Clayton Green (230089109)
kgreen1@unbc.ca
October 9, 2013

# CPSC 425 - Compiler Project Phase II

## Introduction

Included is Phase II of the compiler project partially completed. What's missing for phase II is the construction of the abstract syntax tree (AST). The parser has all productions from the LL1 grammar and is able to receive input from the scanner. Files included are a precompiled Compiler, source files (including header files for a library, TCLAP, used for command line argument parsing), test files and output files for those tests.

## Participation

Clayton Green - All

## Project Status

Scanner - Completed
Parser Basic - Not Complete
+ Complete AST
Parser Full - Not Started
+ Add error recovery (synch)
TODO
+ Fix --err/-e appending to file rather than overwriting
+ Add functionality for --out/-o
+ Strip filename from file path to append to line trace
+ Parser needs more comments
+ Use makefile instead of shell script
+ Fully update Parser Document

## Architecture and Design

The main class parses the command line arguments and creates an Administrator object (AO) with a list of given input source files. Depending on what switches are encountered, the main function causes the AO to call certain methods, usually different phases of compilation.

The Administrator class is used to conduct the different phases of compilation. It also holds the word map, spelling table, and list of source files to be compiled.

The Messenger is a completely static class, allowing it to be accessed from any file or phase of the compiler. The function printMessage() is inserted wherever a message should be reported but is only sent to STDOUT or a ERR file if the --verbose switch is present.

The Token class defines three types of tokens: a standard token (symbols), a word token (reserved words, identifiers and boolean literals) and number tokens (integer literals). Each type of token has its own constructor. A standard token can be identified by its name (PLUS, DIV, GT). A word token has a lexeme associated with it, if it is a reserved word the token name can fully describe it, if it is an identifier its lexeme is stored in a spelling table and the identifier's value is the index of its lexeme in the spelling table. A number token has a value associated with it, which is just an integer. When a word token is created, its lexeme is checked against the reserved word and spelling table list. If it has already been defined its value is retrieved from the table if not, the new value is added to the table for more efficient use later on.

**Scanning Phase:**

```
FOR EACH file IN inputFileList
        create a new Scanner s for file
        if s canScan
                get a token t
                while t is not ENDFILE
                        print trace
                        t = s.getToken()
                ENDWHILE
                print errors
        else ERROR
ENDFOR
```

The scanner's getToken() method is a dfa that reads input from the source file until a token can extracted. Control codes are filtered out of the source file as they come through the dfa. The dfa continues to read in characters until a token is read in or an invalid character is read.

**Parsing Phase:**

```
FOR EACH file IN inputFileList
        create a new Parser p for file
        if p canParse
                p.parse()
        else ERROR
        print errors
ENDFOR
```

When parse() called the parser gets a lookahead token from the scanner using its getToken() method. Starting from the *program* production the parser checks its lookahead token

against the first sets of the production choices and determines if this token is valid or if it's a syntax error. When a non terminal is encountered in a production choice, the parser's match function is called with that non-terminal as the expected value. If the lookahead and the expected value match, then the lookahead is given the next token from the scanner and the production continues. If they do not match then a syntax error has occurred and it is reported to the messenger. The parse function continues until the ENDFILE token is encountered or too many syntax errors have occurred.

The EBNF Grammar used by the parser is the given LL1 grammar. The dangling else ambiguity is handled by associating the else with the closest if statement.

*IF (expression) IF (expression) command ELSE command*
is the same as
*IF (expression) { IF (expression) command ELSE command }*
not
*IF (expression) { IF (expression) command } ELSE command*

**AST Nodes**

***NodeType : SuperClass***
*+ attributes*

**Node**
+TokenName type

**CompoundNode**
+Node* next

**ExpressionNode : Node**

**BinaryNode : ExpressionNode**
+Node* left
+Node* right

**UnaryNode : ExpressionNode**
+Node* right

**AssignNode : BinaryNode**
-- **:=**

**CondNode : BinaryNode**
-- ||, &&, and, or

**RelNode : BinaryNode**

-- =, /=, <=, <, >, =>

**OpNode : BinaryNode**
-- +, -, *, /, mod

**AccessNode : BinaryNode**
-- [ ]

**NotNode : UnaryNode**
-- not

**MinusNode : UnaryNode**
-- -

**RefNode : UnaryNode**
-- ref
+VariableNode* variable

**ReturnNode : UnaryNode**
-- return
+ExpressionNode* expression

**StatementNode : Node**

**IfNode : StatementNode**
+ExpressionNode* expression
+StatementNode* statement

**IfElseNode : IfNode**
+StatementNode* statement

**LoopNode : StatementNode**
+StatementNode* statement

**BranchNode : StatementNode**
+ExpressionNode* expression
+CaseNode* case

**CaseNode : StatementNode**
+int num
+StatementNode* statement

**VariableNode : Node**
+TokenName type -- int, bool

+int id -- index of spelling table entry
+Node* next

**FunctionNode : Node**
+TokenName type -- int, bool, void
+int count -- number of parameters
+VariableNode* param -- pointer to first parameter
+Node* next

# Implementation

Limitations:

Each source file is read into a string so leading and trailing whitespace could be removed. This limits the length of the source file depending on the system.

For multiple input files a switch -I has to be used for each file. Better or more standard functionality would be that a list of files as the last arguments without a switch would be considered the input files.

