The Source Code Generator Project #VYB0705N03x

by

Pierre-Andre Vungoc #260149525

Greg Polosa #260150888

Pascal Brisset #260152097

to

Professors Joseph Vybihal and Ken Fraser

Software Engineering Project

ECSE 494 & ECSE 495

McGill University

Department of Electrical, Computer and Software Engineering

December 7th 2007

ABSTRACT

The Source Code Generator Project is a software development project where a source code file in C-programming language is analyzed and converted to a similar file with automatic generated comments added to it. The goal of this particular project is to produce a graphical user interface as well as a code parser to provide a base the artificial intelligence developers will work with. This project was developed under the Borland[1] and GCC[2] C++ environment. The goals were well achieved and the results can be observed in the content of this report as well as in the demonstrations illustrated in the appendices.

1 Borland C++ Compiler. http://dn.codegear.com/article/20633
2 The GNU Compiler Collection. http://gcc.gnu.org/

ACKNOWLEDGEMENTS

We would like to thank Professor Joseph Vybihal of the department of computer science of McGill university for his time, generosity and great advice concerning his very own project. Professor Vybihal guidance throughout the project was essential for us in order to achieve our goals.

TABLE OF CONTENTS

a) Introduction..p.1

I) Overview..p.1

II) Initial State..p.1

III) Current State and Assigned Tasks..p.2

1. Graphical User Interface..p.3

1.1 Overview..p.3

1.2 Viewing and Editing the File List..p.4

1.3 Adding Comments Manually to Files..p.6

1.4 Generating Comments on a File List..p.7

1.5 Consulting and Refining the Dictionary..p.8

1.6 Consulting the Help Contents and About..p.9

2. Code Merger..p.10

3. Text Parser Output..p.11

4. Parser..p.12

3.1 Overview..p.12

3.2 Parser Structure..p.12

3.3 Extended Parser Details..p.17

b) Conclusions..p.18

I) Summary and Opinions..p.18

II) Possible Enhancements..p.18

c) References..p.19

d) Appendices..p.20

I) Appendix I: Demonstration of Code Merger..p.20

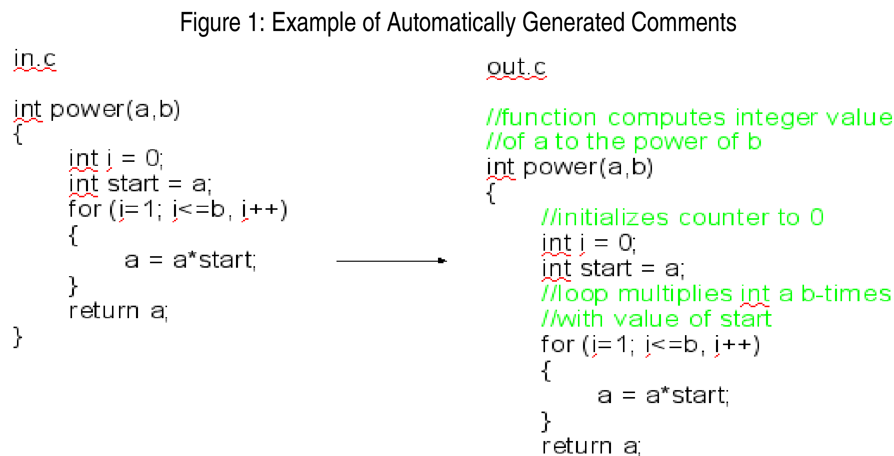
II) Appendix II: Demonstration of Parser Reconstruction..p.21

INTRODUCTION

I) Overview

The main goal of the Source Code Generator Project is to generate automatic comments for a programming code source file in C-programming language. As example, a source file can be taken as input and outputted with added comments to it as seen in figure 1 below:

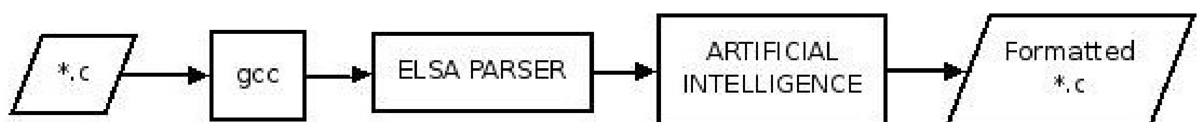
Figure 1: Example of Automatically Generated Comments



