SSL:

String Searching Language

Authors:

Meera Ganesan (meera.ganesan@intel.com)
Dennis Kim (dkim@harris.com)
Sandy MacDonald (sandymac@att.com)
Satheesha Rangegowda (satheesha_rangegowda_923@agilent.com)

Table of Contents

An Introduction to SSL

SSL, which stands for String Searching Language, is intended to be an intuitive string searching and manipulation language. The straightforward, but powerful commands enable users to quickly create programs to perform string manipulation on files.

Programmers will enjoy the compactness of SSL that allows them in several lines of code to perform functions that would have taken many lines of C, C++, or Java. The succinct, but natural character of the language allows users to become productive quickly and to easily debug their SSL code.

In addition to the ability to run SSL programs in a standalone manner, the compiled output of SSL programs can also be called from an existing C, C++, or Java program. The result is comparable to extending those languages to include the string searching and manipulation functions so often required by the serious programmer.

Background

Many forms of computer processing require the searching of text strings in files. Unfortunately, powerful programming languages such as C, C++, and Java are designed to build data structures & algorithms from the scratch, starting from primitive computer elements like bytes & words. String searching in these languages is time consuming and code intensive.

While scripting languages often provide the powerful string manipulation facilities desired, they are generally designed for domain specific, rapid application development. They are not intended for writing applications from scratch and as such often lack the flexibility and function of their more verbose cousins (e.g. C, C++, Java).

The thought is that combining the power and flexibility of these programming languages with the concise, productive abilities of the scripting facilities would be a powerful, productive combination. The opportunity exists for a language capable of filling this void.

Voila – SSL!

SSL is designed to provide an easy to use, concise language for string searching and manipulation that can be used in a stand-alone mode or called from an existing program.

Since useful programs can be created with only a few lines of code, SSL programming is easy to learn. New users will appreciate the intuitive, English-language-like format of the language commands. Capable string manipulation functions of the language include search, replace, output, and print options. The succinct nature of the language facilitates debugging.

SSL allows the user to search for particular string or string concatenation in a specified file, directory, or directory and all subdirectories. When found, the indicated line can be printed or written to a temporary file for further processing. The string replace feature allows the user to look for a particular string in a given document and replace with another string. The programmer has access to the matched string, the portion of the line before and after the string, the line number, and the entire line value.

Where would this be useful?

If you haven't encountered a need to perform complicated string searching and manipulation, consider yourself fortunate and somewhat unique! There are many applications for this function.

One area where such a facility would prove extremely useful is for Natural Language Processing where the need to create programs that identify, extract, and manipulate strings are a regular occurrence. Other potential user of "SSL include system programmers, technical writers, e-book editors, library maintenance system administrators, and many others.

Goals of the Language

The goals of SSL are to provide an intuitive language for string processing which can be incorporated into existing programs. The language is intended to support both new and experienced users, and to be powerful, productive, portable, and performance oriented.

Quick Startup

Since SSL commands are designed to be natural and intuitive, new programmers can quickly establish adequate knowledge of the language and become productive. The concise format of the language allows short programs of a few lines to perform useful functions. These programs can be implemented quickly and debugged easily.

Powerful

SSL provides the ability to perform string identification, manipulation, and retrieval with a very succinct programming structure. Many possible permutations can be expressed in a compact statement. The result is that programs consisting of just a handful of lines can simulate functions that would have taken many lines of C, C++, or Java code.

Productive

Since SSL programs can be run as standalone programs or called from inside a C, C++, or Java program, it enables these programs to perform process whole files with a small set of instructions that can be fully tested outside of the calling program. The reduced number of lines of code and associated quality of the resulting code will support significant productivity gains for programmers in these targeted languages.

Portable

Since the compiled output of SSL is standard C++, programs written in this language can be compiled and run on a variety of platforms (any of which run standard C++) including Sun Solaris, HP Unix, and IBM AIX. Care has been taken with the language to eliminate the use of processor specific C/C++ code.

Performance-Oriented

Rather than generating interpretive or byte code results, the SSL compiler generates a C++ language program. Since C++ programs compile on each processor to among the most efficient of possible machine languages, the resulting SSL code runs quickly and without undue impact of system resources. Further optimization of the C++ code by the local optimizing compiler ensures a performance-oriented implementation of the desired string functions.

