香港中文大學（深圳）

THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

LL(1) parser

Student Name: Cao Zhengjia

Student ID: 117010008

Date: 4/14/2021

Parser for Simplified C language

117010008 Zhengjia Cao

1. Purpose

In this project, I have written a parser for the simplified C language. The parser is a part of the compiler front-end. It's role is to take the input `tokens` from the scanner, and return an abstract syntax tree (AST) using the pre-defined grammar rules as the output. Unlike the YACC or the GNU Bison, this compiler is just a compiler for the given context free grammar (CFG). If the grammar is changed, the parser will fail to do its job. The parser I have implemented for this project is a LL(1) parser using the recursive descend parsing schema. However, in the PDF file, the CFG was specified for a LALR parser. This have been proved to be causing tons of problem as it does not pass the LL(1) grammar requirements. I have made modification to the CFG given in the assignment PDF to pass the LL(1) grammar requirements. The details of the CFG modifications and methods will be covered in the later sections.

Before we start to discuss about the details of the implementation, I would like to introduce again what a parser is doing and how the parser is defferent from yacc, the parser generator. Yacc (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a LALR parser (the part of a compiler that tries to make syntactic sense of the source code) based on a formal grammar, written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement. The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees.

However, a LR parser is hard to implement by hand. Moreover, our grammar rules are fixed. Therefore, it is unnecessary to use LR parser if we can find an efficient method to change the given CFG into a LL(1) recognizable grammar. Moreover, the convertion to LL(1) grammar can be done prior to runtime, which won't sacrifice much system performance. After a trade-off between development efficiency and generalizability, I have chosen to implement a LL(1) parser.

2. Method and Design

In this section, I will first describe some of the methods I have used to make the CFG posted following the LL(1) standard. I will also be discussing the design of the compiler itself in this section.

2.1 Remove Left Recursion

Generally, for a CFG, there will exist two kinds of left recursion: *Immediate Left Recursion* and the *Multistep left recursion*. For example, consider the following description:

$$A \rightarrow A\beta|\epsilon$$

where A is a non-terminal term and β is another production rule. For this case, when we are trying to substitute A, there will be an infinite recursion. This kind of recursion happens within one step of substitution and is therefore recognized as immediate left recursion. To eliminate this infinite recursion, we can use elimination rules to make

$$A \rightarrow A'$$
$$A' \rightarrow \beta|\epsilon$$

With this substituion, we can eliminate the infinite recursion problem when we are using the recursive descend algorithm. However, there is also another type of left recursion. Consider

$$S \rightarrow Aa|b$$
$$A \rightarrow Ac|Sd|\epsilon$$

In this case, by substitute two steps, we will have an implicit left recursion because $S \rightarrow Aa \rightarrow Sda$. To eliminate this kind of recursion, we need to apply the algorithm:

```
1)  arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2)  for ( each $i$ from 1 to $n$ ) {
3)          for ( each $j$ from 1 to $i-1$ ) {
4)                  replace each production of the form $A_i \rightarrow A_j\gamma$ by the
                        productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
                        $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current $A_j$-productions
5)          }
6)          eliminate the immediate left recursion among the $A_i$-productions
7)  }
```

2.2 Remove Left Common Factor

After that, we can have a CFG that is without the worry of left recursion. Instead, we need to focus on the (1) property in the LL(1) grammar. If we want to choose the proper production rule to use with just looking ahead one token, we need to eliminate the common left factors. Here is an example of the common left factors in these two productions:

$$A \rightarrow cf|\epsilon$$
$$B \rightarrow cfgh$$

Then, we have common left factor cf. With the lookahead token being c, there is no way we can identify which production rule we need to use. We need to make the productions:

$$A \rightarrow cfA'$$
$$A' \rightarrow gh|\epsilon$$

The general algorithm that describes this procedure is:

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{array}{l} A \to \alpha A' \mid \gamma \\ A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

2.3 Make First and Follow

After we have finish converting the grammar production rules into the LL(1) form, we can apply the general top-down parsing procedure. In the LL(1) implementation, we should use the recursive descend parsing to solve the problem. In order to know which production to take at each step, we need to calculate the first set and the follow set for each terminal or nonterminal symbol.

