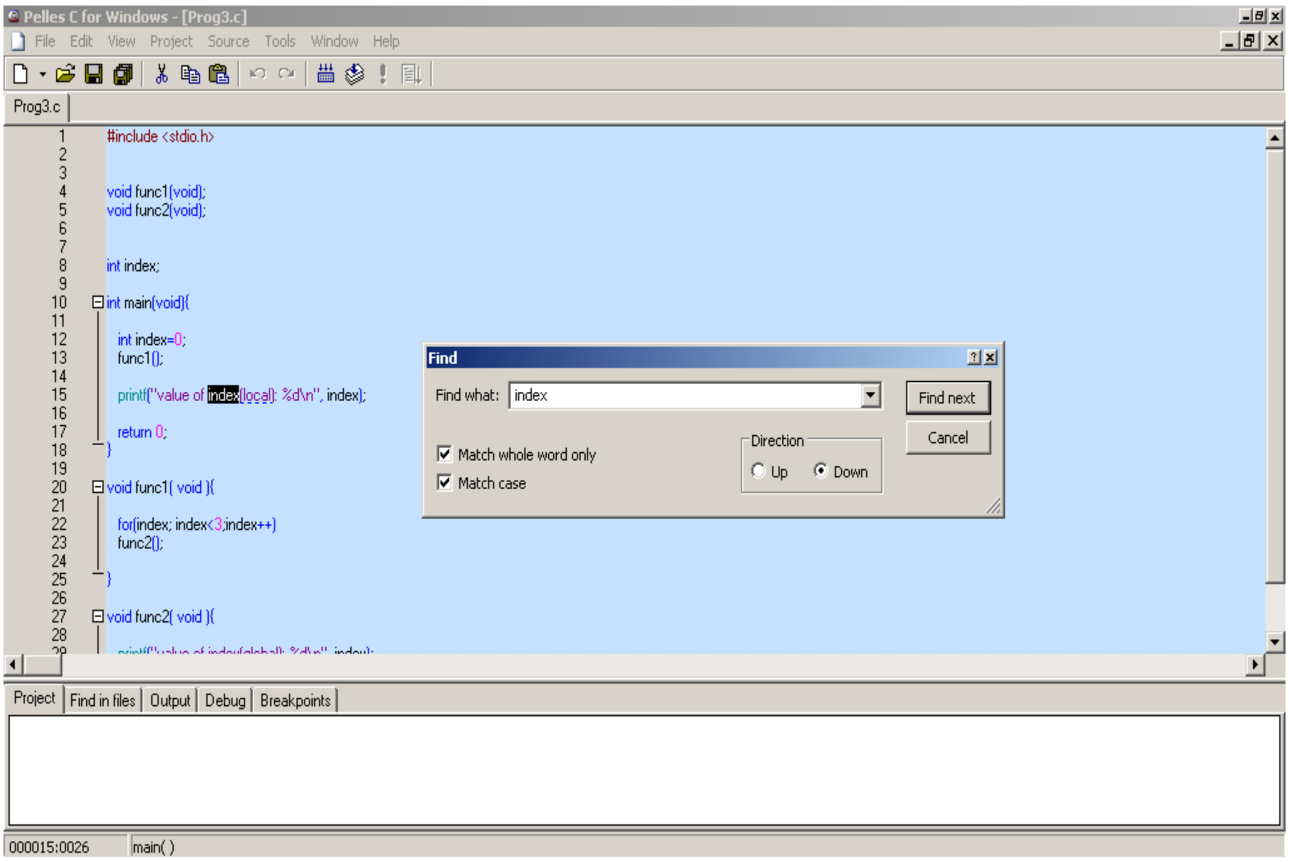
```
C symbol: i

  File    Function Line
0 test1.c main     4 int i=0, index;
1 test1.c main     7 int i = 2;
2 test1.c main     8 i += index;








Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 2: Example output of Cscope

A further annoiance occured when I tried to exit the program or to get from the sourcecode back to the program. I wasn't able to find a proper command to exit this tool.

2.2.2 PellesC

There were no problems with download and installation from [7]. It is a development kit, thus you got a graphical UI you can use intuitionally. It is possible to search for certain expressions in the editor, but the scope is not supported. That means if you are searching for the identifier ‘index’, it will run through the text highlighting every single word containing ‘index’, what is more comments are highlighted as well. If you choose ‘only whole word’ it will highlight every identifier that is called ‘index’. Replacing is also supported, but it’s the same problem. If you replace an identifier ‘index’ by ‘newindex’ you can choose between replacing the whole word only and match the case, but scope isn´t supported as well. Every single identifier ‘index’ will be replaced. That is not very useful for refactoring. See an exemplary output of PellesC in Figure 3 using program 3 from 4.3.2.2.


Pelles C for Windows - [Prog3.c]
File Edit View Project Source Tools Window Help
Prog3.c
Project | Find in files | Output | Debug | Breakpoints
Find
Find next
Cancel

