μC: A Simple C Programming Language

Compiler 2021 Programming Assignment II
Syntactic and Semantic Definitions for μC
Due Date: 2021/5/27 23:59

Your assignment is to build the **parser** for the μC language that supports print IO, arithmetic operations and some basic constructs for C with **yacc**. You will have to define the token classes and the grammar for C using the given Lex and Yacc codes, respectively, to produce the parser. You are welcome to make any changes of the given codes to meet your expectations. In addition to the C syntax, your produced parser will do simple checking for semantic correctness of the testing cases.

1. Yacc Definitions

In the previous assignment, you have built the Lex code to split the input text stream into tokens that will 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.

1. Define tokens and types (Section 1.1)
2. Design μC grammar and implement the related actions (Section 1.2)
3. Handle semantic errors (Section 1.3)

1.1 Define Tokens and Types

1.1.1 Tokens

The tokens need to be defined in both Lex and Yacc code. 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 both 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.

1.1.2 Types

Type refers to one of the μC 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 { int i_val; }` means `yylval` is able to be accessed via the `int` type using the `i_val` variable.
- Define a type for token using `%type` and give the type name within the less/greater than symbols, `<` / `>`; for example, `%type <i_val> INT_LIT` means the token `INT_LIT` has the `int` type.

```
%union {
    int i_val;
    float f_val;
    char* s_val;
    /* ... */
}
%type <i_val> INT_LIT
%type <f_val> FLOAT_LIT
%type <s_val> STRING_LIT
```

1.2 Design Grammar and Implement Actions

1.2.1 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 constructs. The conversion from the productions of a CFG to the corresponding Yacc rules is illustrated as below.

- Grammar productions for A:
  $A \rightarrow B_1 B_2 \dots B_m$

  $A \rightarrow C_1 C_2 \dots C_n$

  $A \rightarrow D_1 D_2 \dots D_k$

- Yacc rules:
  A

  $: B_1 B_2 \dots B_m$

  $| C_1 C_2 \dots C_n$

  $| D_1 D_2 \dots D_k$

  ;

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

1.2.2 Actions

An action is C statement(s) that should be performed as soon as the parser recognizes the production rule from 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 literal will be printed out after token `INT_LIT` is recognized.

```
literal
    : INT_LIT   { printf("type %s value %d", "INT_LIT", $<i_val>1); }
    | FLOAT_LIT { printf("type %s value %f", "FLOAT_LIT", $<f_val>1); }
;
```

1.3 Handle Semantic Errors

Your Yacc code needs to detect semantic errors during parsing the given µC code. When errors occur, your parser should detect and display error messages upon the termination of the parsing procedure. The messages will 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 four cases:

1. Variable errors:
   - Operate on *any* undeclared variable
   - Re-define *any* existed variable
2. Type errors:
   - Handle modulo operation (`%`) involving any floating point number or variable
   - Handle simple type checking for the mismatching (e.g., `3 + 3.14`) and the condition of "if" and "for" statements that the values must be the *boolean* type

Hint: Your `%union` may need to use `struct` with some fields to record the types of the value.

2. Symbol Table

2.1 Functions

Symbol table needs to be built in the Yacc program so as to perform the following tasks.

1. Create a symbol table when entering a new scope. `create_symbol`
2. Insert an entry for a variable declaration. `insert_symbol`
3. Look up an entry in the symbol table. `lookup_symbol`
4. Dump all contents in the symbol table of current scope and its entries when exiting a scope. `dump_symbol`

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

Hint: You may need to link and organize multiple tables as the operation of a stack.

2.2 Scope Level

The global scope level is **zero** and is increased by one when entering a new block. When the program leaving a block, you need to dump the symbol table of current level then decrease the level by one. You can find the example at section 2.4 below.

2.3 Table Fields

The structure of the example symbol table is listed below:

- Index: the variable index in attached symbol table, and should be unique in that symbol table
- Name: the name of the variable
- Type: the type of the variable
- Address: should be unique in the whole program
- Lineno: the line number where define the variable
- Element type: if Type is "array" record its element type in this field, otherwise just fill in "-"

Index	Name	Type	Address	Lineno	Element type
0	x	int	0	1	-
1	y	float	1	2	-
2	z	array	2	3	int

2.4 Symbol Table Example

- Example C code:

```
int height = 99;
{
    float width = 3.14;
}
float length;
{
    string length = "hello world";
    {
        int length[3];
    }
    int width = 66;
}
```

