

# **CS4318: Compiler Constructions**

## **Project Phase 2: Parser for mC**

### **100 Points**

**Submission Deadline: 28th March, Friday. 11:59 PM**

#### **Objectives**

Write a parser for mC

#### **Description**

Your task for this project is to write a parser for mC that will parse the token stream generated by your lexical analyzer. The parser will detect syntax errors for programs that do not meet the specifications of the mC grammar. For all syntactically correct programs the parser will construct an abstract syntax tree (AST) representation. The AST will be used later by the semantic analyzer and code generator. Your parser will also need to build a symbol table for keeping track of names within the program.

This assignment comes with a templated parser with detailed description of what each function and variable should do. Please feel free to implement the project whoever you wish. As long as the directory structure, the names of the files do not change, you are welcome to design your own solution. Please refer to Tables 1 and 3 for a detailed rubric.

#### **Syntax Specification**

The grammar for mC is attached to this handout. Your first task is to express this grammar using yacc specification rules. You will want to run your specification through yacc to make sure there are no conflicts in the grammar. If there are conflicts, you will need to eliminate them by rewriting the specifications without changing the meaning of the grammar.

#### **Abstract Syntax Tree (AST)**

The AST is a tree representation of the syntax of the input program. You have some flexibility as to how you structure this tree. A reasonable strategy might be to use an n-ary tree where the internal nodes correspond to non-terminals in the grammar and the leaf nodes represent terminals (e.g., identifier, integer constants etc). Note, this rule does not need to be enforced strictly. In some cases, it may be convenient to have a node in the tree that is neither a terminal nor a non-terminal. In other situations, it may be helpful to add a child node to a terminal. Whatever your implementation strategy, your AST should contain (or have access to) sufficient information so that by traversing the tree it is possible to reproduce source code that is equivalent to the original source. This implies that for some terminals, you will need to store the corresponding semantic value (i.e., name for identifiers, numeric value for integer constants, ASCII value for character constants).

## **Printing an AST**

The type of each node in the tree is printed on its own line starting with the root node. Child nodes appear on the lines following their parent node and are indented with an additional two spaces. Some nodes are printed with additional information. This includes constants, operators, type specifiers, and identifiers. Some nodes are printed with additional information. This includes constants, operators, type specifiers, and identifiers. See the sample output provided for an example.

## **Symbol Table**

At this phase, the symbol table should contain three types of information for each identifier: name, type and scope. Like C, mC has two kinds of scopes for variables local and global. Variables declared outside any function are considered globals, whereas variables (and parameters) declared inside a function foo are local to foo. Each entry in the symbol table should contain the name of the identifier, its scope as the name of the containing function or blank if its scope is global, the data type of the identifier (int, char, or void), and the type of symbol (scalar variable, array variable, or function). The hashkey for a symbol can be calculated by concatenating the identifier's scope and name, using the DJB2 hash function (found here: <http://www.cse.yorku.ca/~oz/hash.html>) on the resulting string, and taking the result modulo the size of the symbol table. Where "type modifier" is blank for a scalar variable symbol, "[]" for an array variable symbol, and "()" for a function symbol.

## **Error Handling**

The parser should detect two types of issues in the source program.

1. **Syntax Error**: If the input program does not conform to the grammar listed above, the parser reports the line number where the program first deviates from the specified grammar.
2. **Use of Undeclared Symbol Warning**: As the parser builds the symbol table, it is capable of recognizing usage of variables and functions before they are declared. As this is more of a semantic error than a syntactic one, this issue is only reported as a warning at this stage. Later phases of compilation will likely treat this as an error.
3. **Errors detected by the Scanner**: The scanner will recognize and provide line number and column number about the following errors in the input source code:
  - a. Comments that span multiple lines
  - b. Unterminated comments
  - c. Strings that span multiple lines
  - d. Unterminated strings
  - e. Escaped characters other than the four listed above
  - f. An integer with leading zeroes
  - g. An identifier that starts with a number

## **Implementation and Testing Instructions**

- You can use any flavor of yacc/bison to implement your parser
- The template\_code directory contains the following files - driver.c, parser.y, scanner.l, strtab.c, strtab.h, tree.c, tree.h, and the makefile.
- The strtab.h and tree.h files contain data structure that you will need to implement the symbol table and the abstract syntax tree.
- The driver.c file contains a function printAST that you will need to implement. This function will print the contents of the entire AST.
- You need to implement functions in the parser.y, strtab.c, and tree.c files. You can use your scanner.1 file from Assignment 1.
- Several test cases are provided for you to test your code. It is only a subset of the cases that will be used to test your code for grading. Please think about edge cases where your program may fail. Testing your code is equally important as writing the code itself.
- Your program will be tested on Zeus, a university provided computing environment.
- You can test your code by typing:

```
./obj/mcc < test/file1.txt
```

Here, "**file1.txt**" will contain an input program. For more examples, take a look at the files provided in the test folder.

## **Writeup**

You need to write a short document named writeup.txt.

In it, you should include the following (numbered) sections.