Summary

In summary, the String Searching Language SSL is intended to fill the void left by otherwise powerful programming languages where the lack of flexible string searching and manipulation facilities that can be implemented easily, tested quickly, and run in any environment requires the developer to create and test many lines of code unique to the particular problem. We believe that programmers will appreciate the benefits of this language and find it useful in many real world coding situations.

Tutorial

Hello World in SSL

Getting started with the SSL language is fun and easy! Lets start with the classic "Hello World" program. In SSL, this can be a single line program as follows:

Example:
```
Print "Hello World";
```

Compiling a Program in SSL

Lets call our program "Hello". This program would be saved with a .ssl file extension and compiled using the SSL compiler using the program "sslc".

Example:
```
sslc Hello.ssl
```

The result of main would be the Hello.cpp file. The Hello.cpp file is compiled using g++ and linked with SSLLib.cpp to produce an executable C++ program.

More Interesting Examples

To find the string "Hello World" in an input file (input.txt) and print the line number for each line on which it was found, create the following SSL program:

Example:
```
String string1 = "Hello World";

Find string1 in "input.txt" {
  Print LineNum;
};
```

To send the output to a file called "output.txt" and show the whole line, we would alter the program slightly as follows:

Example:
```
String string1 = "Hello World";
File fileo = "output.txt";

Find string1 in "input.txt" {
  Output to fileo LineValue ;
};
```

Now we'll try something a little fancier. The following program searchs for "Hello World" in directory "C:/temp" and creates an output file indicating the file name, line number, and the text of each line including "Hello World. The search is performed with no case sensitivity:

Example:

```
String string1 = "Hello World";
Directory mydir = "C:/temp";
File fileo = "output.txt";

Find string1 in mydir using CaseOff {
  Output to fileo FileName ' ' LineNum ' ' LineValue;
};
```

To replace the phrase "Hello World" with "Greetings Earth" and print the lines on which we made the replacement:

Example:

```
String string1 = "Hello World";
String string2 = "Greetings Earth";

File filei = "input.txt";
File fileo = "output.txt";

Replace string1 in filei with string2 using CaseOff {
  Print LineValue;
};
```

For our last example, we will create a program that combines both the find and the replace operators.

Example:

```
String string1 = "Hello World";
String string2 = "Greetings Earth";
File filei = "input.txt";

Find string1 in filei using CaseOn, SubDirectoryOff {
  Print LineValue;
  Output to "result.txt"  LineValue;
}

Replace string1 in filei with string2  {
  Print LineValue
  Output to "result.txt" LineValue;
 };
```

Congratulations! You are now an SSL expert!

Language Reference Manual

Introduction

SSL, String Searching Language is designed to do basic string operations, string manipulation and mainly string searching. This Manual discusses the primary characteristics of SSL.

Note: Syntax is indicated in italics with keywords in bold. The vertical bar indicates multiple choices.

Lexical conventions

SSL has the following tokens: comments, identifiers, keywords, strings, constants, and separators. In general, tokens are greedy in that the longest string of characters that constitutes a valid token is the one selected. Tokens are also case sensitive unless specifically noted.

Comments

SSL supports both C (/* */) and C++ (//) style comments like many other contemporary languages.

C style comments begin with the character sequence /* and end with the opposite sequence */. Multiple such sequences may appear on a single line. Alternatively, this sequence may span multiple lines. The SSL compiler removes all comments prior to processing the SSL program input.

C++ style comments begin with // and terminate at the end of the line as identified by the appropriate end of line characters for the machine (e.g. \r \n, \n, or \r). Again, the SSL compiler removes all comments prior to processing the SSL program input.

Keywords

Keywords are reserved and include the following case sensitive values:

AfterPattern	BeforePattern	
CaseOn	CaseOff	Char
Directory	DirectoryName	
File	FileName	Find

in	Int	
LineNum	LineValue	
Output		
Pattern	Print	
Replace	ReplaceLine	
String	SubDirectoryOn	SubDirectoryOff
using	with	

White Space

The following characters are denoted as white space by SSL:

	blank
\n	newline
\t	tab
\r	carriage return

White space is mainly ignored by SSL except in the case of comments where it is used to terminate C++ style (//) comments. However, there are certain situations in which it is significant. Specifically, white space is of interest when included as part of a string (e.g. one which includes blanks) or when its placement results in the termination of a token (e.g. results in the end of an identifier or an integer).