- Example output of the symbol table:

```
INT_LIT 99
> Insert {height} into symbol table (scope level: 0)
FLOAT_LIT 3.140000
> Insert {width} into symbol table (scope level: 1)
> Dump symbol table (scope level: 1)
Index     Name      Type      Address   Lineno    Element type
0         width     float     1         3         -
> Insert {length} into symbol table (scope level: 0)
```

```
STRING_LIT hello world
> Insert {length} into symbol table (scope level: 1)
INT_LIT 3
> Insert {length} into symbol table (scope level: 2)
> Dump symbol table (scope level: 2)
Index     Name      Type      Address   Lineno    Element type
0         length    array     4         9         int
INT_LIT 66
> Insert {width} into symbol table (scope level: 1)
> Dump symbol table (scope level: 1)
Index     Name      Type      Address   Lineno    Element type
0         length    string    3         7         -
1         width     int       5         11        -
> Dump symbol table (scope level: 0)
Index     Name      Type      Address   Lineno    Element type
0         height    int       0         1         -
1         length    float     2         5         -
Total lines: 12
```

3. What Should Your Parser Do?

- You will get **105pt** if your scanner successfully generates the answers for all eleven programs. Otherwise, the mapping
  between grade and correct count is listed below:
  {"0":"0", "1":"30", "2":"50", "3":"60", "4":"70", "5":"75", "6":"80", "7":"85", "8":"90", "9":"95", "10":"100", "11":"105"}
- Functionalities mapped with the test cases:
  1. Handle arithmetic operations, where brackets and precedence should be considered. (`in01`, `in02`)
  2. 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 μC language. (`in03`)
  3. Handle the declarations and operations for the array type. (`in04`)
  4. Support the variants of the assignment operators. (i.e., `=`, `+=`, `-=`, `*=`, `/=`, `%=`) (`in05`)
  5. Handle the type conversion between integer and floating-point. (`in06`)
  6. Support if statements. (`in07`)
  7. Support for statements. (`in08`)
  8. Detect semantic error(s) and display the error message(s). The parser should display at least the error type and the line number. (`in09`, `in10`)
  9. Complex program includes all features mentioned above. (`in11`)

3.1 Example

Example input code and the expected output from your parser.

- Input #1:

```
int sum = 0;
int i = 0;
while (i <= 10) {
    sum += 1;
    i++;
}
print(sum); // 55
```

- Output #1:

```
INT_LIT 0
> Insert {sum} into symbol table (scope level: 0)
INT_LIT 0
> Insert {i} into symbol table (scope level: 0)
IDENT (name=i, address=1)
INT_LIT 10
LEQ
IDENT (name=sum, address=0)
INT_LIT 1
```

```
ADD_ASSIGN
IDENT (name=i, address=1)
INC
> Dump symbol table (scope level: 1)
Index     Name      Type      Address   Lineno    Element type
IDENT (name=sum, address=0)
PRINT int
> Dump symbol table (scope level: 0)
Index     Name      Type      Address   Lineno    Element type
0         sum       int       0         1         -
1         i         int       1         2         -
Total lines: 7
```

- Input #2 (with error):

```
float y;
x += y
y %= 3
```

- Output #2:

```
> Insert {y} into symbol table (scope level: 0)
error:1: undefined: x
IDENT (name=y, address=0)
ADD_ASSIGN
IDENT (name=y, address=0)
INT_LIT 3
error:3: invalid operation: REM_ASSIGN (mismatched types float and int)
REM_ASSIGN
> Dump symbol table (scope level: 0)
Index     Name      Type      Address   Lineno    Element type
0         y         float     0         0         -
Total lines: 3
```

3.2. Output format definition

- 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("> Dump symbol table (scope level: %d)\n", table->scope_level);
printf("%-10s%-10s%-10s%-10s%-10s%s\n", "Index", "Name", "Type", "Address", "Lineno",
"Element type");
printf("%-10d%-10s%-10s%-10d%-10d",
    cur->index, cur->name,
    get_type_name(cur->type),
    cur->address, cur->lineno);
```

4. Submission

- Hand in your homework with Moodle.
- Allow only `.zip` and `.rar` formats for file compression.
- The directory organization should be exactly as follows.

```
Compiler_StudentID_HW2.zip/
└── Compiler_StudentID_HW2/
    ├── compiler_hw2.l
    ├── compiler_hw2.y
    ├── common.h
    └── Makefile
```

!!! **Incorrect format will lose 10pt.** !!!

5. Appendix: µC Specification

In this specification, the syntax is specified using Extended Backus-Naur Form (EBNF). The following table lists the operators defined in EBNF.

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

5.1 Types

A type determines a set of values together with operations and methods specific to those values.

- "int": the set of all signed 32-bit integers (-2147483648 to 2147483647)
- "float": the set of all IEEE-754 32-bit floating-point numbers
- "string": a (possibly empty) sequence of bytes
- "bool": the set of Boolean truth values denoted by the predeclared constants true and false
- ArrayType: a numbered sequence of elements of a single type (defined at Section 5.3 Declarations statements)