```
#include <stdio.h>


void func1(void);
void func2(void);


int index;

int main(void){

    int index=0;
    func1();

    printf("value of index(local): %d\n", index);

    return 0;
}

void func1( void ){

    for(index; index<3;index++)
    func2();

}

void func2( void ){

    printf("value of index(global): %d\n", index);
```

Find what: index

[✓] Match whole word only
[✓] Match case

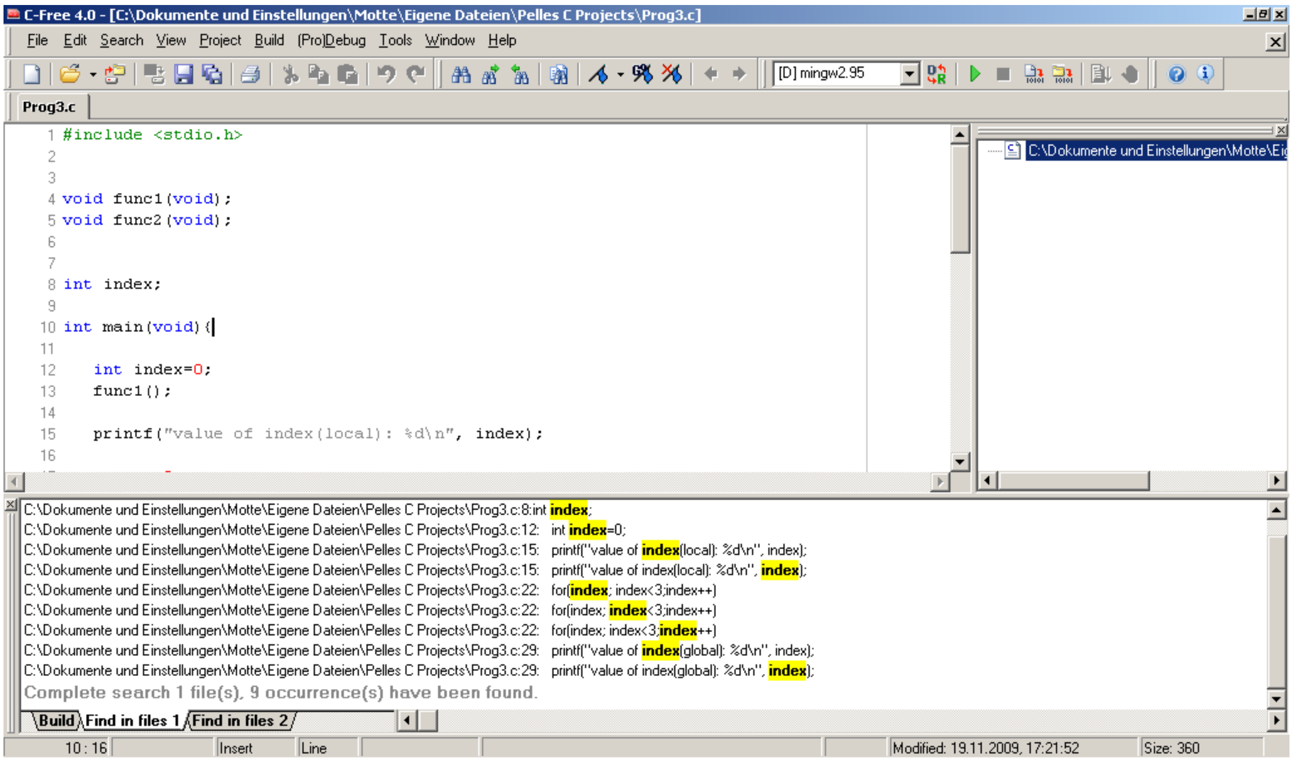
Direction: Up [✓ Down]

000015:0026 main( )

Figure 3: GUI of PellesC in search mode

2.2.3 Cfree

Download and installation works as usual from [6]. There were no problems. Cfree is very similar to Pelles C. It has an intuitional Windows-like menu bar, but it offers the same search and replace tools as Pelles C. Scope is not supported, therefore searching and replacing matches every identifier having the same name. The difference to PellesC is, that there is a concluded search result window, as you can see in Figure 4 using program 3 from 4.3.2.2.


C-Free 4.0 - [C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c]
File Edit Search View Project Build (Pro)Debug Tools Window Help
[D] mingw2.95
Prog3.c
C:\Dokumente und Einstellungen\Motte\Ei
Build
Find in files 1
Find in files 2
10 : 16
Insert
Line
Modified: 19.11.2009, 17:21:52
Size: 360