II) Initial State of the Project

It is important to first consider that previous team(s) worked on this project. In their design, an input *.c file was compiled through the GCC[2] compiler and then translated to the Elsa[3] parser, a C-language parser found on the web. The parsed code was then going through the artificial intelligence (AI) module to finally create a final formatted *.c file with comments added. All these operations were executed through the Unix command-line. The schematic of this process is shown below in figure 2:

Figure 2: Initial Process of the Source Code Generator

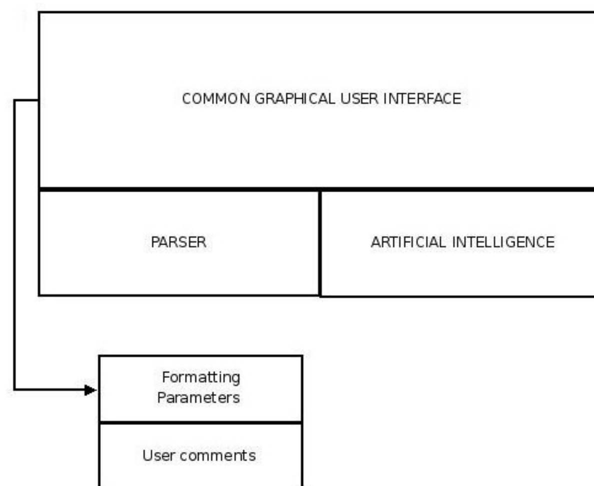


3 Elsa: An Elkhound-based C++ Parser. http://www.cs.berkeley.edu/~smcpeak/elkhound/

III) Current State and Assigned Tasks

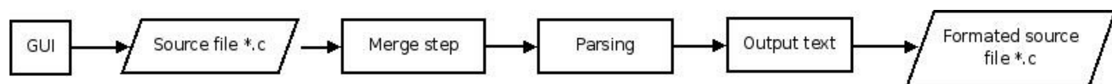
At the beginning of this current project, the design was slightly revised. Instead of going through the command-line interpreter, it was preferable to implement a simple graphical user interface (GUI) to make the work of the user as easy as possible. From this interface, all the different processes executed for the code generation become invisible to the user, just as one single operation. The structure of this system can be seen in figure 3 below:

Figure 3: Current Structure of the Source Code Generator



Since artificial intelligence is an advanced domain of computer science, it was decided that is was preferable to work on the parsing part of the project. The following structure in figure 4 was adopted:

Figure 4: Diagram of the Specific Tasks to Accomplish



It was decided that Pascal, student in the computer science department, was going to design the GUI. On the other hand, Pierre-André, student in software engineering, was going to code the merge step and the text parser output while Greg, in computer engineering, was going to implement the parsing part. All the specific steps to the process described earlier are further explained later on in the report.

GRAPHICAL USER INTERFACE

1.1) Overview

The basic motivation for using a graphical user interface for our Source Code Generator is simply to increase user friendliness and create a greater sense of accessibility and usability. This is accomplished through allowing easier access to the application's information and actions available to the user. Actions are performed through direct manipulation of graphical elements, which allows the user to actually see what he or she is working with. Furthermore, it gives the user the ability to operate the program visually, rather than having to input commands on a keyboard through a command line. As a matter of fact, previous implementations of this project were constructed in such a way that required the user to input commands through the command line which was very time consuming and cumbersome. Finally, using a GUI greatly reduces the learning curve of program usage, since there no need to memorize and learn how to use thousands of commands.

The GUI implementation consists of five major options which are:

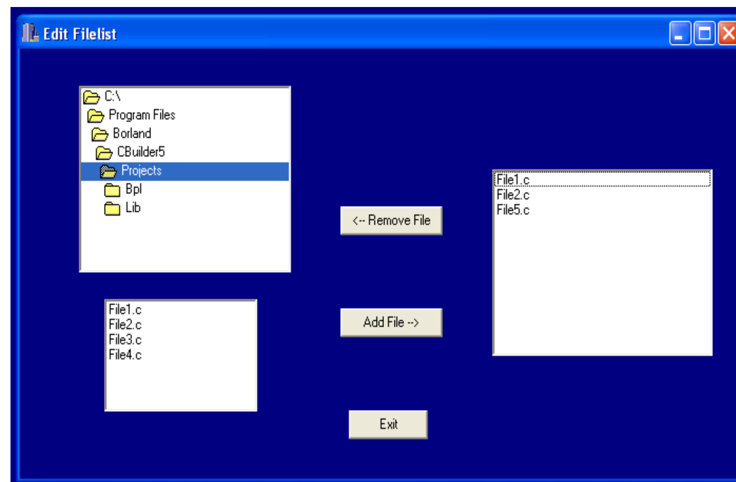
- Viewing and editing the file list
- Adding comments manually to files
- Generating comments on a file list
- Consulting and refining the dictionary
- Accessing the help contents and program description (About)

Although, we will only elaborate thoroughly on the three first ones since they are crucial functions to the program and only discuss briefly the other two.