1. Your name, course id.
2. If a group project, then who contributed in which part.
3. Provide two examples of how you will test the correctness of your code.  
These examples should be different from the ones provided in this document.
4. Extra grading instructions if you choose not to use makefile.

## **Compilation, Testing, and Submission**

- Your program should be compiled using the commands  
make clean; make.
- You are allowed to do otherwise but make sure to add proper instructions in the writeup.txt file.
- The makefile for building your scanner is included.
- We have provided the following skeleton files – driver.c, scanner.l, parser.y, strtab.c, strtab.h, tree.h, tree.c and makefile. During the final submission, please make sure that the latest versions of all of the files are present in the zipped folder.
- If you have made corrections to your previous scanner file, make sure to include the updated scanner.l.

### **Submission Instruction**

You can do this project individually or may form a group of two. Please create a zipped folder named “Project1\_<netId1>\_<netId2>” and submit it on canvas.

Please make sure that the folder name is exactly as we have provided. The grading rubric is given in table 1.

Category	Points
Test Cases (See Table 3)	56
AST	24
Warning Messages for Undeclared Variables	5
Identification of Syntax Errors	5
Compile & Run	5
Documentation	5
Write up	5

**Table 1: Rubric for Project Phase 2**

Submission Deadline: 28th March, Friday. 11:59 PM

Late Amount	Points Deducted
1 Day	10%
2 Days	20%
3 Days	30%
4 Days	40%
5 Days	50%
> 5 Days	No Points

**Table 2: Late Submission Policy**



Category	Points
TestAddExpr	4
TestArithExpr	6
TestAssgnStmt	4
TestCondStmt	4
TestFunDecl	4
TestFuncCall	6
TestGloblVars	4
TestLocalVars	4
TestLoopStmt	4
TestMulExpr	6
TestRelOpExpr	6
TestRetrnStmt	4

**Table 3: Test Cases to Handle**

### **Sample Input & Output:**

A set of sample input and output is provided. Your code should match the outputs when run against the sample inputs. Don't worry about the numbers in the symbol table, they might vary depending on the hashing/mapping strategy you use. Also **don't worry about the indentations. Points won't be deducted as long as your parse tree is printed in the correct order.**

### **Yacc Tutorial:**

1.  Part 01: Tutorial on lex/yacc
2.  Part 02: Tutorial on lex/yacc.

## mC Grammar

---

<i>program</i>	: <i>declList</i>
<i>declList</i>	: <i>decl</i>   <i>declList decl</i>
<i>decl</i>	: <i>varDecl</i>   <i>funDecl</i>
<i>varDecl</i>	: <i>typeSpecifier ID [ INTCONST ] ;</i>   <i>typeSpecifier ID ;</i>
<i>typeSpecifier</i>	: <b>int</b>   <b>char</b>   <b>void</b>
<i>funDecl</i>	: <i>typeSpecifier ID ( formalDeclList ) funBody</i>   <i>typeSpecifier ID ( ) funBody</i>
<i>formalDeclList</i>	: <i>formalDecl</i>   <i>formalDecl , formalDeclList</i>
<i>formalDecl</i>	: <i>typeSpecifier ID</i>   <i>typeSpecifier ID[ ]</i>
<i>funBody</i>	: { <i>localDeclList statementList</i> }
<i>localDeclList</i>	:   <i>varDecl localDeclList</i>
<i>statementList</i>	:   <i>statement statementList</i>
<i>statement</i>	: <i>compoundStmt</i>   <i>assignStmt</i>   <i>condStmt</i>   <i>loopStmt</i>   <i>returnStmt</i>
<i>compoundStmt</i>	: { <i>statementList</i> }
<i>assignStmt</i>	: <i>var = expression ;</i>   <i>expression ;</i>
<i>condStmt</i>	: <b>if</b> ( <i>expression</i> ) <i>statement</i>   <b>if</b> ( <i>expression</i> ) <i>statement</i> <b>else</b> <i>statement</i>
<i>loopStmt</i>	: <b>while</b> ( <i>expression</i> ) <i>statement</i>
<i>returnStmt</i>	: <b>return ;</b>   <b>return</b> <i>expression ;</i>

<i>var</i>	: <b>ID</b>   <b>ID</b> [ <i>addExpr</i> ]
<i>expression</i>	: <i>addExpr</i>   <i>expression</i> <i>relop</i> <i>addExpr</i>
<i>relop</i>	: <=   <   >   >=   ==   !=
<i>addExpr</i>	: <i>term</i>   <i>addExpr</i> <i>addop</i> <i>term</i>
<i>addop</i>	: +   -
<i>term</i>	: <i>factor</i>   <i>term</i> <i>mulop</i> <i>factor</i>
<i>mulop</i>	: *   /
<i>factor</i>	: ( <i>expression</i> )   <i>var</i>   <i>funcCallExpr</i>   <b>INTCONST</b>   <b>CHARCONST</b>   <b>STRCONST</b>
<i>funcCallExpr</i>	: <b>ID</b> ( <i>argList</i> )   <b>ID</b> ( )
<i>argList</i>	: <i>expression</i>   <i>argList</i> , <i>expression</i>