```
#include <stdio.h>


void func1(void);
void func2(void);


int index;

int main(void){

    int index=0;
    func1();

    printf("value of index(local): %d\n", index);

```

```
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:8:int index;
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:12:   int index=0;
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:15:   printf("value of index(local): %d\n", index);
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:15:   printf("value of index(local): %d\n", index);
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:22:   for(index; index<3;index++)
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:22:   for(index; index<3;index++)
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:22:   for(index; index<3;index++)
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:29:   printf("value of index(global): %d\n", index);
C:\Dokumente und Einstellungen\Motte\Eigene Dateien\Pelles C Projects\Prog3.c:29:   printf("value of index(global): %d\n", index);
Complete search 1 file(s), 9 occurrence(s) have been found.
```

Figure 4: GUI of CFree, output of search mode

2.2.4 Cxref

Again I first tried to run it on Windows, but as the first error occured I switched to Linux. Now it was quite easy to download from [8], install and run, because I have done it before for Cscope(see 2.2.1). Unfortunately it didn't work as it is supposed to work. The command 'cxref -html filename' should provide html-files for the chosen C-file. There was an html file indeed, but it merely displayed: 'int main(void){'.

3. Starting ideas for my tool

First of all there will be no graphical user interface, but the tool will be based on command line, because there are already lots of tools for browsing sourcecode. My tool will merely create a datastructure with crossreferenced identifiers without displaying it on a GUI. There will be probably a plugin for an existing development kit. The main functionality of the tool will focus on cross referencing C code. Therefore html files will be built that refer to identifiers. Each name appearance, declaration, definition and maybe first usage will be displayed. It is very important to consider the scope of the variables, so that two variables having the same name are identified as two different variables.

A further approach is to include comments to the html files like JavaDoc annotations for Java. Perhaps a search and replace/rename option will be added.

4. C language

4.1 Overview

The C language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s. It is a small and efficient high level language with a huge run-time library. It is widely used in system programming which includes operating systems and embedded systems. C language has been and remains popular, because programs can be written quickly and well by an experienced programmer. A further factor of its popularity is due to standardisation for nearly 20 years.

A C program consists of one source file or can be divided into several source files. Each of the files contains a part of the entire program code. Functions that are used very often are usually collected in so called 'Header' files, that can be included by using the command '#include'. A special function, called 'main', is the function where the program starts and it contains operations and different function calls.[9]

A compiler translates the sourcecode into object code, where the first step is invoking the C preprocessor to translate preprocessing commands into legal C-Code.[10] When the compiler has finished its work, the linker resolves references between the modules and detects occuring errors, finally creating an executable program. [9]

4.2 C-Standards

There were created two main C standards, whereas the second standard includes the first one chronologically. The first standard of the language was ratified in 1989, known as ANSI C, Standard C and C89. One year later, in 1990, ISO(International Organisation of Standardisation) transferred ANSI C to an ISO standard, which is called C90. Thus C89 and C90 include the same C standard.

The second standard C99, ratified in 1999, adapted the standard of C89/90 and added several new features. With C99 you can use inline-functions, data types like ‘long long int’ and ‘complex’, variable-length arrays, variadic macros and comments beginning with ‘//’ lasting one line.

Current C compilers support the C89 standard and almost every program written in C refers to C89. Nonetheless various C compilers support also features of C99. [19]

Therefore I will focus on C89 as well.

4.3 Scope

4.3.1 Overview

The scope of a variable is the area of a program, in which the variable is visible. There are six different scopes in which variables can appear[9]:

1. Top-level identifiers: visibility lasts until the end of the code
2. Formal parameters in function definitions: visibility lasts until the end of the function body
3. Block local identifiers: visibility lasts until the end of a block (e.g. loops,if-else-statements)
4. Formal parameters in function prototypes: visibility lasts until the end of the prototype
5. Statement labels: compasses entire function body in which it appears
6. Pre-processor macros: visibility lasts from its definition #define until #undef or the end of the source program file

4.3.2 Examples

It is useful to think about problematic structures considering the scope of variables. Therefore a couple of examples are shown that focus on identifiers having the same name in different scopes. In order that it won't get too complicated examples are divided into local and global appearance of variables. Following sections are inspired by[9].

4.3.2.1 Local variables