1.2) Viewing and Editing the File List

Visual Overview:

Figure 5: Edit File List Window



As a basic overview, this option enables the users to add and remove files from the Source Code Generator's working directory. As you can see from the illustration above, the user can browse through directories and only relevant *.c files will appear in the file list box under it, where he gets the option to add a file in the file list box on the right which is the program's working directory. Note that if there are any *.h files in the directory the user wishes to add a file from, they will all be transferred as well into the working directory, in case one of the c-files might need it. If the file already exists, the GUI will give the option to overwrite it or not. Finally, the user can remove the file from the working directory which permanently removes the file from the directory, but not from the system.

Functional Overview:

Figure 6: Code Snippet of the File List Window

```
{
    int i = 0;
    while(i < FileListBox1->Items->Count){
        if(FileListBox1->Selected[i]){
            char file_path_from[50];        /* path to source file.  */
            char file_path_to[50];          /* path to target file.  */
            FILE* f_from;                   /* stream of source file. */
            FILE* f_to;                     /* stream of target file. */
            char buf[MAX_LINE_LEN+1];       /* input buffer */

            sprintf(file_path_from, "%s/%s", Edit2->Text , FileListBox1->Items->Strings[i].c_str());
            sprintf(file_path_to, "C:/Work/%s", FileListBox1->Items->Strings[i].c_str());

            /* open the source and the target files. */
            f_from = fopen(file_path_from, "r");
            if(f_from == NULL) perror ("Error writing data");

            else{
                fgets(buf, MAX_LINE_LEN+1, f_from);
                fputs(buf, f_to);
                }
        fclose(f_from);
        fclose(f_to);
        else i++;
    }
}
```

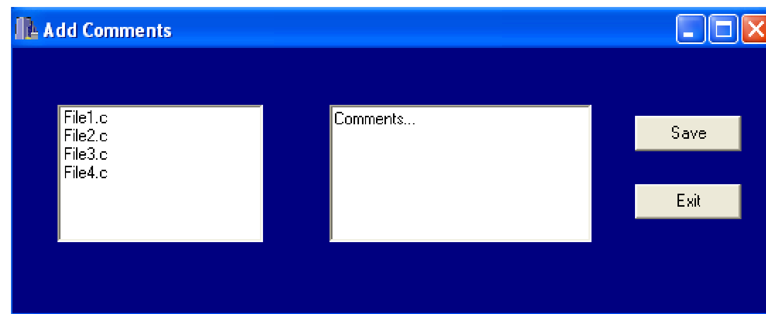
This illustration above is part of the code to add a file to the file list; we believe it is relevant to demonstrate it because it is the template of how the manipulation and transactions of files and strings are accomplished throughout most of the GUI implementation. In other words, this type of coding is frequently reused in other GUI options. As you can see, we basically iterate through the items in the file list box we wish to add a file from and as soon as it detects which file has been selected to add, it creates the appropriate character arrays, which will contain the file paths and creates input/output file streams. The output file stream, the file selected for adding, serves as a sort of intermediate file which is read from and the input file stream is created and then written to. This means that the file is not actually moved or added, in fact, it is copied in such a way that the output file copies its contents into a created file, the input file. Finally, the files are closed and the working directory is updated accordingly with a new copy of the file.

Removing a file takes place in a similar fashion as adding a file, but without having to actually create input and output streams. It simply iterates through the file list box which is the working directory and removes the selected file from the directory, but not from the system as we mentioned earlier.

1.3) Adding Comments Manually to Files

Visual Overview:

Figure 7: Adding Comments Window



Here, the user fills in comments in the memo box that he or she wishes to add to a selected file. The comments are then appended to the end of the file in the form of comments and can be added to later on if desired. The files shown in the left window are the actual files in the working directory.