Separators

The following separators are recognized by SSL: , ;
(comma, semicolon).

Block Identifiers

The following identify the beginning and end, respectively, of blocks in SSL: { }
(Open braces, Close braces)

Constants

The following are the constants supported by SSL.

Integer constants

Integer constants consist of a sequence of one or more consecutive digits.

Character constants

Character constants consist of one alphabetic (a – z, A – Z), numeric (0 – 9), or printable special character (e.g. # * + ? !) enclosed in single quotes (‘ ‘). These characters must fall in the ASCII range of \33 to \126

Limited unprintable characters are also supported including:

blank
tab

String constants

A string constant is a sequence of one or more characters surrounded by double quotes (“”). These characters may include alphabetic (a – z, A – Z), numeric (0 – 9), and printable special characters from the ASCII range of \3 to \377. A double quote may appear within a string but must be accompanied by another double quote (e.g. “”). SSL will remove the second double quote and process the double quote as part of the string.

Identifiers

Identifiers consist of a sequence of letters (a - z, A - Z) and digits (0 – 9) where the first character must be a letter. Note that SSL is case sensitive so identifiers created with the same sequence of letters and digits but differing in case, will not be recognized as the same identifier.

Various specific types of identifiers are supported including the following:

String Identifiers

SSL allows a string constant to be identified and referred to in later processing.

Syntax : ***String*** *string1 = “abcde”;*
Find *string1* ***in*** *“file1.txt”* ***{...};***

File Identifiers

SSL allows a file to be identified and referred to in later processing.

Syntax: ***File*** *file1 = “test2.txt”;*

***Find** string1 **in** file1 {...};*

Directory Identifiers

SSL allows a directory or files of a particular type in a directory to be identified and referred to in later processing. Files of a particular type in a directory can be identified by including the

Syntax : ***Directory*** *mydir = "c:\temp";*
***Find** string1 **in** mydir {...};*

Additional Language Specifications

Model of Computation

SSL's model of computation assumes that files in one or more directories are sequential read and processed one after another.

Storage Classes

SSL supports static and automatic storage types since tokens can have a life associated with the entire program or with the processing of a single complex function.

Type Conversion

SSL will automatically convert to a string any combination of strings, characters, and integers appearing sequentially separated only by spaces.

Operators

SSL supports the assignment operator (=) in select places in the language. Please refer to the Basic and Complex String Operations for examples.

Example: *string = expression ;*

Statements

SSL programs can consist of multiple expressions separated by a semicolon (;).

Example: *string = expression ;*
expression ;
expression ;

Scope

SSL identifiers and constants are statically scoped. Expressions are processed sequentially by SSL. Files created as the result of one expression can be used as input by a subsequent expression. Values associated with a block (e.g. values generated by a particular Find or Replace operation are valid only until the end of that block.

Example: ***Find** string_a **in** dir1 **{***
 ***Print** DirectoryName '/' FileName ' ' LineNum ':' LineValue;*
 };

String Operations

Concatenation

This command will concatenate an unlimited combination of string constants, string identifiers, integer constants, integer identifiers, character constants, and character identifiers (including white space characters) into a single string.

Syntax: *value1 value2 value3 ...;*

Where values can be any combination of strings, string variables, integers, integer variables, characters, and character variables including white space characters and character variables with white space values.

Example: ***String** string1 = 'a' 'b' 'c';*

Will assign the value "abc" to string1.

Example: ***String** test1 = "String Searching";*
***String** test2 = " Language";*

***String** test3 = test1 test2;*

test3 will contain "String Searching Language"

Example: ***Print*** *"String" ' ' "Searching" ' ' 3 ' ' "Way" 's';*

Will print "String Searching 3 Ways" with blanks separating the first 4 values.

Print

This command will print a string or concatenation (see Concatenation) to the console.

Syntax: ***Print*** *value1 value2 value3 ...;*

Where values can be any combination of string constants, string variables, integer constants, integer variables, character constants, and character variables.

Example: ***String*** *test1 = "String Searching";*
String *test2 = " Language";*

Print *test1 test2;*

Will print "String Searching Language"

Output

This command will write a string or concatenation (see Concatenation) to the indicated file.