```
Prog1:
#include <stdio.h>

/* declaration of function */
void function(void);

int main(void){

int i_main = 1;

  function();
  return 0;
}

/* definition of function */
void function(void){

int i_func = 2;

i_main += i_func; /* this will cause an error*/

21 }
```

The problem is as follows: i_main is a variable that is defined in main()[line 8] and known only in this block. Allocating a value to i_main in function()[line 19] causes an error, because i_main is not known in this block. In the following my tool will have to handle only correct C code, so that the mentioned problem is not worth worrying about.

The scope of an identifier can be limited within a function, because the scope is not geared to the function, but rather to blocks. The body of a function is just a special case of a block, because it has always to be put in curly brackets. Therefore you can also define further variables within certain blocks considering that definitions have to be made at first after the left curly bracket.

This is an example for identifiers having the same name:

```
Prog2:
1  #include <stdio.h>
2
3  int main(void){
4
5    int i=0, index;
6
7    for( index = 0; index <=3; index++){
8
9         int i = 2;
10        i += index;
11   }
12
13   return 0;
14 }
```

The identifier i is defined twice; in main()[line 5] and in for-loop[line 9]. In line 9 i is defined in the body of the loop and is valid until the end of the loop. i[line 9] overlays i[line 5]. i[line 9] is recreated in every single loop and initialised by the value 2.

This is an example that the tool will have to solve properly. If i[line 5] will be renamed, i[line 9] has to have still the same name: i.

4.3.2.2 Global variables

A variable is global if it is defined outside of a function. Its scope lasts until the end of the program. Usually all global variables are defined before the main()-function. If there is a global and local variable having the same name, the outer (global) variable will be overlayed by the inner (local) variable, as we have already seen in Prog2.