```
Type      = TypeName
TypeName  = "int" | "float" | "string" | "bool"
```

5.2 Expressions

```
Expression = UnaryExpr | Expression binary_op Expression
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr

binary_op  = "||" | "&&" | cmp_op | add_op | mul_op
cmp_op     = "==" | "!=" | "<" | "<=" | ">" | ">="
add_op     = "+" | "-"
mul_op     = "*" | "/" | "%"

unary_op   = "+" | "-" | "!"
```

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).

Category	Operators	Precedence
Array subscripting	`[]`	1
Unary	`+` `-` `!`	2
Multiplication	`*` `/` `%`	3
Addition	`+` `-`	4
Comparison	`<` `>` `<=` `>=` `==` `!=`	5
Logical AND	`&&`	6
Logical OR	`||`	7

Note: 1. The expression within `()` needs to be evaluate first. 2. `++` `--` `=` `+=` `-=` `*=` `/=` `%=` are invalid in an expression. 3. `!`, `&&`, and `||` are only for boolean type and boolean type does not support arithmetic operation, e.g., multiplication, addition, and comparison. 4. `%` is only for integer numbers.

Primary expressions

```
PrimaryExpr = Operand | IndexExpr | ConversionExpr
Operand     = Literal | identifier | "(" Expression ")"
Literal     = INT_LIT | FLOAT_LIT | BOOL_LIT | STRING_LIT
```

Index expressions

```
IndexExpr = PrimaryExpr "[" Expression "]"
```

Example:

```
a[i]
b[32 - y]
```

Conversions (Type casting)

A conversion changes the type of an expression to the type specified by the conversion.

```
ConversionExpr = Type "(" Expression ")"
```

Example:

```
int(3.2)
float(x + 3)
```

5.3 Statements

```
Statement =
    DeclarationStmt
    | AssignmentStmt
    | IncDecStmt
    | Block
    | IfStmt
    | WhileStmt
    | ForStmt
    | PrintStmt
```

Declarations statements

A variable declaration creates one variables, binds corresponding identifiers to them, and gives a type and an initial value.

```
DeclarationStmt = Type identifier [ "=" Expression ] SEMICOLON
                | Type identifier "[" Expression "]" SEMICOLON
```

Example:

```
int i;
float k = 3.14;
```

Assignments statements

Each left-hand side operand must be addressable.

```
AssignmentExpr = Expression assign_op Expression
AssignmentStmt = AssignmentExpr SEMICOLON
assign_op      = "=" | "+=" | "-=" | "*=" | "/=" | "%="
```

Example:

```
a = 99;
b -= c + a;
```

IncDec statements

The "++" and "--" statements increment or decrement their operands by the untyped constant 1. As with an assignment, the operand must be addressable. You can assume that the Expression in this statement must be an identifier in our assignment, i.e., valid statements are like `x++`, `i--`, and `3++`, `arr[4]++` will not appear in any test case.

```
IncDecExpr = Expression ( "++" | "--" )
IncDecStmt = IncDecExpr SEMICOLON
```

Example:

```
x++;
y--;
```

Block

A block is a possibly empty sequence of declarations and statements within matching brace brackets.

```
Block         = "{" StatementList "}"
StatementList = { Statement }
```

Example:

```
{
    x += y;
    y--;
}
```

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the `Condition` evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" Condition Block [ "else" ( IfStmt | Block ) ]

Condition = Expression
```

Example:

```
if (x > max) {
    max = x;
}
```

```
if (x < max) {
    x = max;
} else {
    max = x;
}
```

```
if (x <= y) {
    x++;
} else if (x > z) {
    y++;
} else {
    z++;
}
```

While and For statements

A "for" statement specifies repeated execution of a block. There are two forms: the iteration may be controlled by a single condition or a "for" clause. A "while" statement, on the other hand, accepts `condition`.

```
WhileStmt = "while" "(" Condition ")" Block
ForStmt   = "for" "(" ForClause ")" Block

ForClause = InitStmt ";" Condition ";" PostStmt
InitStmt  = SimpleExpr
PostStmt  = SimpleExpr
SimpleExpr = AssignmentExpr | Expression | IncDecExpr
```

Example:

```
while (a < b) {
    a *= 2;
}
```

```
for (i = 0; i < 10; i++) {
    sum += i;
}
```

Print statements

Implementation restriction: "print" need not to accept arbitrary argument types, but printing of boolean, numeric, and string types must be supported.

- "print": prints evaluated expression.

```
PrintStmt = "print" "(" Expression ")" SEMICOLON
```

Example:

```
print(x);
```

6. References

- The C Programming Language Specification: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf
- ANSI C Yacc grammar: http://www.quut.com/c/ANSI-C-grammar-y.html