Syntax: ***Output to*** *file value1 value2 value3 ...;*

Where file can be a file constant or file identifier and values can be any combination of string constants, string variables, integer constants, integer variables, character constants, and character variables.

Example: ***String*** *test1 = "String Searching";*
String *test2 = " Language";*

Output to *"file1.txt" test1 test2;*

Will write "String Searching Language" to file1.txt.

ReplaceLine

This command is only valid within the scope of a Replace statement. It causes the line with the matching pattern to be completely replaced with the indicated string value.

Syntax: ***ReplaceLine*** *value1 value2 value3 …;*

Where values can be any combination of string constants, string variables, integer constants, integer variables, character constants, and character variables.

Example:
```
Replace string_a in file1 with string_b {
  ReplaceLine BeforePattern '*' Pattern '*' AfterPattern;
};
```

Will replace any file line containing the search pattern with the same line but with '*'s before and after the matching pattern.

Find

This command will find each instance of the given string in a given file and allow the directory name (if directory search), file name, line number, and line value associated with each to be utilized within the block of statements associated with the find request.

Find has the following required and optional parameters as follows:

Syntax:
```
Find string in file|directory using optionalkeyword(s) {
        (statements)
};
```

Required:
1. The ***Find*** keyword.
2. The string being searched for. This can be specified statically as a string (e.g. "abc") or using a previously defined string identifier (e.g. string2).
3. The ***in*** keyword.
4. The file or directory to be searched. The file or directory can be specified statically as a string (e.g. "file1.txt" or "c:\temp\test1.txt" or

"c:\project\*.cgi") or using a previously defined file or directory identifier (e.g. file1, directory_abc).

Optional:

5. The ***using*** keyword.
6. An optional parameter indicating whether the search should be case sensitive CaseOn or not (CaseOff). Note: the default is case sensitive.
7. An optional parameter indicating whether a directory search should be confined to the current directory (SubDirectoryOff) or inclusive of all sub directories (SubDirectoryOn). Note: the default is to search all subdirectories. Note that this keyword has no meaning for file searches and is ignored.
8. If both CaseOn|CaseOff and SubDirectoryOn| SubDirectoryOff are included, they may appear in any sequence, but must be separated by a comma.

Required:

9. The block initiator *{*
10. One or more statements each followed by a semicolon. These statements may utilize keywords created by SSL for the find operation in progress including:
    a. LineNum – the line number in which the search string appears
    b. LineValue – the text of the line in which the search string appears
    c. FileName – the name of the file in which the search string appears
    d. DirectoryName - the name of the directory in which the search string appears (only populated when searching a directory)
    e. BeforePattern – the portion of the line which appeared before the pattern matched
    f. AfterPattern – the portion of the line which appeared after the pattern matched
    g. Pattern – the pattern matched
11. The block terminator *}*

Syntax1: ***Find*** *string* ***in*** *file* ***{***
Statement(s)
};

Syntax2: ***Find*** *string* ***in*** *directory* ***{***
Statement(s)
};

Syntax3: ***Find*** *string* ***in*** *file* ***using*** *CaseOff* ***{***

```
              Statement(s)
            };

Syntax4:    Find string in directory using SubDirectoryOff {
              Statement(s)
            };

Syntax5:    Find string in directory using CaseOff, SubDirectoryOff {
              Statement(s)
            };
```

The following examples demonstrate possible syntax variations:

In this example, both the file name and string value are constant values. The search will be for the string *abc* (case sensitive by default) in the file *file1.txt*.

```
Example:    Find "abc" in "file1.txt" {
              Print LineNum;
            };
```

In the following example, both the file name and string value are identifiers. Note that CaseOff will result in a non-case sensitive search.

```
Example:    Find string_a in file1 using CaseOff {
               Print LineValue;
            };
```

Replace

This command will find and replace each instance of the given string in a given file and allow the directory name (if directory search), file name, line number, and line value associated with each to be utilized within the block of statements associated with the find request.

