µC: A Simple C Programming Language

Programming Assignment II

Syntactic and Semantic Definitions for µC

Due Date: 23:59, 5/26, 2017

Your assignment is to write an LALR(1) parser for the **µ**C language that supports arithmetic operations. You will have to write the grammar based on the given **lex** code and 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 semantic correctness.

1. Yacc Definitions

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

Specifically, you must do the following three tasks in this assignment.

- Define tokens and types
- Design grammar and implement actions
- Handle syntax errors

i. Define Tokens and Types

- **Token**

  You must define ***token*** 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. You are welcome to add/modify the token definitions in the given **lex** 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.

- **Type**

  Type is one of the predefined data types **integer** and **double**.

  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/greater than symbols, < >; for example, "**%type<integer_num> A"** means the nonterminal token **A** has the **integer** type**.**

ii. Design Grammar and Implement Actions

- **Grammar**

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

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

The example grammar rules that would be used in this assignment are listed below.

```
Line    →  Line  Stmt             /* Read in sequence */
Stmt    →  Decl  SEM              /* Declaration (e.g. int a = 6; ) */
Stmt    →  Print  SEM             /* Print */
Stmt    →  Assign  SEM            /* Assignment (e.g. a = 5; ) */
Stmt    →  Arith  SEM             /* Arithmetic */
Decl    →  Type  ID
Decl    →  Type  ID  ASSIGN  Arith
Type    →  INT
Type    →  DOUBLE
Assign  →  ID  ASSIGN  Arith
Arith   →  Term
Arith   →  Arith  ADD  Term       /*print operator when you meet */
Arith   →  Arith  SUB  Term
Term    →  Factor
Term    →  Term  MUL  Factor
Term    →  Term  DIV  Factor
Factor  →  Group
Factor  →  NUMBER
Factor  →  ID
Print   →  PRINT  Group
Print   →  PRINT  LB  STRING  RB
Group   →  LB  Arith  RB
```

Note: Arithmetic operations are written in infix notation, where the operator precedence is defined as: **'('** = **')'** > **'*'** = **'/'** > **'+'** = **'-'** > **"print"**

- **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 code snippet shows that "after B1" will be printed out once B1 is recognized: "***A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3*** ".

iii. Handle Syntax Errors

Your **yacc** program should detect errors during parsing the given **μ**C code. For example, it could look for the errors: misspelled variable and function names, improperly matched parentheses and curly braces, divide by zero (i.e., B = A / 0), and undeclared variables. When errors are detected, your parser should display helpful messages upon the termination of the parsing procedure. The messages should include the ***type*** of the syntax error and the ***line number*** of the code that causes the error.

In this assignment, you should at least handle the following three cases:

- Operate on undeclared variables
- Re-define variables
- Handle *divided by zero* error

2. Symbol Table

You may enhance the symbol table, which you built in the previous assignment, to perform the following tasks:

i. Create a symbol table.

ii. Insert entries for variables declarations.

iii. Look up entries in the symbol table.

iv. Dump all contents in the symbol table and its value.

v. Assign the value to the entry of symbol table entry, e.g., A = 6.

The structure of the example symbol table is listed below.

Index	ID	Type	Data
1	height	int	45
2	width	int	80

Hint: You may add some data fields in the table to facilitate syntax error handling.

3. What Should Your Parser Do?

Your parser is expected to offer the basic features. To get bonus points, your scanner should be able to provide the advanced features.

- **Basic features (100pt)**
  - Parse the input code, which contains variable declarations and arithmetic operations. (60pt)
  - Handle arithmetic operations for integers. The parser should consider brackets, print function, and precedence. (10pt)
  - Detect syntax error and display the error message. The parser should display at least the error type and the line number. (15pt)
  - Implement the essential functionalities defined in Section 2 Symbol Table, for example, create, insert, lookup, and dump. (15pt)

  Note: You should check the below examples to format the output messages for the above symbol table functions.

- **Advanced features (35pt)**
  - Handle arithmetic operations for integers and doubles. The parser should consider brackets, print function, precedence, and data type. (20pt)
  - Design the grammar for the C-style while loop. (15pt)

Example input code and the expected output from your scanner:

Input #1:

```
int a;
int b = 5;
a = b * 20;
b = a / (5 + 5);
print(b);
```

Output #1:

```
Create symbol table
Insert symbol: a
Insert symbol: b
Mul
ASSIGN
Add
Div
ASSIGN
Print : 10

Total lines: 5

The symbol table:

ID    Type    Data
a     int     100
b     int     10
```

Input #2:

```
int a = 20;
int b = 30;
int a;
c = b + 20;
print(b / 0);
```

Output #2:

```
Create symbol table
Insert symbol: a
Insert symbol: b
<ERROR> re-declaration for variable a (line 3)
Add
ASSIGN
<ERROR> can’t find variable c (line 4)
<ERROR> The divisor can’t be 0 (line 5)

Total lines: 5

The symbol table:

ID    Type    Data
a     int     20
b     int     30
```

4. Yacc Templete

```
%{
#define YYSTYPE double
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
extern int yylineno;
extern int yylex();
void yyerror(char *);
%}

%token NUMBER

%%

lines
        :
        | lines expression '\n'     { printf(" = %lf\n", $2); }
        ;
expression
        : term                      { $$ = $1; }
        | expression '+' term       { printf("Add   \n"); $$ = $1 + $3; }
        | expression '-' term       { printf("Sub   \n"); $$ = $1 - $3; }
        ;
term
        : factor                    { $$ = $1; }
        | term '*' factor           { printf("Mul   \n"); $$ = $1 * $3; }
        | term '/' factor           { printf("Div   \n"); $$ = $1 / $3; }
        ;
factor
        : NUMBER                    { $$ = $1; }
        | group                     { $$ = $1; }
        ;
group
        : '(' expression ')'        { $$ = $2; }
        ;

%%

int main(int argc, char** argv)
{
        yyparse();
        return 0;
}

void yyerror(char *s)
{
        printf("%s on %d line \n", s, yylineno);
}
```