```
Prog3:
1  #include <stdio.h>
2
3  /* declaration of functions */
4  void func1(void);
```

```
5  void func2(void);
6
7  /* global identifier index */
8  int index;
9
10 int main(void){
11
12   /* local identifier index */
13   int index=0;
14   func1();
15
16   printf("value of index(local): %d\n", index);
17
18   return 0;
19 }
20
21 void func1( void ){
22
23   for(index; index<3;index++)
24   func2();
25
26 }
27
28 void func2( void ){
29
30   printf("value of index(global): %d\n", index);
31 }
```

Local variable index in line 13 is only visible in main(). It overlays the global variable index, defined in line 8. The functions func1() and func2() access global variable index[line 23&30], but they can't 'see' the local variable index. The output is as follows:

```
value of index(global): 0
value of index(global): 1
value of index(global): 2
value of index(local): 0
```

You can see that a change of one index does not influence the other one.

4.4 Preprocessing

4.4.1 Overview

C sourcecode actually does not include preprocessor commands. Therefore there is a C preprocessor, which translates preprocessing commands into valid C code, so that the compiler can read the program properly. That is important, because the syntax of preprocessor commands is completely independent of C syntax. As mentioned there are few preprocessor commands.Those commands are always introduced by the single character '#' and can be expanded to further lines by ending the '#'-line with backslash '\'. Preprocessing commands can be divided into 3 different blocks: file inclusion, macro definition and conditional compilation.[9]

4.4.2 File inclusion

Including files into C code is implemented by using '#include <filename>' or '#include "filename" '. Both forms are searching for files in certain directories, whereas files included by angle brackets are generally standard implementation files and files included by quotes refer to programmer written files. File inclusion means that the sourcecode can refer to the entire content of the included file as if the included file appears instead of the #include command. There is also a nesting of inclusions possible, e.g. the included file includes files as well etc.. [9]

exemplary definitions:

```
#include <stdio.h>          /* header file for standard I/O */
#include <string.h>         /* header file to handle strings */
#include "complex.h"        /* file written by programmer */
```

4.4.3 Macro definition

Macros are implemented by using '#define'. There are two different ways of defining a macro: objectlike macro definition and macros with parameters.

An objectlike macro has no arguments, it starts with a name,usually written in capital letters, followed by a sequence of tokens. #include , name and sequence are separated by a white space.

#include name sequence-of-tokens

The name is now associated with the following tokens. The macro can now be invoked by merely including the name into the sourcecode. Objectlike macros are basically used to define certain numbers, like length of an array, so that changes to it can be made easily. The programmer has to exercise care in creating macros, because there may occur errors while adding '=' between name and sequence or adding semikolon at the end of a macro by mistake.

Exemplary definition:

```
#define ARRAY_LENGTH 12           /* a constant is defined */
#define ERRMSG „Error %d: %s.\n‟ /*output to include e.g. in printf()*/
```

Contrary to objectlike macros, there are also functionlike macros that take arguments. There is still a name, but followed by a list of parameters seperated by commas within brackets. Finally there is again a sequence of tokens.

#define name(parameter-list) sequence-of-tokens

Now the name of the functionlike macro is associated with the sequence of tokens, but there are more constraints when invoking it. You have to take care that, by invoking the macro, there are always the same number of parameters as defined within the brackets that follow the name. There is also a condition that the left paranthesis has to follow immediately the name, otherwise the bracket sequence is interpreted as the sequence of tokens, like an objectlike macro. The programmer has also to pay attention to the sequence of tokens. While invoking the macro you have to consider how the macro is defined. There are for instance already curly brackets defined in the body of the macro, so that you don't have to add those within the sourcecode.

While the C preprocessor interprets the commands, macro invokings are replaced by their definitions. A copy of the macro body is made, in which each formal parameter is replaced by the actual argument. This operation is called 'macro expansion'.

Exemplary definition:

```
#define product(x,y)  ((x)*(y))   /* multiplies two arguments */
#define getchar() getc(stdin)     /* get next character */
```

Generally, the definition of a macro can be canceled by writing '#undef'. From now on the macro is not known anymore by the preprocessor. There is also a way to check, if a macro has already been defined, for instance within an included file.[9]

Therefore a definition can be ensured by adding following to the definition:

```
#ifndef name
#define name sequence-of-tokens
#endif
```

4.4.4 Conditional compilation

C preprocessor allows also the use of conditional commands: #if, #else and #endif. That means, that lines of sourcecode can be passed through or eliminated on the basis of computed condition.

The resulting condition must be a constant arithmetic value that can be interpreted as a truth value. This value needs to be established already at build time.
The range of the conditional compilation will be completed with the command #endif (on a separate line). It is also possible to introduce an "otherwise"-sector with the term '#else' in the conditional compilation. Alltogether it looks like:

```
#if constant-expression
        group-of-lines1
#else
        group-of-lines2
#endif
```

Adding further condition branchs can be implemented by using '#elif', which is just a shorthand for '#else #if'. [9]

5. Parsing Generator

5.1 Overview

If there is a tool planned using some kind of interpreting a programming language, you will need to know something about parser generators.

First of all the sourcecode has to be lexical analysed. In the lexical analysis of source code the input stream will be separated into individual symbols (tokens). The lexer (or even called 'scanner') distinguishs between the symbols defined by the language syntax, variables and operators.[11]

After the lexical analysis, there is a syntactic analysis made by a parser. The parser imports the input stream that is made up of tokens and proves if the syntax is properly observed. Finally, symbol groups with a hierarchical structure are built, which are usually displayed as trees, called parse-tree.[12]

Since the creation of lexer and parsers, especially for complex programming languages such as C or C++, can be a very tedious affair, tools were developed, called compiler-compiler. Those tools generate the grammar from a given source code for a lexer and parser. The most famous 'team' is Lex (a Lexergenerator) and Yacc (the corresponding parser)[13]. A further very popular tool is Antlr[15].

5.2 Antlr versus Lex/Yacc

As mentioned above, Lex/Yacc and Antlr are the most popular tools for generating usable code from a certain grammar. Therefore a short overview about both of them is given in the following.

Lex and Yacc are two seperate tools, whereas Lex is a lexical and Yacc a syntactical analyser. They are usually used together, so that Lex provides the input-tokenstream for Yacc. Those tools were actually built for Unix systems, but meanwhile there are Windows versions available as well.[13]

ANTLR, ANother Tool for Language Recognition, is a language tool that contains both, a lexical and syntactical analyser. Furthermore Antlr generates a parse-tree. There is a graphical user interface, called AntlrWorks, which can be used intuitively. It comes with an executable jar-File, therefore it is platform independent.

One major difference is that ANTLR generates an LL(*) parser, whereas YACC generates parsers which are LL(1).That means,that Antlr uses k tokens of lookahead when parsing a sentence, which makes Yacc grammar more complex.[14]

A further topic worth considering is the output language. While Yacc provides code exclusively for C/C++, Antlr provides code in Java, Python, C#, C and C++. [15]

Antlr is also characerised by an easy readable code that is generated and an advanced IDE in AntlrWorks that makes writing the grammar much easier.[16]

5.3 Conclusion

Considering the mentioned arguments, I decided to use Antlr as a parser generator tool for my project. Especially the fact that there is an easy to use interface for Antlr, which provides your needs in one tool, and that the generated code is easy to understand and to include into your project made me choose this one. There is an exemplary C grammar especially for Antlr given in appendix 1.

6. Target Programming Language

There are lots of possible programming languages you can use. Considering the parser generator I am going to use, Antlr supports C/C++/JAVA/C#/Python.

C and C++ can be difficult, because of the memory management about which you don’t have to worry in Java. Furthermore languages from the 'C family' are made to access specific hardware addresses and allow the user to program close to hardware. That is why those languages are usually picked for programming operating systems.

I don’t know C# at all and I have just started using Python. I was really angry as I tried to use similar tools (see 2.2), because lots of tools just run on Unix/Linux. I really like to avoid this problem; therefore I pick Java or Python, because it is platform independent. I have got a lot experience with Java, but Python seems interesting as well.

Thus I have to consider what my tool will be like and which languages suites in a better way. The most important factor in a refactoring tool is to handle large data structures. According to some ‘Java versus Python’ elaborations[17] Python has very high level built-in data structures (lists, tuples, sets, maps), but Java provides better performance. A further argument for Java is good support of programming mistakes while compiling, whereas several programming errors appear in Python code during runtime, maybe days, months or years later.

At conclusion Java will be my choice as programming language for my cross referencing tool.

7. Problems with C code cross referencing (Challenges of refactoring C programs)

7.1 Preprocessor

Before a C program is actually translated into machine code, the compiler starts the preprocessor that executes changes and extensions of the source code. Each preprocessing command starts with '#'.As elaborated in 4.4 there are three main types of preprocessing:

1. Including a file with #include

2. Macro definitions implemented with #define, #undef

3. And conditional directives like #if, #elif plus #else and #endif

Considering refactoring, preprocessing is probably the most difficult thing to handle, because it is no legal C code. Therefore you have to consider whether refactor the sourcecode before or after preprocessing.

7.1.1 File inclusion

File inclusion isn't as difficult as macros and conditional directives, because #include <filename> expands the scope of refactoring, i.e. a further file has to be refactored. There will be differentiation between system files like <stdio.h>, that won't be included into refactoring, and programmers files, that will be refactored as well. The only problem is that the original file needs to be identified after preprocessing.

7.1.2 Macro definition

It is more difficult to handle macros. I will focus on finding appearances of macros and its parameters for cross referencing. Therefore it is important to find the definition and each call of a macro. It is also important to find appearances of global variables, which are called within a macro, and also variables, which a macro can call only localy.

According to A. Garrido[2], there are few sources of errors you have to worry about:

Suppose a macro refers to a global variable 'var' and there is also a local variable 'var' defined in main(), a function or a block statement. The macro is called in the scope of the local variable, consequently the local 'var' is used in the macro.

Exemplary macro definition and call:

```
Prog 4:
#define MACRO var = 1
int var;                   /* global variable */
int main(void){
   ...
   if(condition)
         MACRO;            /* macro calls global var */
   ...
}
int function1(){
   int var;                /* local variable */
   ...
   if(condition)
         MACRO;            /* macro calls local var */
   ...
}
```

Therefore it is important to check if a macro calls a variable, that has the same name as a variable in each scope of a macro call. Then it is impossible to e.g. rename the global variable without causing the code to become wrong.

The concatenation operator ## adds two variables in a macro. If those tokens are strings the result might be a variable as well, that is going to be referenced.

Exemplary use of concatenation:

```
Prog 5:
#define CAT(x,y)     x ## y

int main(void){
   int tablesize;
   ...
   if(CAT(table, size) < 10){
         ...
```

Therefore the tool first has to find out if there is a macro call including concatenation in the scope of a variable, which is to be referenced.

7.1.3 Conditional directives

Conditional directives offer alternatives of a conditional. Obviously there are the same sources of errors as figured out for macro definitons, but the main problem is to handle the directives, to refer to [2]. There is always just one directive executed, so that the remaining parts become discarded. Nonetheless each alternative of a conditional directive has to be considered while cross referencing.

Exemplary definition of conditional directives:

```
Prog 6:
#ifndef  _BUFFER
   #define _BUFFER
   int nelems;
#else
```

```
    int nelems;
#endif
```

7.1.4 Conclusion

Actually the usual approach is to preprocess macros and conditional directives before refactoring those. Because of the problem, especially in conditional directives, that some code is discarded and therefore not referenced after preprocessing, I will try a different approach, where the original C code will be refactored including special rules for preprocessing commands. Furthermore problems with file inclusion won't probably occur, because each file can still be treated on its own.

8. Conclusion

Based on this document, I create a C maintenance tool, that primarily produces a crossreferenced datastructure for each identifier in a given sourcefile. Furthermore there is an adequate output format, like html, and the facility to rename identifiers. Finally I aim at creating a plugin for a development kit, like eclipse, visual studio, etc..

The tool is based on ANSI C (C89) and considers the scope of C language as well as preprocessing commands. Analysing a programming language forces the use of a parser generator. Because of mentioned reasons above, my choice is Antlr, for which you can see a C grammar in the appendix 9.1. The programming language is Java.

The main problems will be to handle the sourcefiles before preprocessing and the preprocessing commands by itself. Furthermore a suitable rule for handling the scope of variables needs to be found.

9. Appendix

9.1 C grammar for Antlr[18]