Replace has the following required and optional parameters as follows:

```
Syntax:     Replace string in file|directory with string using optionalkeyword(s) {
```

(statements)

};

Required:

1. The ***Replace*** keyword.
2. The string being searched for. This can be specified statically as a string (e.g. "abc") or using a previously defined string identifier (e.g. string2).
3. The ***in*** keyword.
4. The file or directory to be searched. The file or directory can be specified statically as a string (e.g. "file1.txt" or "c:\temp\test1.txt" or "c:\project\*.cgi") or using a previously defined file or directory identifier (e.g. file1, directory_abc).
5. The ***with*** keyword
6. The target string that will replace the search string.

Optional:

7. The ***using*** keyword.
8. An optional parameter indicating whether the search should be case sensitive CaseOn or not (CaseOff). Note: the default is case sensitive.
9. An optional parameter indicating whether a directory search should be confined to the current directory (SubDirectoryOff) or inclusive of all sub directories (SubDirectoryOn). Note: the default is to search all subdirectories. Note that this keyword has no meaning for file searches and is ignored.
10. If both CaseOn|CaseOff and SubDirectoryOn| SubDirectoryOff are included, they may appear in any sequence, but must be separated by a comma.

Required:

11. The block initiator *{*
12. One or more statements followed by a semicolon. These statements may utilize keywords created by SSL for the replace operation in progress including:
    a. LineNum – the line number in which the search string appears
    b. LineValue – the text of the line in which the search string appears
    c. FileName – the name of the file in which the search string appears
    d. DirectoryName - the name of the directory in which the search string appears (only populated when searching a directory).
    e. BeforePattern – the portion of the line which appeared before the pattern matched
    f. AfterPattern – the portion of the line which appeared after the pattern matched
    g. Pattern – the pattern matched
13. The block terminator *}*

Syntax1: ***Replace*** *string* ***in*** *file* ***with*** *string* ***{***
Statement(s)
};

Syntax2: ***Replace*** *string* ***in*** *directory* ***with*** *string* ***{***
Statement(s)
};

Syntax3: ***Replace*** *string* ***in*** *file* ***with*** *string* ***using*** *CaseOff* ***{***
Statement(s)
};

Syntax4: ***Replace*** *string* ***in*** *directory* ***with*** *string* ***using*** *SubDirectoryOff* ***{***
Statement(s)
};

Syntax5: ***Replace*** *string* ***in*** *directory* ***with*** *string* ***using*** *CaseOff, SubDirectoryOff* ***{***
Statement(s)
};

The following examples demonstrate possible syntax variations:

In this example, the file name and both strings are constant values. The search will be for the string *abc* (case sensitive by default) in the file *file1.txt* and for each instance of the search string to be replaced with the target string.

Example: ***Replace*** *"abc"* ***in*** *"file1.txt"* ***with*** *"xyz"* ***{***
Print *LineNum****;***
};

In the following example, both the file name and string value are identifiers. Note that CaseOff will result in a non-case sensitive search, followed by each instance of the search string being replaced by the target string.

Example: ***Replace*** *string_a* ***in*** *file1* ***with*** *string_b* ***using*** *CaseOff* ***{***
Print *LineValue****;***
};

SSL Compiler Architecture

The SSL Compiler translates the SSL source file into C++ Source files.

As shown in Figure 1, the Lexical Analyzer reads a stream of characters from the SSL Source file and creates valid tokens based on the SSL language syntax. If the token is not valid in SSL Syntax, the Lexical Analyzer will generate compiler errors.

The Parser processes the sequence of tokens generated by the Lexical Analyzer and generates an AST (Abstract Syntax Tree) used by later components for processing the SSL program. The Parser generates errors if syntax errors are encountered.