Functional Overview:

Figure 8: Code Snippet of the Adding Comments Window

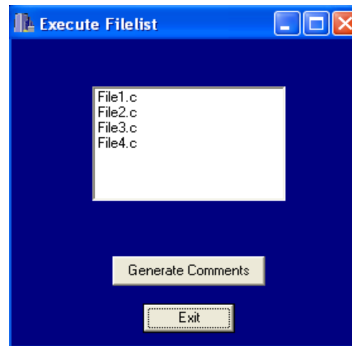
```
if(Memo1->Lines->Count == 0){
    MessageBox(0,"The comment field is empty, please add text to it.", "Notice", MB_OK);
    return FAdd->Show();
    }
else{
    int j=0;
    while(j < Memo1->Lines->Count){
        sprintf(buf, "/* %s */", Memo1->Lines->Strings[j].c_str());
        fputs(buf, f);
        j++;
    }
}
```

The reasoning behind the selection of this code is simply because it represents the actual act of adding comments to a file stream “f”. In the first part, it ensures that the user has actually entered text and if he or she did then the comments in the memo box are read line by line and added to a temporary buffer array, which then outputs the lines in the form of comments at the end of the selected file. These comments are delimited by “/*” and “*/” to ensure that the comments remain as comments and do not interfere with the actual code in the file.

1.4) Generating Comments on a File List

Visual Overview:

Figure 9: Execute File List Window



This option is the core component of the Source Code Generator. It is the window that calls the actual program on the files selected in the window above. Once the user has made a final decision on which files he or she wants to generate comments for, they simply click on the Generate Comments button and voilà, problem solved.

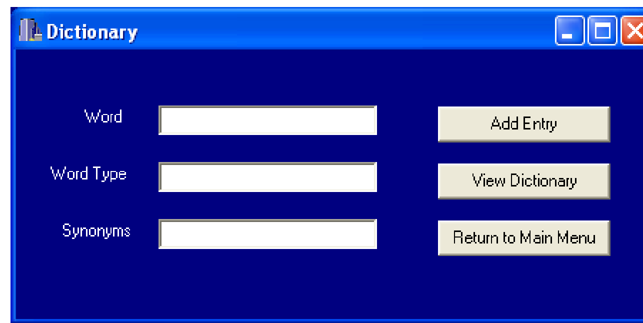
Functional Overview:

Basically, what really happens during the comment generation is that the GUI calls AiMerger.exe on each individual file in the file list that was selected by the user, which will merge into a single output file using the system command "call". Afterwards, it calls the AiParser.exe on the output file which will then contain the relevant automated comments.

1.5) Consulting and Refining the Dictionary:

Visual Overview:

Figure 10: Dictionary Window



This option is a key component for the AI module, since it permits it to refine its knowledge database by consulting a dictionary entries added by the user. So basically, the user fills out the word field with a variable or word he or she desires to add to the dictionary. Then the user fills out the word type in order to define its type which gives a little more insight about the word. Finally, the user fills out the synonyms filed with one or many actual definitions of the variable or word that was originally entered. So for example, the user would enter "i" in the word field, along with "noun" in the word type field and finally in the synonyms' field the user would enter "count increment", which would indicate the AI that is consulting the dictionary that the letter "i" stands for an increment. Nevertheless, a user cannot add entries into the dictionary if all fields are not filled out.

Functional Overview:

Figure 11: Code Snippet for the Dictionary

```
{
    char text1[50];
    char text2[50];
    char text3[50];

    sprintf(text1, "%s", Edit1->Text.c_str());
    sprintf(text2, "%s", Edit2->Text.c_str());
    sprintf(text3, "%s", Edit3->Text.c_str());

    if(strlen(text1) == 0 || strlen(text2) == 0 || strlen(text3) == 0){
        MessageBox(0,"You have not entered the fields correctly!","Warning", MB_OK);
        return FDic->Show();
    }
    {...}
    fputs("\n", f); fputs(text1, f); fputs("\n", f);
    fputs(text2, f);
    fputs("\n", f); fputs(text3, f); fputs("\n", f);
    {...}
}
```

Basically what happens here is that, the three strings entered by the user are stored in respective character arrays and are then evaluated for emptiness, in which case a warning message is issued. Otherwise, the text is appended to the dictionary, which is a text file.

