µC: A Simple C Programming Language

Programming Assignment II
Syntactic and Semantic Definitions for µC

Due Date: 23:59, May 16, 2019

Your assignment is to write an LALR(1) parser for the **µ**C language that supports print IO, arithmetic operations, if statements, loop statements and other µC language basic concepts. You will have to write grammar and create symbol tables based on the given **lex** code to create a parser using **yacc**. You are welcome to make any changes of the given **lex** code to meet your expectations. Furthermore, you will do some simple checking of syntax correctness.

1. Yacc Definitions

In the previous assignment, you have built the **lex** code to split the texts of input stream into tokens that should be accepted by **yacc**. For this assignment, you need to build the code to analyze these tokens and check the syntax validity based on the grammar rules you designed.

In particular, you need to do the following three tasks in this assignment.

- Define tokens and data types for **µ**C
- Design grammars and implement actions
- Handle syntax and semantic errors

i. Define Tokens and Types

- **Tokens**

You must define ***tokens*** in both **lex** and **yacc** code. Hence, **lex** recognizes a token when it gets one, and **lex** forwards the occurrence of the token to **yacc**. You should make sure the consistency of the token definitions in **lex** and **yacc** code..

Some tips for token definition (in **yacc**) are listed below:

- Declare tokens using **"%token"**.
- The name of grammar rule, which is not declared as a token, is assumed to be a nonterminal.

- **Types**

Type is one of the predefined data types **integer, float, string and boolean.**

Useful tips for defining a type are listed below:

- Define a type for *yylval* using "**%union { }**" by yourself.
  For example, "**%union** type{ int integer_num; }" means *yylval* is an **integer**.

- Declare a type using **"%type"** and give the type name within the less than (<) and greater than symbols (>)
  For example, "**%type<integer_num> A**" means the nonterminal token **A** has the **integer** type.

```
%union {
   int integer_num;
   float float_num;
}

%type<integer_num> A;
%type<float_num> B;
```

ii. Design Grammar and Implement Actions

- **Grammar**

You should use the CFG (Context Free Grammar) that you learned in the course to design the grammar for arithmetic operations of integers (and floating points). The conversion from the productions of a CFG to the corresponding **yacc** rules is illustrated as below.

Grammar productions for A		Yacc rules
$A \rightarrow B_1\ B_2\ \ldots\ B_m$ $A \rightarrow C_1\ C_2\ \ldots\ C_n$ $A \rightarrow D_1\ D_2\ \ldots\ D_k$	⇨	$A \rightarrow B_1\ B_2\ \ldots\ B_m$ $\mid\ C_1\ C_2\ \ldots\ C_n$ $\mid\ D_1\ D_2\ \ldots\ D_k$;

Note: Arithmetic operations are written in infix notation, and the precedence for the operators is defined as below (**the smaller number has the higher precedence**).

Precedence	Operators	Category
1	++ --	Postfix
2	* / %	Multiplication
3	+ -	Addition
4	== != < <= > >=	Comparison

Hint: You could design your parser grammar based on the ANSI C Yacc grammar rules.

- **Actions**

An action is C statement(s) that should be performed as soon as the parser recognizes the production rule for the input stream. The C code surrounded by '{' and '}' is able to handle input/output, call sub-routines, and update the program states. Occasionally it is useful to put an action in the middle of a rule. The following example code snippet shows that integer constant will be printed out once token I_CONST is recognized.

```
constant
: I_CONST { printf("type %s value %d", "int", $1); }
| F_CONST { printf("type %s value %f", "float", $1); }
;
```

iii. Handle Syntax and Semantic Errors

Your **yacc** program should detect syntax errors during parsing the given **μ**C code. When errors are detected, your parser **should stop when syntactic error detected but continue when meet a semantic error**. Then, display helpful messages upon the termination of the parsing procedure. The messages should include the ***type*** of the ***semantic error*** and the ***line number*** of the code that causes the error. The examples are given in **Assignment files.**

Syntactic error means the statement does not match the grammar.