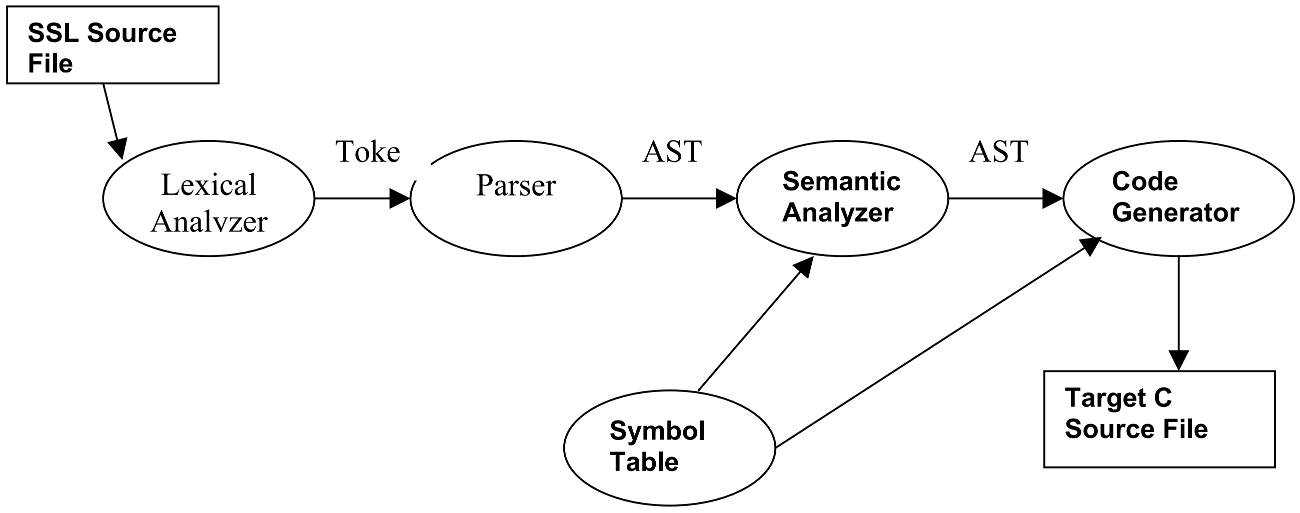

Figure 1:

The Semantic Analyzer takes the AST as input and checks for semantic properties of the SSL Language. It generates entries in the Symbol table for new identifiers and ensures that previously defined identifiers exist. It will generate error messages if the program is recognized to be semantically incorrect.

The Code Generator utilizes C++ libraries to transform the AST into a C++ program (<file>.h and two <file>.cpp files).

The SSL Lexer and Parser programs were generated using ANTLR V7.2.1 based upon SSL grammar rules. The AST is the standard tree generated automatically by the ANTLR tool. The Symbol Table is a custom coded C++ program. Only one Symbol Table is created for each SSL program, hence all variables exist in a common name space. The ANTLR Tree Walker was used to generate both the Semantic Analyzer and the Code Generator since they walk the AST to perform their functions. (See figure 2).

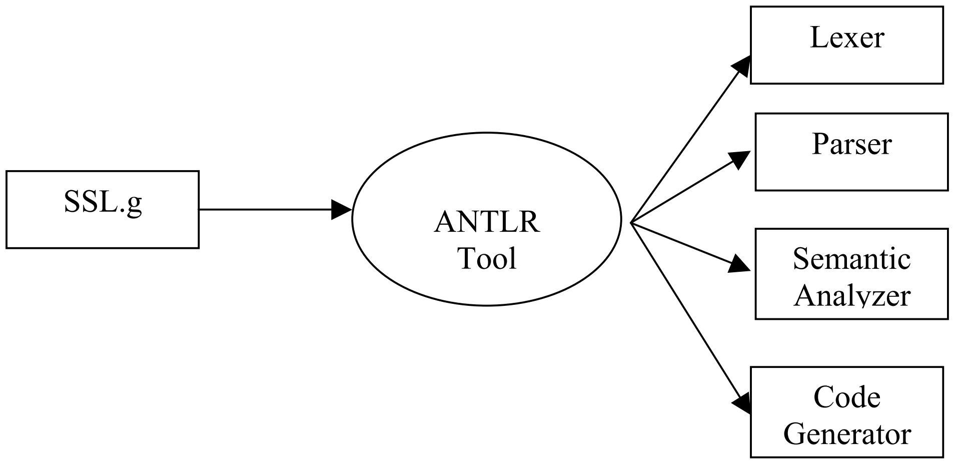

Figure 2:

The source files include the .h and .cpp files necessary to run the program in a stand-alone mode or called from another program. These programs are compiled on the target processor using the standard C++ compiler.

Assignments and Approach

The team approach was to create and test the find function end to end, then to implement the remainder of the functions as independently as possible. The compiler work was divided into the following components with the structure of the AST driving the design interface for the front-end components and tying the front and back ends together. The C++ libraries drove the design interfaces for the back end components.