The text is appended in the following form:

1) Word = a, Word Type = b, Synonyms = c;

2) Word = d, Word Type = e, Synonyms = f, g;

------------- (This represents an empty space)
a
b
c
------------- (Empty space)
------------- (Empty space)
d
e
f, g
------------- (Empty space)

1.6) Consulting the Help Contents and About

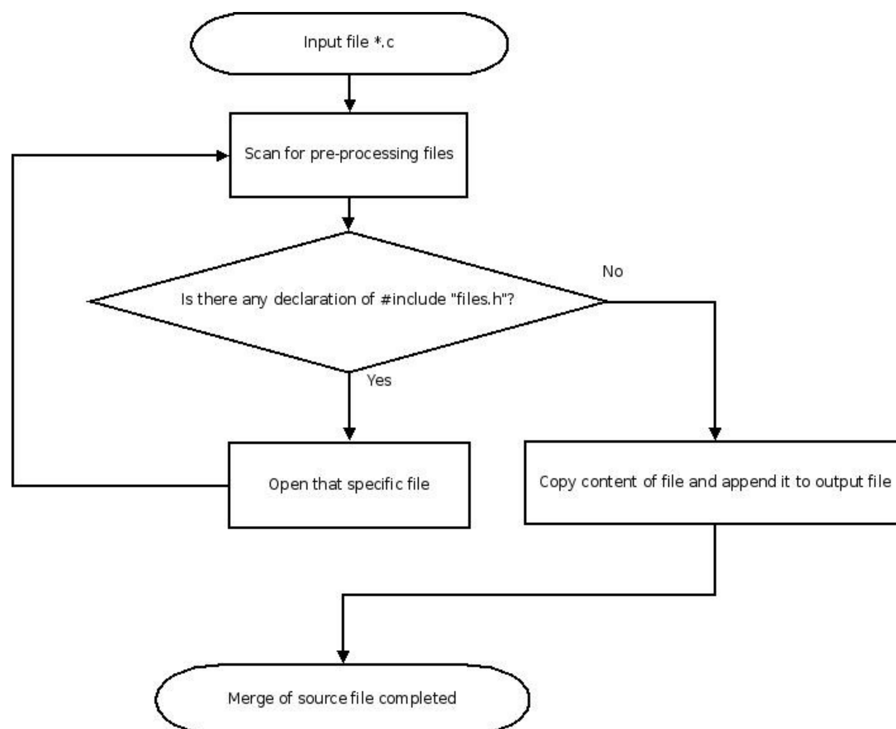
This section simply comprises of a simple help menu which explains the usage of different functionalities of the program such as the ones mentioned above. Finally, the about option gives the user a short description of the program version and the team responsible for its implementation.

CODE MERGER

The code merger consists basically of regrouping together all the required input files to comment. This primarily facilitates the task of the parser developer. This way, the parser has only one single file to analyze. Furthermore, the artificial intelligence is able to obtain the definitions of multiple functions, variables, etc. not included in the selected dictionaries.

The required files to combine with the initial input file to comment have the same pattern. They can be found in the textual code of the input file as “#include .h” where “*” represent any name of file. These files are called pre-processed or header files and contain additional relevant information about variables and functions used in the input file but not necessarily defined in it. A demonstration of the text parser can be found in the appendices. It is compiled under the Borland[1] C++ environment. The functionality of the merge step is fairly simple and can be found in the flow chart of figure 5:

Figure 12: Flowchart of the Coder Merger



TEXT PARSER OUTPUT

There are two main goals for developing a textual output to the parsing step. The first goal is to test and prove the correct functionality of the parser. The evaluators of the project can therefore validate the correct functionality of the parser and identify faults of implementation or weaknesses if needed. The second one is to provide a testing tool to the artificial intelligence developers to work with in the future. This way, they will be able to understand better the structure created by the parser in order to use it in their artificial intelligence analysis and modify the parser if necessary. As example, they can take their own self-created C-language code, parse it and see the result in the output text file.

The structure created by the parsing process is either in memory or on a specific file on the drive. However, to show its content textually, it is necessary to design code that will nicely translate the structure and content of the parsed file. Code spaces, functions, variables etc. need to be listed in a predefined order attached to their respective groups. A demonstration of the text parser output can be found in the appendices for the reconstruction of the text file part only.