In theory, the first set and the follow set are calculated like this:

FIRST(α)

If α is any string of grammar symbols, let FIRST(α*)* be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

1. If X is terminal, then FIRST(X) is {X}.

2. If X → ε is a production, then add ε to FIRST(X).

3. If X is nonterminal and X →Y_1 Y_2 ... Y_k. is a production, then place *a* in FIRST(X) if for some *i, a* is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, Y_1, ... ,$Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k,$ then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow \varepsilon$, then we add FIRST($Y_2$) and so on.

FOLLOW(A)

Define FOLLOW(A), for nonterminal A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form, that is, the set of terminals *a* such that there exists a derivation of the form $S \Rightarrow \alpha A a \beta$ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and *a*, but if so, they derived ε and disappeared. If A can be the rightmost symbol in some sentential form, then \$, representing the input right endmarker, is in FOLLOW(A).

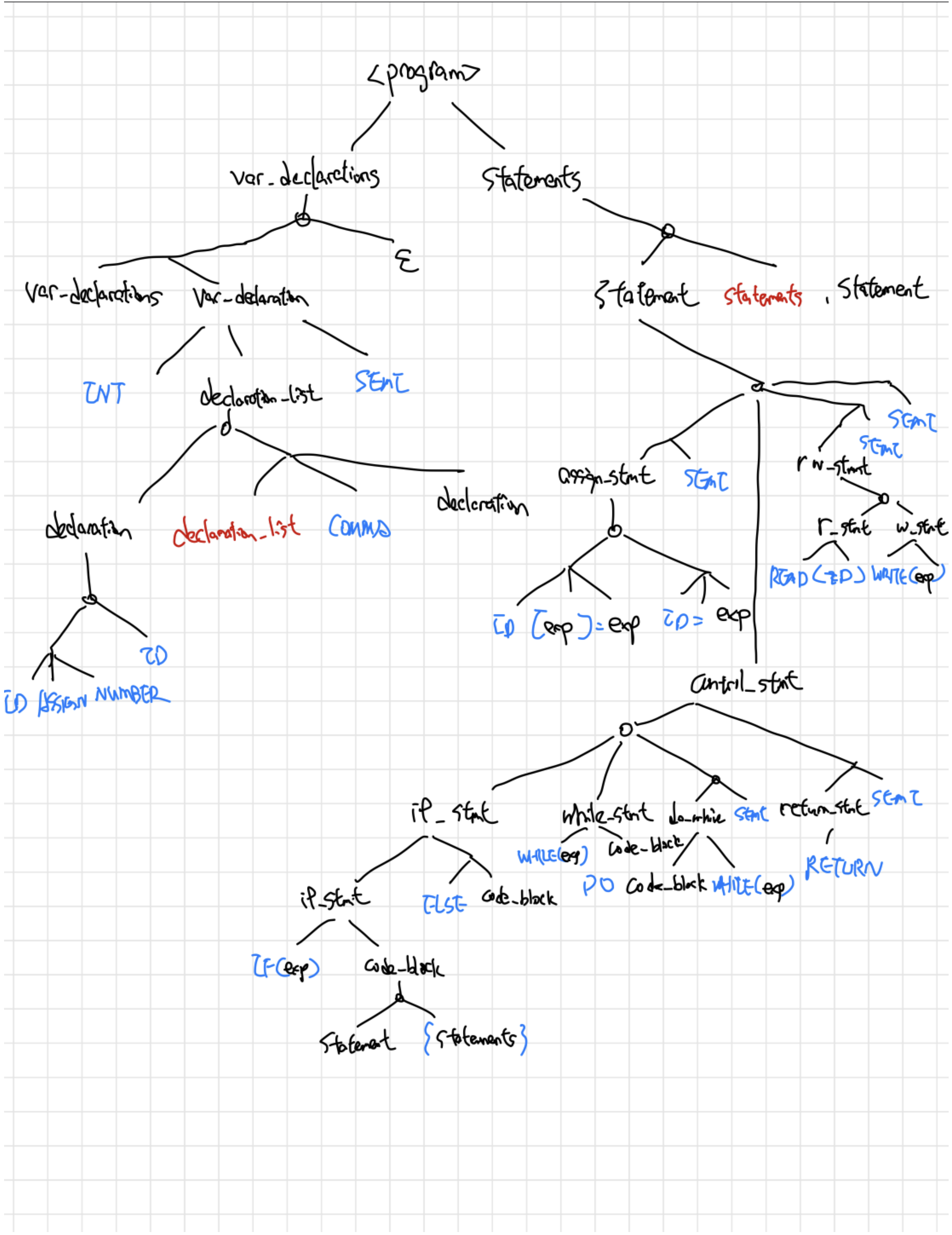
To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.