The types of **semantic error** defined as below:

- Operate on undeclared variables
- Re-define variables

iv. More on the μC Language Definitions

- ### Print Functions

Your **yacc** should implement **print()** function. The function only supports token defined in assignment #1 (i.e., **string constant and ID token**). The syntax of print function is define as below.

print(string or ID);

- ### If...else if...else Statements

Your grammar should consider C-style nested if statements. The syntax of an **if...else if...else** statement is define as below.

```
if( expression 1 ) {
/* Executes when the expression 1 is true */
   if( expression 1.1 ){
      /* Nested if statement */
      /* Executes when the expression 1.1 is true */
   }

} else if( expression 2 ) {
      /* Executes when the expression 2 is true */
} else if( expression 3 ) {
      /* Executes when the expression 3 is true */
} else {
      /* executes when the none of the above condition is true */
}
```

- **While Loop**

Your grammar should consider while loop statements. The syntax of while loops is define as below. Note your grammar should support nested loops.

```
while ( expression ) {
   statement(s);
}
```

- **Functions**

A function is a group of statements that together achieve a certain task. Every **µ**C program has at least one function, which is the main function, and additional functions can be added.

A function declaration has the following form.

type identifier (<zero or more formal arguments>) ;

A function definition has the following form:

```
type identifier (<zero or more formal arguments>){
   /*body of the function,
    *if type of function identifier is not void, return value to the caller
    */
   statement(s);
}
```

- **Scoping Rules**

A scope in any programming is a region of the program, where a defined variable can have its existence and beyond that variable it cannot be accessed. A scope block is a set of statements enclosed within left and right braces (i.e., **{** and **}**). There are three types of variables can be declared in **µ**C language: *local*, *global* and *formal parameters*. The scope rules are defined as following.

- Scope rules are similar to C.
- Names must be unique within a given scope. Within the inner scope, the identifier designates the entity declared in the inner scope, the entity declared in the outer scope is hidden within the inner scope.
- A local variable defined in an inner scope and no longer exist after the block exits.

2. Symbol Table

You are required to implement symbol tables to store all identifiers according to the scope of the code. Symbol tables should be designed for efficient insertion and retrieval operations. Hence, they are usually organized as hash tables. In order to create and manage the tables, **at least the following functions should be provided (you are allowed to add other functions)**:

- **create_symbol ():** Create symbol tables. Declare a data structure and assign the memory space to store variable.
- **insert_symbol ():** Insert symbol into a new entry of the symbol tables.
- **lookup_symbol ():** Check the symbol if it exists.
- **dump_symbol ():** Dump all entries of the symbol tables, no need to dump if the symbol table is empty.

The structure of the example symbol table is listed as below:

Index	Name	Entry Type	Data Type	Scope Level	Formal Parameters
0	height	function	void	0	int, float
1	width	variable	int	1	
2	length	parameter	float	1	

Each entry of the symbol table consists of several data fields, including name, kind, scope level, type, value, and additional attributes of the symbol. Precise definitions of each symbol table entry are as follows.

Name	The name of the symbol.
Entry Type	The name type of the symbol. There are five kinds of symbols: function, parameter and variable.
Scope Level	The scope level of the symbol. 0 represents global scope, and local scope starts from 1, 2, 3, etc.
Data Type	The data type type of the symbol. Each symbol is of types *int*, *float*, *bool*, *string* and *void*. Note that this field can be used for the return type of a function.
Formal Parameters	List of the types of the formal parameters of a function.

Hint: You may add additional data fields in the table to facilitate syntax error checking / handling, but the output of your symbol tables should conform our formats listed below.

3. What Should Your Parser Do?

- **Assignment Requirements (100pt)**
  - Implement the essential functionalities of the symbol tables, as shown in **Section 2**. Your symbol table should at least support these functions: create, insert, lookup, and dump. (25pt)
  - Support the variants of the assignment operators (i.e., =, +=, -=, *=, /=, %=), as shown in **Section 1.ii**. (10pt)
  - Handle arithmetic operations, where brackets and precedence should be taken into account, as shown in **Section 1.ii**. (10pt)
  - Design grammar for print functions, as shown in **Section 1.iii**. (10pt)
  - Design the grammar for the C-style if statements, as shown in **Section 1.iv** (15pt)
  - Design the grammar for the C- style while loop statements, as shown in **Section 1.v**. (15pt)
  - Detect syntax or semantic errors and display the error messages, as shown in **Section 1.viii**. (15pt)

Our grading system uses the Ubuntu environment. We will revise your uploaded code to adopt to our environment. In order to facilitate the automated code revision process, we need your help to arrange your code in the format specified in **Section 5 Submission**.