CODE PARSING

3.1) Overview

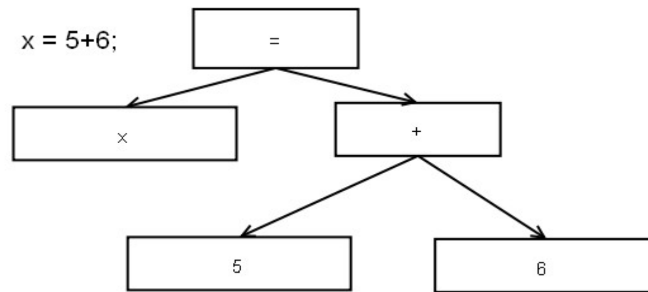
A parser is a program that takes text input, for example source codes files, analyzes it as a whole and tries to break up it's contents into single elements. They are typically found in compilers or command interpreters. It is only after everything has been separated, stored and identified that the data becomes useful for interpretation in following steps.

In the case of our C parser, everything is done in one pass. The code is read from left to right, top to bottom only once and everything is extracted. The complexity in performing parsing of the sort lies in determining the order of evaluation in which expressions will be stored. Since our type of code is mainly made out of complex expressions, it is convenient to store them in the order and grouping that their evaluation will take place to be easily able to tell what happens within it. For example, an expression such as 'x = 5 + 6' should be stored as '5 + 6' and 'x =' to that, instead of 'x = 5' and '+ 6' to that. Because of this, the parser must perform the task of grammatically interpreting the complete meaning and type of each element in expressions. If we think in terms of the AI comment generator, it is for it a helpful step in the right direction. It will immediately be able to tell that 5 is added to 6, and that this result will then be stored inside of x.

3.2) Parser Structure

Every expression will be stored in the form of a tree for easy access and for the convenience that a tree grants us by being expandable to infinity both sides as we keep constructing the expression. For clarity, an expression such as the one above would be stored as follows:

Figure 13: Parsing Tree Structure Example



As can be observed, the operator is always stored in the middle, whereas the operands on the left side are stored in the left leaves and the ones on the right side are stored in the right leaves. Of course, expressions that are huge in size such as 'int x = c^5 – (fn1(y+7)*8 – (5+fn2/fn3(z^3, y, y*z)) + 18;' are supported. The tree produced will be huge, but a properly coded interpreter for it such as the AI engine will understand it just as it would understand a simple expression. Each element in a tree is stored in a node, which is actually a structure containing many fields:

Figure 14: Code Snippet of the Node Structure

```
// each element entry

struct node {
 int value; // in case of number
 int flags; // special attributes

 int  token; // type
 int  opcode; // sub-type
 int  prefix; // int, return, case...

 struct node *lleaf; // pointer to left leaf
 struct node *rleaf; // pointer to right leaf

 struct node *next; // pointer to next expression (statement list)
 struct node *block; // pointer to block
 struct node *args; // pointer to argument list
};
```

With these fields only, we are able to fully define any element in C that our parser can identify. If, for example, we had to store the expression 'getTime(1)' here which is a function call, we would directly store it as follows:

Figure 15: Code Snippet for Storing 'getTime(1)'

```
struct node func;
struct node arg;

func->token = TOKEN_FUNCTION;
func->opcode = 5; // assuming 5 is the index of 'getTime' entry in symbol table
func->args = &arg;

arg->token = NUM;
arg->value = 1;
```

If, for instance, the expression 'return 0' followed this call, then its node would be linked as the next statement in 'func->next' which makes it the statement list. All statements are linked this way together and define the sequence of the entire code. In this fashion, the code can be reconstructed entirely as it was, in the order that the expressions were listed in the source file, and in the same exact way because every element in expressions is identified and stored properly to allow reconstruction.

Another thing that the parser does is grouping useful information together for the upcoming AI analysis. The two types of information that are collected and grouped are global variables and functions. They contain relevant information such as the type, return type and arguments list.

All of the data that will be used by the AI is finally dumped into files, with the assisstance of which subsequent work can ensue. Firstmost, as a preview of the correctness of the work accomplished by the parser, the entire contents of the statements linked list of expression trees are traversed and printed out to provide the exact copy of the original source code, white spaces and extra line feeds apart. The symbol

table that contains all of the names encountered during the parsing pass dumps all it's entries one after the other as the following structure:

Figure 16: Code Snippet for the Symbol Table

```
struct entry {  /*  form of symbol table entry, with name directly stored  */

 char name[64];
 int  token;
 int  opcode;
};
```

There is space for a 64 byte null-terminated string for the name, the 'token' or type of the entry to distinguish between things like operators or simple names, and an 'opcode' field to further specify the type. For example, special entries may look like this, and they are all defined in the source :

```
{ "int", TOKEN_PREFIX, OP_INT },
{ "+", TOK_OPERATOR, OP_PLUS },
```

In the case of a simple name entry, the token 'ID' is used. Each entry is 72 bytes apart in the file assuming an int is 4 bytes on the machine used and the extension '.symbols' is appended to the '.c' filename.