2. If there is a production $A \Rightarrow \alpha B \beta$, then everything in FIRST(β), except for ε, is placed in FOLLOW(B).

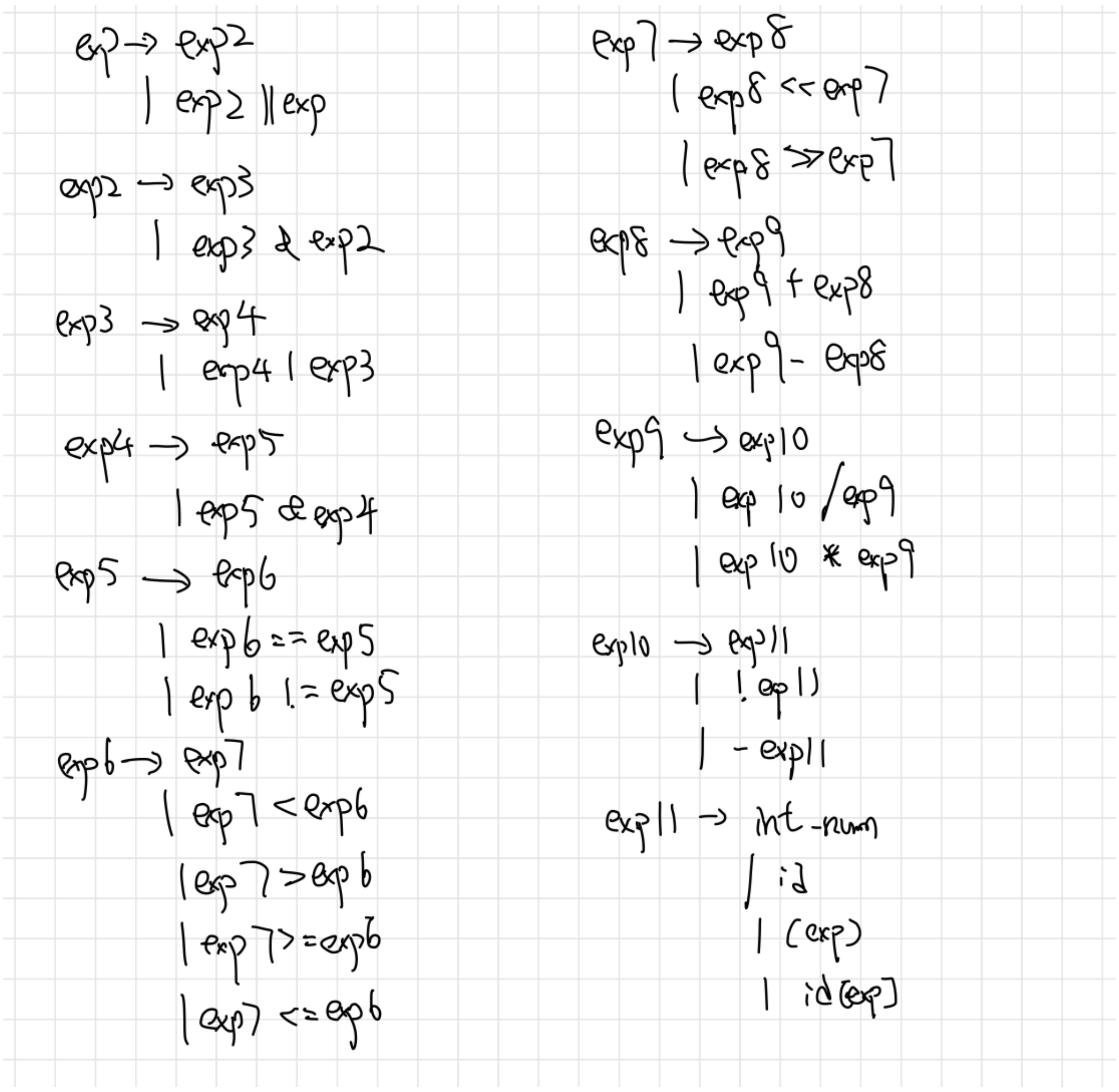
3. If there is a production $A \Rightarrow \alpha B$, or a production $A \Rightarrow \alpha B \beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Based on the first set and the follow set, we can construct a LL prediction table. For my implementation, I have directly constructed a tree denoting the CFG based on the first and follow set:



2.4 Ambiguity in the grammar given

However, if we look closer to the grammar given, we can see that we have ignored the precedence between the different operators. For example, * should have a higher priority than + in calculation. Therefore, we need to apply another transformation on the productions rules associated with `exp`.



```
exp -> exp2
     | exp2 || exp

exp2 -> exp3
      | exp3 & exp2

exp3 -> exp4
      | exp4 | exp3

exp4 -> exp5
      | exp5 & exp4

exp5 -> exp6
      | exp6 == exp5
      | exp6 != exp5

exp6 -> exp7
      | exp7 < exp6
      | exp7 > exp6
      | exp7 >= exp6
      | exp7 <= exp6

exp7 -> exp8
      | exp8 << exp7
      | exp8 >> exp7

exp8 -> exp9
      | exp9 + exp8
      | exp9 - exp8

exp9 -> exp10
      | exp10 / exp9
      | exp10 * exp9

exp10 -> exp11
       | ! exp11
       | - exp11

exp11 -> int-num
       | id
       | (exp)
       | id[exp]
```

Moreover, there is a **dangling else problem** in this grammar. A dangling else refers to this kind of situation:

```
statement = ...
   | selection-statement

selection-statement = ...
   | IF ( expression ) statement
   | IF ( expression ) statement ELSE statement
```

Thus, without further rules, the statement

```
if (a) if (b) s; else s2;
```

could ambiguously be parsed as if it were either:

```
if (a)
{
  if (b)
    s;
  else
    s2;
}
```

or:

```
if (a)
{
  if (b)
    s;
}
else
  s2;
```

In practice in C the first tree is chosen, by associating the `else` with the nearest `if`.

Thus, in the implementation, I have chosen the conventional way in C, just to match the else with the nearest if.

2.5 Program Design

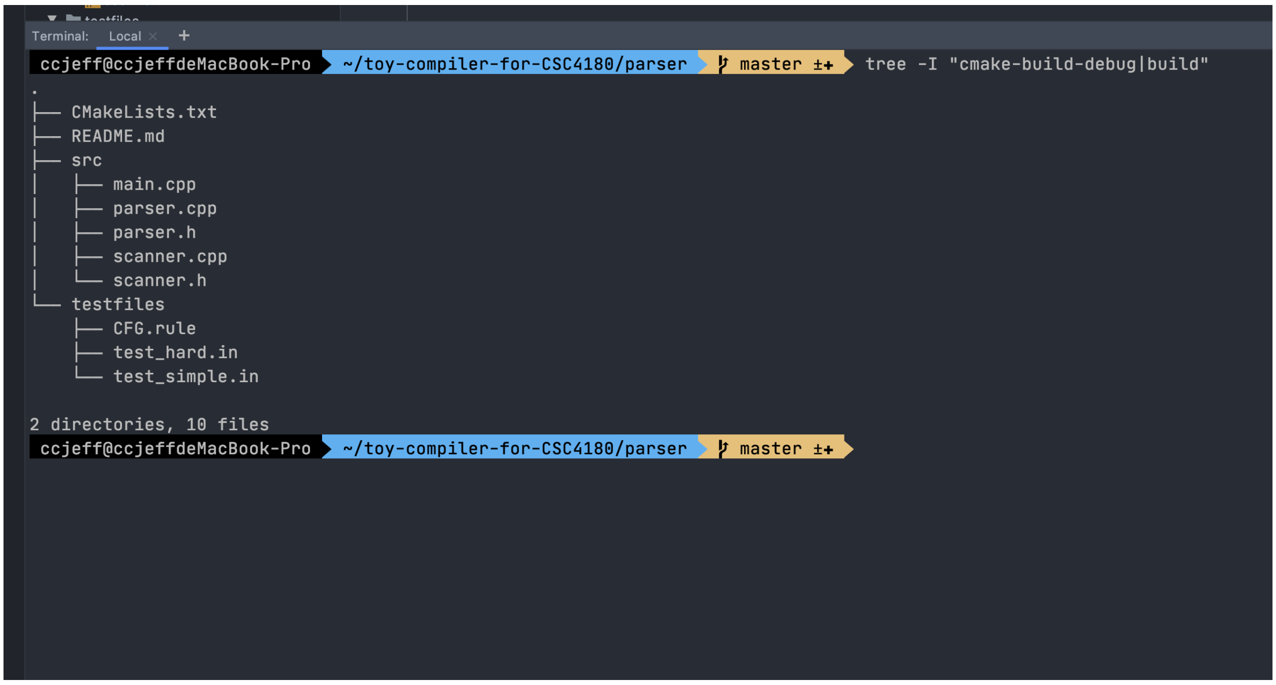
In this program, I have created an abstract syntax node to maintain the AST structure. The production rules have inherited this node and implemented different parsing sub-routines based on the production rules. The header definition is like follows:

```
// the general syntax node for the AST. Inherited by different nodes.
class AbstractSyntaxNode {
public:
    static void validateTokenType(Scanner &scanner, TokenType tokenType) {
        if (!scanner.hasToken()) {
            throw UnexpectedTokenException(token2string(tokenType));
        }
        if (scanner.getTokenType() != tokenType) {
            throw UnexpectedTokenException(token2string(tokenType),
scanner.getTokenValue());
        }
    }

    virtual void fillPrettyStrings(std::vector<string> &rows, unsigned
indentBase, unsigned indentFactor) = 0;
};

class Program;

class VarDeclarations;

class VarDeclaration;

class DeclarationList;

class Declaration;

class CodeBlock;

class Statements;

class Statement;

class ControlStatement;

class ReadWriteStatement;

class AssignStatement;

class IfStatement;

class IfStmt;

class WhileStatement;

class DoWhileStatement;

class ReturnStatement;

class ReadStatement;

class WriteStatement;

class Exp;

class Exp2;

class Exp3;

class Exp4;
class Exp5;

class Exp6;

class Exp7;

class Exp8;

class Exp9;

class Exp10;

class Exp11;
```