# Building and Use

Extract:

*$ tar zxf green-parser-basic.tar.gz* -- Compressed tar
*$ tar xf green-parser-basic.tar* -- Standard tar

This extracts the project into a directory named green-parser-basic.

Build:

*$ cd green-parser-basic*
*$ ./build.sh* -- make sure build.sh can be ran as an executable

This will build the compiler and put the executable in the folder
$ */green-parser-basic/executables/Compiler*

Usage:

*$ ./Compiler {-l | -p | -s | -t | -c} {-q | -v} -I <filepath> { -I <filepath> }* [-o]*

*Option Listing:*

| | |
|---|---|
| *-h | --help* | *Display usage info and exits* |
| *-v | --version* | *Display version and exit* |
| -l | --lex | Process up to the Lexer phase |
| -p | --parse | Process up to the Parser phase |
| -s | --sem | Process up to the Semantic Analyser phase |
| -t | --tup | Process up to the Tuple phase |

-c | --compile  Process all phases and compile (default)
-q | --quite  Only display error messages (default)
-v | --verbose  Display all trace messages
-I  | --input  File to be compiled
-e | --err  Error file (defaults to STDOUT)
-o | --out  Output file (defaults to STDOUT)

Examples:

*$ ./Compiler -p -v -I ./../Tests/input/parser/correct.cs13*
*-I ./../Tests/input/parser/incorrect.cs13 -e ./../Tests/results/parser/out.txt*

This will run the files *correct.cs13* and *incorrect.cs13* through the compiler up to the parser phase ( *-p* ) with trace messages ( -v ). Then output the trace messages and error messages to *out.txt*. (Note: this is when running Compiler from the executables directory).

# Code

Parser Class

You can create a Parser object, p, by,

*Parser p( filepath, filename );*

where *filepath* is the path to the source file to be parsed and *filename* is the name of the file for error and trace purposes. Parsing on the source file begins when *p.parse()* is called. This parses the source file, using the productions from the LL1 grammar, until the endfile token is received or when too many syntax errors have occurred where the arser cannot move on. The user is responsible for checking if the parser is good to parse (i.e. the source file was successfully opened) by calling *p.canParse()*.

The only interesting part of the Parser is the trace method. The trace method was created to reduce duplicate trace code. It is used to enter production methods and takes three arguments: a filename, a reference to the calling object and a pointer to the function that object is to call. Because each production method has the same arguments and return value, only one trace method was needed. A trace message ENTERING is printed when the function is called and LEAVING is printed when the pointer function returns. To make it more readable a define macro was used to call the function.

Scanner Class

You can create a Scanner object, s, by,

*Scanner s( filepath, filename );*

where *filepath* is the path to the source file to be scanned and *filename* is the name of the file for error and trace purposes. The main purpose of the scanner is to create tokens from its source file, the tokens are retrieved by the user calling s.getToken(). It is the

user's responsibility for checking if the scanner is good to scan (i.e. the source file was successfully opened) by calling s.isOpen(). The functions *getFilename()* and *getLine()* are used in trace and error reporting outside of the scanner when it is still in use.

Administrator Class

You can create an Administrator object, a, by,

*Administrator a( inputFileList );*

where *inputFileList* is the list of source file paths to be compiled. The parser is used to run the different phases of compilation. Currently *a.lexerPhase()* and *a.parserPhase()* are implemented. Along with compiling the administrator holds the data structures for the compiler, like the word and spelling tables and the AST.

Messenger Class

The messenger class is completely static, so you don't have to create one. To enable the printing of messages the user has to call *Messenger::setMode(Messenger::DEBUG)* and wherever the method *printMessage( message )* is called a message will appear on the standard output. To print messages to file the function *Messenger::setOutput(file)* is called. This changes output stream to point a file rather than the standard output. Throughout compilation encountered errors are added to a list by calling the method *Messenger::addError( filename, lineNumber, message)*. When the phase is completed *Messenger::printErrors()* should be called to report encountered errors. If no errors were added to the list then nothing is printed.

Token Class

You can create a Token object, t, by,

*Token t( TokenName ); -- For a symbol like + or {*
*Token t( TokenName, lexeme ); -- For a word token*
*Token t( TokenName, value ); -- For a number token*

where TokenName is the predefined name of the token (ID, PLUS, OR, etc), *lexeme* is string of the token ( identifier's name, reserved word) and value is an integer value for a number token. Whenever a token of type ID is created it is added to a spelling table if it is not already there. The *getValue()* method has different meaning depending on what type of token is calling it. If it is a ID token then the *value* is the index of its spelling table entry, if it is a NUM token then the *value* is the integer value of the token or if it is a BLIT token then 0 is *false* and 1 is *true*.

# Tests and Observations

I tried to create two different types of tests for the basic parser: a syntactically correct source file (SC) and a syntactically incorrect source file (SI). The SC file consists of all of the productions defined in the CS*13 LL1 grammar. This file when run through the parser phase of the compiler will return 0 errors and its trace output is in the folder Tests/results/parser-basic/correct.txt. The SI file consists of a syntactically incorrect CS*13 program. The problem with the incorrect test is that because there is no error recovery at the moment the parser will exit before reading in all of the source file. Testing of the SI file will make more sense when error recovery

is implemented.