The next thing to be dumped is the expression tree contents and the statement list that will allow reconstruction of the code. The entire array of nodes is dumped as is and can be accessed by using the node structure defined previously. To retrieve the names of these elements (if such is the case), the 'opcode' field can be matched with a corresponding 'opcode' field in the symbol table. One thing that creates some incompatibility, however, is that pointers remain the direct memory addresses that they were in the parser memory space. To remedy this, the first 4 bytes (sizeof(pointer)) in the dump file will be the head pointer at which the whole array was originally stored in memory. The dump of the tree nodes proceeds right after that pointer. To use any of the pointers contained in the dump (next, block, args), this value must be first subtracted from it and the new memory base pointer into which the dump

was copied to must be added. Each entry is 40 bytes apart in the file assuming an int is 4 bytes on the machine used and the extension '.tree' is appended to the '.c' filename.

Out of all these nodes, several few are extracted and stored inside of two other files to group useful information together. All global variables are dumped to a '.vars' file. They are the same nodes as above with the only exception that they are stored together sequentially. Whenever the AI encounters an unidentified variable, all it has to do is look it up in this dump. The second file is the '.functions' file which lists all the encountered defined functions as nodes. They contain pointers to 'block' and 'args' which link to the statements and arguments that the function contains. To access these, it is required to convert the pointer into the '.tree' dump using a similar procedure to the one outlined above. It would probably be a good idea to create accessor functions for clear and reusable pointer conversion at this point. These two files are structured just like the '.tree' file, so they contain the base pointer in the first 4 bytes which is followed by the dump of the nodes. As is apparent, the AI engine has now access to a lot of data with which it can work with. Since nothing is missing either, it possesses all the data that it may possibly need for analysis, during which, it can of course also build on top of it or perform any other task with.

3.3) Extended Parser Details

The following topics of discussion are about the parser itself in more detail. The parser was built from a fundamental base suggested in the COMPILERS[4] book. A reading of chapter two is recommended for better understanding of the code if the purpose is to expand upon it. Over the basic code sample in the book, the parser has been further improved with significant grammar extensions, statement storage in trees and print outs through tree traversals. Some of the things incorporated include operator priorities, parenthesis support, function definitions and declarations, argument lists, variables and their types, complex expressions, conditional and switch statements, nesting, comments, and more. Some things which are not yet supported are arrays, strings, conditional statements without curly brackets, decimal point constants and preprocessor statements.

The parser continuously loops recursively on its element-extraction code, following certain branches depending on the currently evaluated element type and the 'lookahead' which is the element that lies ahead. Based on these two things, it is able to fully traverse the code in a single pass. During this phase, it also stores everything inside of trees during a careful linking process where current, past or future left and right leaves are selected as containers. The exact choices of leaves were determined experimentally as each feature was implemented. Upon adding more functionality, a careful examination and understanding of the current storage mechanism is necessary. Each element has to be stored in a different way, and possibly needs to be rearranged furthermore as the expression progresses. When adding more functionality, it is recommended following the code in debug mode and observing the entire construction phase. Alterations can then be made to store things in the right place, and for them to work in all possible cases. More information is provided with the code in the form of comments.

---

4 V. AHO, Alfred. SETHI, Ravi. D. ULLMAN, Jeffrey. Compilers: Principles, Techniques and Tools

CONCLUSIONS

I) Summary and Opinions

The tasks assigned to us as well as the goals to achieve have been well respected overall. However, some troubles were encountered during the development process for many reasons. To name a few, the development under GCC[2] or Borland[1] compilers instead of Microsoft Visual Studio[5] made the work harder for not being able so use the MFC[6] libraries. Also, the design of the parser was at first not the design expected by Prof. Vybihal, so that it had to be redesigned to fit the requirements.

Overall, it is important to acknowledge that this is a long term project that may take several years before being truthfully realized. The longest part of the project is certainly the AI section since a lot of work needs to be done in order to generate intelligent and precise comments.