- **Example input code and the expected output for your parser (without error)**

Example input for your parser:

```
/*
 * 2019 Spring Compiler Course Assignment 2
 */

float c = 1.5;

bool loop(int n, int m) {
    while (n > m) {
        n--;
    }
    return true;
}

int main() {
    // Declaration
    int x;
    int i;
    int a = 5;
    string y = "She is a girl";

    print(y); // print

    // if condition
    if (a > 10) {
        x += a;
        print(x);
    } else {
        x = a % 10 + 10 * 7; /* Arithmetic */
        print(x);
    }
    loop(x, i);
    print("Hello World");

    return 0;
}
```

Example output for your parser:

```
1: /*
2:  * 2019 Spring Compiler Course Assignment 2
3:  */
4:
5: float c = 1.5;
6:
7: bool loop(int n, int m) {
8:     while (n > m) {
9:         n--;
10:     }
11:     return true;
12: }

Index     Name      Kind        Type      Scope     Attribute

0         n         parameter   int       1
1         m         parameter   int       1

13:
14: int main() {
15:     // Declaration
16:     int x;
17:     int i;
18:     int a = 5;
19:     string y = "She is a girl";
20:
21:     print(y); // print
22:
23:     // if condition
24:     if (a > 10) {
25:         x += a;
26:         print(x);
27:     } else {
28:         x = a % 10 + 10 * 7; /* Arithmetic */
29:         print(x);
30:     }
31:     loop(x, i);
32:     print("Hello World");
33:
34:     return 0;
35: }

Index     Name      Kind        Type      Scope     Attribute

0         x         variable    int       1
1         i         variable    int       1
2         a         variable    int       1
3         y         variable    string    1


Index     Name      Kind        Type      Scope     Attribute

0         c         variable    float     0
1         loop      function    bool      0         int, int
2         main      function    int       0


Total lines: 35
```

- **Output format definition:**

  i. You must print the code line in the following format:

  ```
  printf("%d: %s\n", yylineno, code_line);
  ```

  ii. You must dump the symbol tables after every scope with newline in the top and bottom of the tables, the symbol table output format is shown as below:

  ```
  printf("\n%-10s%-10s%-12s%-10s%-10s%-10s\n\n",
         "Index",  "Name", "Kind", "Type", "Scope", "Attribute");
  printf("%-10d%-10s%-12s%-10s%-10d",
         index, name, kind, type, scope_level, attribute);
  printf("\n");
  ```

  iii. The syntactic error output format:

  ```
  printf("\n|---------------------------------------------|\n");
  printf("| Error found in line %d: %s\n", yylineno, code_line);
  printf("| %s", error_message);
  printf("\n|---------------------------------------------|\n\n");
  ```

  Error message includes:

  - syntax error (This message is generated by yacc in default)

  iv. The semantic error output format:

  ```
  printf("\n|---------------------------------------------|\n");
  printf("| Error found in line %d: %s\n", yylineno, code_line);
  printf("| %s", error_message);
  printf("\n|---------------------------------------------|\n\n");
  ```

  Error message includes:

  - Undeclared variable <variable_name>
  - Undeclared function <function name>
  - Redeclared variable <variable_name>
  - Redeclared function <function_name>

4. Yacc Template

You can download the template from Moodle.

```
/* Definition section */
%{
extern int yylineno;
extern int yylex();
/* symbol table function */
int lookup_symbol();
void create_symbol();
void insert_symbol();
void dump_symbol();
%}

/* Using union to define nonterminal and token type */
%union {
    int i_val;
    double f_val;
    char* string;
}
/* Token without return */
%token PRINT PRINTLN
%token IF ELSE FOR
%token VAR

/* Token with return, which need to sepcify type */
%token <i_val> I_CONST
%token <f_val> F_CONST
%token <string> STRING

/* Nonterminal with return, which need to sepcify type */
%type <f_val> stat

/* Yacc will start at this nonterminal */
%start program

/* Grammar section */
%%

program
    : program stat
    |
;
stat
    : declaration
    | compound_stat
    | expression_stat
    | print_func
;
```

5. Submission

- Upload your homework to Moodle before the deadline.
  - **Deadline: 23:59, May 16, 2019 (THU)**
- Compress your files with either the .zip or .rar file. **Other file formats are not acceptable**.
  - Your uploaded assignment must be organized as follow, **otherwise we will ignore it. And you will have to live demo for your homework to get the scores.**

```
Compiler_StudentID_HW2.zip
↳ Compiler_StudentID_HW2/
  ↳ Makefile
  ↳ compiler_hw2.l
  ↳ compiler_hw2.y
  ↳ input/
  ↳ output/
```