- Front end: Lexer and Parser
- Front end: Semantic Analysis
- Front end: Symbol Table
- Back end: Code Generation – Tree Walker
- Back end: Code Generation – C++ Libraries

Roles

Project roles and assignments were as follows:

- White Paper – Sandy and Sateesha
- LRM – Meera and Dennis
- Compiler Architecture/Design - Sateesha
- Lexer and Parser – Sandy
- Semantic Analysis – Sateesha
- Symbol Table – Meera
- Code Generation Tree Walker – Sateesha, Meera
- C++ Libraries – Dennis
- Testing – Meera
- Final Report – Sandy

Project Plan

Key project dates:

- 02/20/03 White Paper
- 03/18/03 Compiler Architecture
- 03/27/03 LRM
- 04/17/03 First function
- 04/30/03 Revised first function
- 05/10/03 Full function and Testing
- 05/11/03 Test on Columbia machines

Testing Strategy

- Unit tests were performed by team members for individual deliverables.
- Integration tests were performed by the back-end team during final stages and were necessary to implement language features.
- End to end Function tests were performed and documented. See appendix A for actual documentation.

Lessons Learned

- Nothing could have made this more difficult to work in a teaming environment than to never be able to meet face to face. We met weekly throughout the semester and spent most of the time trying to ensure common understanding.
- The time difference was a significant difficulty since we had team members on both coasts. It made it impossible to communicate during the week except via emails.
- The LRM was an indispensable tool for communications and, in effect, our contract for content. That deliverable helped force many decisions and discussions during the development process not only within the team, but also between the team, the T.A., and the professor.
- The lack of a common code platform that was familiar and comfortable for all team members was a significant impact. It impacted our productivity and eroded our ability to communicate effectively.

Caveats and Defects

- Concatenation function could not be implemented in time for our release.
- ReplaceLine function could not be implemented in time for our release.
- If Print and Output are used in the same Find or Replace block, they will be forced to use the same format.
- A Print or Output within the scope of a Find or Replace block will not support pure string text (e.g. text which does not include any of the system provided values documented in the LRM).
- Compile errors on Columbia Unix machine. More work needed to make compiler portable.

Appendix A: End to End Test Cases

Syntax Error checking

Following are the tests done to find the syntax error from SSL

1. Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing */
String myname = “abc”;
```

Output: test_syn.cpp

```
#include "SSLLib.h"
void main()
{
```

Result: we do not have closing bracket in main if we just the comments in the source file

2. Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing
String myname = “abc”;
```

Output:

```
exception: expecting '*', found 'EOF'
```

3.Input:test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
Print  myname;
```

Output:

```
#include "SSLLib.h"

void main()
{

	std::cout << "abc"<< endl;

}
```

4.**Input:** test_syn_txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
```

```
String myname = "abc";
Print  myname myname1;
```

Output:
unexpected AST node: ConcatValue

5.**Input:** test_syn_txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
Print  myname1;
```

Output:

```
identifier myname1 not defined
```

6.**Input:** test_syn_txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
Print1  myname1;
```

Output
line 4: expecting EOF, found 'Print1'

7.Input:test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File = "test.txt;
Print  myname;
```

Output:

```
line 4: expecting ID, found '='
exception: expecting '"', found 'EOF'
```

8.Input:test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File = "test.txt";
Output myname;
```

Output:

```
line 4: expecting ID, found '='
line 5: expecting "to", found 'myname'
```

9. Input:test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File "test.txt";
Output to myname;
```

Output

```
line 4: expecting ID, found 'test.txt'
line 5: unexpected token: ;
line 6: expecting SEMI, found ''
```

10. Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Output to file1 myname;
```

Output

```
#include "SSLLib.h"

void main()
{

        ofstream outputFile;
        outputFile.open("test.txt",ios:app);
        outputFile << "abc"<< endl;
        outputFile.close();

}
```

11. Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
Output to file1 myname;
```

Output

```
#include "SSLLib.h"

void main()
{
        ofstream outputFile;
        outputFile.open("test.txt",ios:app);
        outputFile << "abc"<< endl;
        outputFile.close();

}
```

12. Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
Output to file1 myname;
ReplaceLine  myname "kzt" "next";
```

Output

line 7: expecting EOF, found 'ReplaceLine'

13.Input: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
Output to file1 myname;