However this project helped up to learn a lot about parsers that are used in some other applications such as compilers. Also, this kind of project is mostly new to the software world and is very promising. It was therefore pleasant in that way to work on this particular project.

II) Possible Enhancements

There are many possible enhancements that can be done to the project. However, for now, the most important task is to focus on the AI implementation for the next teams. We could extend the Source Code Generator to the C++ and C# languages since they share many similarities with the C-language. However, these languages are much more advanced than C and they possess deeper concepts related in particular to object oriented programming that would require extended implementation and work by the parsing and AI developers.

5 Microsoft Visual Studio. http://msdn2.microsoft.com/en-us/vstudio/default.aspx
6 Microsoft Foundation Classes. http://en.wikipedia.org/wiki/Microsoft_Foundation_Classes

REFERENCES

V. AHO, Alfred. SETHI, Ravi. D. ULLMAN, Jeffrey. Compilers: Principles, Techniques and Tools. Addison-Wesley; US Ed edition. January 1, 1986. 796pp.

The GNU Compiler Collection. GCC. Version 4.2.2 http://gcc.gnu.org/

Borland C++ Compiler. Version 5.5 http://dn.codegear.com/article/20633

Elsa: An Elkhound-based C++ Parser. Version 2005.08.22b http://www.cs.berkeley.edu/~smcpeak/elkhound/

Microsoft Corporation. Microsoft Visual Studio. http://msdn2.microsoft.com/en-us/vstudio/default.aspx

Wikipedia Organization. Microsoft Foundation Classes. http://en.wikipedia.org/wiki/Microsoft_Foundation_Classes

APPENDICES

APPENDIX I: DEMONSTRATION OF CODE MERGER

jax2.c input.c jax.c

```
#include <mustafa.h>

int addition(int x, int y)
{
    sum = x+y;
    return sum;
}

#include "jax.c"

int main()
{
    int x = 1;
    int y = 1;
    return addition(x, y);
}
```

Display Quick Info. jax2.c

```
#include "jax2.c"

int jax(float xor, double xir)
{
    return power(xor,xir);
}
```

input.c jax.c jax2.c

```
double jax2_power(double r, double z)
{
    return r^z;
}
```

output.c

```
//$$content of file input.c

int addition(int x, int y)
{
    sum = x+y;
    return sum;
}

//$$content of file jax.c

//$$content of file jax2.c

double jax2_power(double r, double z)
{
    return r^z;
}

//$$end of content of input file jax2.c
int jax(float xor, double xir)
{
    return power(xor,xir);
}

//$$end of content of input file jax.c
int main()
{
    int x = 1;
    int y = 1;
    return addition(x, y);
}

//$$end of content of input file input.c
```

APPENDIX II: DEMONSTRATION OF PARSER RECONSTRUCTION

Example input.c file:

```
int global1 = 5;


// this is a comment
float global2 = 18;

int getTime(int dst);
int getSwitch(int s);

/*this is also a comment
float global3 = ??;
*/

int getFile(int x, float y){
	int z;
	int i = 0;

	while (i < 5)
	{
		z = x-(y*2-(10+y)+2*2) + getTime(1) + getSwitch(i); i += 1;
	}

	return z;}

int getTime(int dst)
{
	if (dst>=1)
	{
		return getTickCount()+60*60*1000;
	}
	else if (dst == 0)
	{
		return getTickCount();
	}
	else
	{
		return 0;
	}
}

int getSwitch(int s)
{
	switch(s)
	{
		case 0:
		s +=5;
		break;
```

```
                case 1:
                return 5;
                break;
                default:
                s*=2;
        }

over:
        return s;
}
```

Example output:

```
int global1=5;

float global2=18;
int getTime(int dst);
int getSwitch(int s);

int getFile(int x, float y)
{
        int z;
        int i=0;

        while(i<5)
        {
                z=x-(y*2-(10+y)+2*2)+getTime(1)+getSwitch(i);
                i+=1;
        }
        return z;
}

int getTime(int dst)
{

        if(dst>=1)
        {
                return getTickCount()+60*60*1000;
        }

        else if(dst==0)
        {
                return getTickCount();
        }

        else
        {
                return 0;
        }
}

int getSwitch(int s)
{

        switch(s)
        {
                case 0:
```

```
                s+=5;
                break;
                case 1:
                return 5;
                break;
                default:
                s*=2;
        }
        over:
        return s;
}
```