The detailed implementation of the classes can be found in the code.

3. Demo & Testings

The tree view of the directory is shown here:



```
ccjeff@ccjeffdeMacBook-Pro  ~/toy-compiler-for-CSC4180/parser  master ±+  tree -I "cmake-build-debug|build"
.
├── CMakeLists.txt
├── README.md
├── src
│   ├── main.cpp
│   ├── parser.cpp
│   ├── parser.h
│   ├── scanner.cpp
│   └── scanner.h
└── testfiles
    ├── CFG.rule
    ├── test_hard.in
    └── test_simple.in

2 directories, 10 files
ccjeff@ccjeffdeMacBook-Pro  ~/toy-compiler-for-CSC4180/parser  master ±+
```

This is tested under the environment:

OS: MacOS Catalina 10.15.5

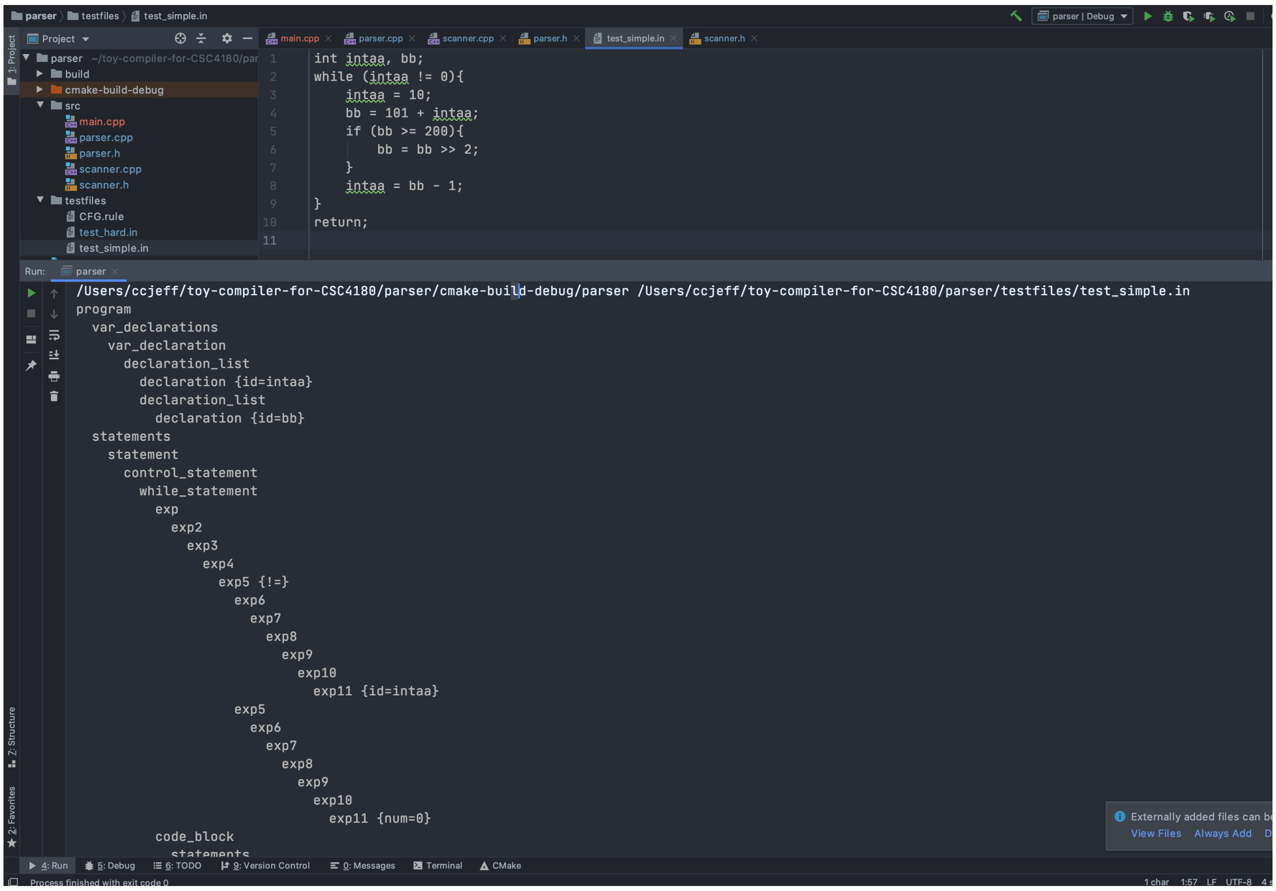
Compiler: Apple clang version 12.0.0 (clang-1200.0.32.29)
Target: x86_64-apple-darwin19.5.0
Thread model: posix

The program can be compiled using the CMake file provided:

```
mkdir build
cd build
cmake ..
```

If you do not have a cmake installed, I have included the build files in the folder. Just make using the makefile.

The sample output is shown here:



```
int intaa, bb;
while (intaa != 0){
    intaa = 10;
    bb = 101 + intaa;
    if (bb >= 200){
        bb = bb >> 2;
    }
    intaa = bb - 1;
}
return;
```

```
/Users/ccjeff/toy-compiler-for-CSC4180/parser/cmake-build-debug/parser /Users/ccjeff/toy-compiler-for-CSC4180/parser/testfiles/test_simple.in
program
  var_declarations
    var_declaration
      declaration_list
        declaration {id=intaa}
        declaration_list
          declaration {id=bb}
  statements
    statement
      control_statement
        while_statement
          exp
            exp2
              exp3
                exp4
                  exp5 {!=}
                    exp6
                      exp7
                        exp8
                          exp9
                            exp10
                              exp11 {id=intaa}
                    exp5
                      exp6
                        exp7
                          exp8
                            exp9
                              exp10
                                exp11 {num=0}
          code_block
            statements
```

```
Process finished with exit code 0
```

4. Analysis

Although in this parser, I have got most of the grammar analysis done, (e.g. symbol precedence, production rules), the parser here is still missing something. Because I have been taking the LL(1) parsing technique, the operators are now all right associative. This was determined by the nature of recursive descend parsing routines. In order to allow the right association rules to be applied, we need to spin the grammar tree. This will be covered by the future work.

5. Reflections

What I have learned through this project is how to write a parser without using a parser generator. Moreover, the most important thing I have learn is the trade-off between LL parser and LR parser. LL parser is certainly easier to implement compared with the LR parser, but it is more restrictive in terms of the CFG. In order to use the LL parser, CFGs need to be pre-processed. However, as the CFGs only needs to be prepocessed and can be done. before runtime, it won't sacrific much performance.