Find myname in file1 using CaseOn, SubDirectoryOff ;
```

Output

```
line 8: expecting RBRACE, found ';'
line 9: expecting SEMI, found ''
```

1**4. Input**: test_syn.txt

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp”;

Find myname in file1 using CaseOn, SubDirectoryOff {
Output to file1 myname;
```

Output

```
 line 12: expecting LBRACE, found ''
line 12: expecting SEMI, found ''
```

15. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
	Output to file1 myname;
}
```

Output

```
line 12: expecting SEMI, found "
```

16. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
	Output to file1 LineNum;
};
```

Output

```
#include "SSLLib.h"

void main()
{


}
```

17. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
	Print LineValue;
}
```

Output

```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {false, "", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

}
```

18. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print Linevalue;
}
```

Output:

```
identifier Linevalue not defined
```

19. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print Linevalue;
        Output to "result.txt" LineNum;
};
```

Output:

```
identifier Linevalue not defined
identifier LineNum not defined
```

20 Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print Linevalue;
        Output to "result.txt" LineNum;
};
```

Output:

```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

}
```

22. Input:

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};


Replace myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};
```

Output:

```
line 13: expecting "with", found 'using'
line 16: expecting EOF, found '}'
```

23.Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};


Replace myname in file1 with CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};
```

Output

```
line 13: unexpected token: CaseOn
line 16: expecting EOF, found '}'
```

24. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};


Replace myname in file1 with "abc" CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};
```

Output

```
line 13: unexpected token: abc
line 16: expecting EOF, found '}'
```

25. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
String newname = "ghk";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};


Replace myname in file1 with newname {
        Print LineValue;
};
```

Output:

```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

PrnRec tempPrnRec2 = {false, "", true, true, true, true, true}

ReplaceStr("abc", "test.txt", ghk", true, true, &tempPrnRec2);

}
```

26. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
String newname = "ghk";

Find myname in file1 using CaseOn, SubDirectoryOff {
        Print LineValue;
        Output to "result.txt" LineValue;
};


Replace myname in file1 with newname using CaseOff{
	Print LineValue;
	Output to "result.txt" LineValue;
};
```

Output:

```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

PrnRec tempPrnRec2 = {true, "result.txt", true, true, true, true, true}

ReplaceStr("abc", "test.txt", ghk", false, true, &tempPrnRec2);

}
```

28. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
String newname = "ghk";

Find myname in file1 using CaseOn, SubDirectoryOff {
	Print LineValue;
	Output to "result.txt" LineValue;
};


Replace myname in file1 with newname using CaseOff, SubDirectoryOff{
	Print LineValue;
	Output to "result.txt" LineValue;
};
```

Output
```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

PrnRec tempPrnRec2 = {true, "result.txt", true, true, true, true, true}

ReplaceStr("abc", "test.txt", ghk", false, false, &tempPrnRec2);

}
```

29. Input
```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
String newname = "ghk";

Find myname in file1 using CaseOn, SubDirectoryOff {
	Print LineValue;
	Output to "result.txt" LineValue;
};


Replace myname in file1 with newname using SubDirectoryOff{
	Print LineValue;
	Output to "result.txt" LineValue;
};
```

Output:
```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);
PrnRec tempPrnRec2 = {true, "result.txt", true, true, true, true, true}

ReplaceStr("abc", "test.txt", ghk", true, false, &tempPrnRec2);

}
```

30. Input

```
// This is  for testing the SSL Comments
/* this is C++ Style testing  */
String myname = "abc";
File file1 = "test.txt";
Directory dir1 = "C\temp";
String newname = "ghk";

Find myname in file1 using CaseOn, SubDirectoryOff {
      Print LineValue;
      Output to "result.txt" LineValue;
};


Replace myname in file1 with newname using SubDirectoryOff, CaseOn{
      Print LineValue;
      Output to "result.txt" LineValue;
};
```

Output:

```
#include "SSLLib.h"

void main()
{


PrnRec tempPrnRec1 = {true, "result.txt", true, true, true, true, true}

FindStr("abc", "test.txt", true, false, &tempPrnRec1);

PrnRec tempPrnRec2 = {false, "", false, true, true, false, false};

ReplaceStr("abc", "test.txt", ghk", true, false, &tempPrnRec2);

}
```