µGo: A Simple Go Programming Language

Programming Assignment II

Syntactic and Semantic Definitions for µGo

Due Date: 23:59, 5/31, 2018

Your assignment is to build an LALR(1) parser for the **µGo** language that supports print IO, arithmetic operations and some programming language basic concepts. 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 will do the following three tasks in this assignment.

i. Define tokens and types
ii. Design grammar and implement actions
iii. Handle 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. 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.

- **Types**

  Type refers to one of the predefined data types **integer** and **float32**.

  Useful tips for defining a type are listed below:

  - Define a type for *yylval* using "%**union**" by yourself.

    For example, "%**union** { **int** i_val; }" means *yylval* is able to be accessed via the **int** type.
  - Define a type for ***token*** using "%**type**" and give the type name within the less/greater than symbols, '<' / '>'; for example, "%**type<i_val> I_CONST**" means the token **I_CONST** has the **int** type.

```
%union {
    int i_val;
    double f_val;
    char* string;
}

%type<i_val> I_CONST
%type<f_val> F_CONST
```

ii. Design Grammar and Implement Actions

- **Grammar**

The concept of CFG (Context Free Grammar) that you learned in the courses should be used to design the grammar for print IO, arithmetic operations and basic **µGo** concept. The conversion from the productions of a CFG to the corresponding **yacc** rules is illustrated as below.

Grammar productions for A

$A \rightarrow B_1\ B_2 \ldots B_m$

$A \rightarrow C_1\ C_2 \ldots C_n$

$A \rightarrow D_1\ D_2 \ldots D_k$

⇨

Yacc rules

```
A
   : B1 B2 ... Bm
   | C1 C2 ... Cn
   | D1 D2 ... Dk
;
```

Note: Your grammar should consider the precedence of operators, the precedence are defined in the µGo specification.

Hint: The link is ANSI C grammar rules, you could design your parser grammar base on it.

- **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 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", "float32", $1); }
;
```

iii. Handle Semantic Errors

Your **yacc** program should detect semantic errors during parsing the given **µGo** code. For example, it could look for the following errors: divide by zero (i.e., A = B / 0), and undeclared/re-define variables. When errors occur, your parser should detect and display error 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.

To be precise, in this assignment, you should at least handle the following three cases:

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

2. Symbol Table

You may enhance the symbol table, which you built by **Lex** in the previous assignment. In this assignment, you should build symbol table by **Yacc** 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.

Hint: You may add some data fields in the table to facilitate semantic error handling or scoping check.

The structure of the example symbol table is listed below.

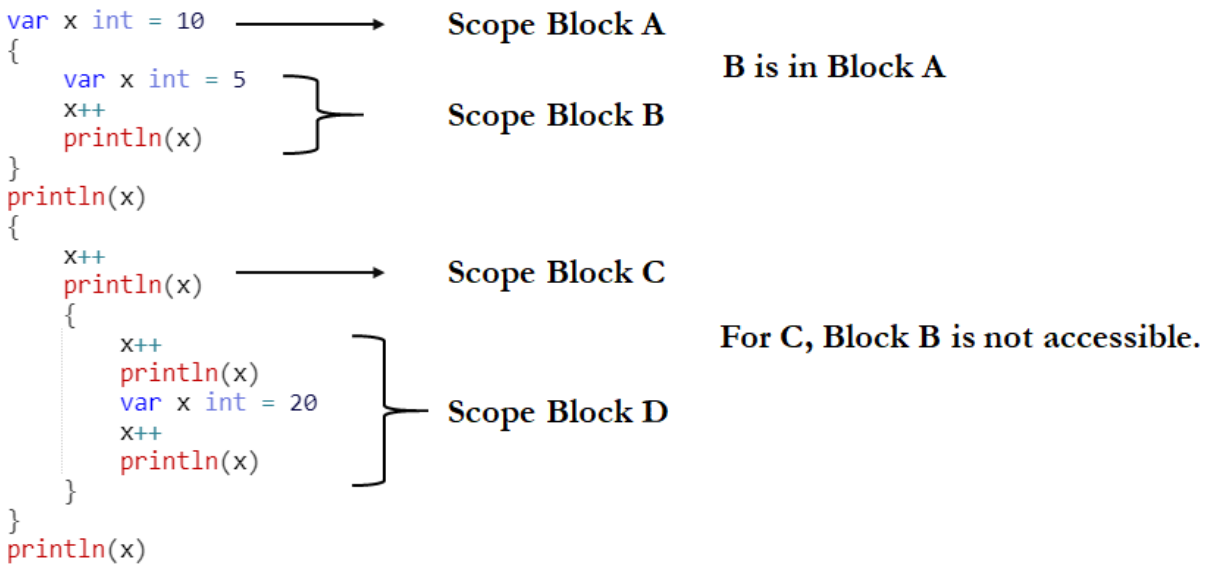
Index	ID	Type	Data
1	height	int	45
2	width	int	80

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)**
  - Design the grammar to match the syntax rules for variable declarations. (15pt)
  - Implement the essential functionalities of the symbol table (defined in Section 2 Symbol Table). Your symbol table should at least support these functions: create_symbol, insert_symbol, lookup_symbol, and dump_symbol. (20pt)
    Note: 1. Your parser are expected to print out the ACTION which matches your grammar. You should reference the examples below to format the output messages for the above symbol table functions. 2. The TAs check the value of each table entry to see if the parser is implemented correctly.
  - Support the variants of the assignment operators.
    (i.e., =, +=, -=, *=, /=, %=) (15%)
  - Handle arithmetic operations, where brackets and precedence should be considered. (20pt)
    Note: The modulo operation (%) does not involve any floating point variables. Hence, your parser should perform the type checking.
  - Design the grammar for accepting the **print** and **println** invocations and display the contents of the arguments for the function calls. (15%)
  - Detect semantic error(s) and display the error message(s). The parser should display at least the error type and the line number. (15pt)
    Notice: Once the semantic error is detected, the correctness of the variable content is not important, e.g., you can do nothing or still assign the value.

- **Advanced features (30pt)**
  - Design the grammar for the case: "*if … else if … else …*", the grammar must allow zero or more occurrences of the "*else if*". (15pt)
  - Implement the scoping check function in your parser. To get the full credits for this feature, your parser is expected to correctly handle the scope of the variables defined by the μGo language. (15pt)



If you decide to challenge the advanced features in Assignment 2, please attach the README for explaining WHAT and HOW advanced function(s) you have implemented.

Example input code and the expected output from your scanner:

Input #1:

```
var a int
var b int = 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:

```
var a int = 20
var b int = 30
var a int
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

```
/*     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
;
```

```
declaration
    : VAR ID type '=' initializer NEWLINE
    | VAR ID type NEWLINE
;

type
    : INT { $$ = $1; }
    | FLOAT { $$ = $1; }
    | VOID { $$ = $1; }
;

%%
/* C code section */
int main(int argc, char** argv)
{
    yylineno = 0;
    yyparse();
    return 0;
}

void create_symbol() {}
void insert_symbol() {}
int lookup_symbol() {}
void dump_symbol() {}
```

5. Submission

- Hand in your homework by using moodle.
- Only allow **.zip** and **.rar** types of compression.
- The directory organization should be

```
Compiler_StudentID_HW2.zip
  ↳ Compiler_StudentID_HW2/
      ↳ input/
        Makefile
        compiler_hw2.l
        compiler_hw2.y
        README ( if required )
        xxx.h ( header file if required )
```

!!! Incorrect format will take your grade 10% off. !!!