```
/** 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 // K&R style
                |       compound_statement                              // ANSI
style
                )
        ;

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
//      {System.out.println($IDENTIFIER.text+" is a type");}
    ;

struct_or_union_specifier
options {k=3;}
scope Symbols; // structs are scopes
@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
                        {
                        if ($declaration.size()>0&&$declaration::isTypedef) {
                                $Symbols::types.add($IDENTIFIER.text);
                                System.out.println("define type "+
$IDENTIFIER.text);
                        }
                        }
                |       '(' 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;}
    ;
```

10. Bibliography

[1] DedaSysLLC, 13-10-2009, '*Normalized Comparison*', retrieved on 08-11-2009 from http://langpop.com/

[2] Garrido, A, Johnson, R, 2002, '*Challenges of Refactoring C Programs*', retrieved on 24-10-2009 from http://www.lifia.info.unlp.edu.ar/papers/2002/Garrido2002.pdf

[3] Fowler, M, 2002, '*What is refactoring?*' from '*Refactoring: improving the design of existing code*', Addison-Wesley, 431 pages, ISBN 0-201-48567-2

[4] Xref-Tech, '*Xrefactory for C/Java*', retrieved on 09-11-2009 from http://www.xref.sk/xrefactory-java/main.html

[5] Steffen, J, Bröker, H.-B., 09-10-2009, '*CSCOPE*', retrieved on 09-11-2009 from http://cscope.sourceforge.net/

[6] Program Arts Co., Ltd, 20-04-2008, '*C-Free*', retrieved on 09-11-2009 from http://www.programarts.com/cfree_en/index.htm

[7] 02-08-2009, '*PellesC*', retrieved on 09-11-2009 from http://www.smorgasbordet.com/pellesc/index.htm

[8] Bishop, A. M., 16-02-2007, '*C Cross Referencing & Documenting tool*', retrieved on 09-11-2009 from http://www.gedanken.demon.co.uk/cxref/

[9] Harbison III, S. P., Steele Jr., G. L., 2002, '*C: A Reference Manual*', Prentice Hall, 533 pages, ISBN 0-13-089592x

[10] Wikipedia contributors, 3-11-2009, '*C preprocessor*', Wikipedia, The Free Encyclopedia, retrieved on 09-11-2009 from http://en.wikipedia.org/w/index.php?title=C_preprocessor&oldid=323705586

[11] Wikipedia contributors, 11-11-2009, '*Lexical analysis*', Wikipedia, The Free Encyclopedia, retrieved on 15-11-2009 from http://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=325188260

[12] Wikipedia contributors, 14-11-2009, '*Parsing*', Wikipedia, The Free Encyclopedia, retrieved on 15-11-2009 from http://en.wikipedia.org/w/index.php?title=Parsing&oldid=325877343

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

[14] Spiewak, D., 17-10-2008, '*Advantages of Antlr (versus say, lex/yacc/bison)*', retrieved on 16-11-2009 from http://stackoverflow.com/questions/212900/advantages-of-antlr-versus-say-lex-yacc-bison

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

[16] Deva, P., '*ANTLR-Studio zum Entwickeln von ANTLR-Grammatik benutzen*', retrieved on 16-11-2009 from http://placidsystems.com/mag/ANLTR_DE.pdf

[17] de Jong, I., 17-08-2008, '*Python compared to Java*', retrieved on 16-11-2009 from http://www.razorvine.net/python/PythonComparedToJava

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

[19] Wikipedia contributors,22-11-2009, '*C (programming language)*', Wikipedia, The Free Encyclopedia, retrieved on 22-11-2009 from http://en.wikipedia.org/w/index.php?title=C_(programming_language)&oldid=327285035