CSE 340 Spring 2014 – Project 4

Due on **May 2nd, 2014 by 11:59 pm**

Abstract

The goal of this project is to give you some hands-on experience with implementing a compiler. You will write a compiler for a simple language. You will be translating source code to a simple intermediate code. The execution of the program will be done after compilation by interpreting the generated intermediate code.

1. Introduction

You will write a compiler that will read an input program and translate it to a simple intermediate code. The intermediate code will contain instructions to be executed as well as a part that represents the data of the program (symbol table). Then a virtual machine (interpreter), provided to you, will execute the intermediate code; the interpreter will read your file, load the symbol table, load the instructions, and then execute each instruction line by line.

2. Grammar

The grammar for this project is mostly a simplified form of the grammar from previous projects, with the exception of a couple of extensions.

```
PROGRAM       → VAR_SECTION BODY
VAR_SECTION   → ID_LIST ';'
ID_LIST       → id ',' ID_LIST | id
BODY          → '{' STMT_LIST '}'
STMT_LIST     → STMT STMT_LIST | STMT
STMT          → ASSIGN_STMT | PRINT_SMT | WHILE_STMT | IF_STMT | SWITCH_STMT
ASSIGN_STMT   → id '=' PRIMARY ';'
ASSIGN_STMT   → id '=' EXPR ';'
EXPR          → PRIMARY OP PRIMARY
PRIMARY       → id | num
OP            → '+' | '-' | '*' | '/'
PRINT_STMT    → print id ';'
WHILE_STMT    → 'WHILE' CONDITION BODY
IF_STMT       → 'IF' CONDITION BODY
CONDITION     → PRIMARY RELOP PRIMARY
RELOP         → '>' | '<' | '!='
SWITCH_STMT   → 'SWITCH' id '{' CASE_LIST '}'
SWITCH_STMT   → 'SWITCH' id '{' CASE_LIST DEFAULT_CASE '}'
CASE_LIST     → CASE CASE_LIST | CASE
CASE          → 'CASE' num ':' BODY
DEFAULT_CASE  → 'DEFAULT' ':' BODY
```

Appendix A at the end of this document shows the syntax diagram for each rule above.

Some highlights of the grammar are as follows:

1. Expressions are greatly simplified.
2. There is only one ID_LIST in the global scope, and it contains all the variables.
3. There is no type specified for variables. All variables are INT by default.
4. There are no functions.
5. Division is integer division, and the result of the division of two integers is an integer.
6. The *if* statement does not have an *else* section.
7. A *switch* statement is introduced.
8. A *print* statement is introduced.
9. All non-terminals are written in capital case in the grammar.
10. Terminals are written in lower case or quoted in the grammar.

3. Boolean Condition

A boolean condition takes two operands as parameters and returns a boolean value. It is used to control the execution of *while* and *if* statements.

4. Execution Semantics

All statements are executed sequentially according to the order in which they appear. An exception is made for the body of IF_STMT, WHILE_STMT, and SWITCH_STMT as explained below.

4.1. *If* statement

IF_STMT has the standard semantics:

1. The condition is evaluated.
2. If the condition evaluates to **true**, the body of the IF_STMT is executed, and then the next statement following the *if* is executed.
3. If the condition evaluates to **false**, the statement following the *if* in the STMT_LIST is executed.

These semantics apply recursively to a nested IF_STMT.

4.2. *While* statement

WHILE_STMT has the standard semantics:

1. The condition is evaluated.
2. If the condition evaluates to **true**, the body of the WHILE_STMT is executed, then the condition is evaluated again, and the process repeats.
3. If the condition evaluates to **false**, the statement following the WHILE_STMT in the

STMT_LIST is executed.

These semantics apply recursively to a nested WHILE_STMT.

4.3. *Switch* statement

SWITCH_STMT has the standard semantics:

1. The value of the switch variable is checked against each case number in order.
2. If the value of the switch variable is equal to the case number, the body of the case is executed; then the statement following the SWITCH_STMT in the STMT_LIST is executed.
3. If the value of the switch number is not equal to the number of the case being considered, the next case is considered.
4. If a default case is provided and the value does not match any of the case numbers, the body of the default case is executed, and then the statement following the SWITCH_STMT in the STMT_LIST is executed.
5. If there is no default case and the value does not match any of the case numbers, then the statement following the SWITCH_STMT in the STMT_LIST is executed.
6. If there are multiple identical case numbers which are equal to the switch variable value, only the body of the first case is executed.

These semantics apply recursively to a nested SWITCH_STMT.

Notice that the language defined by the given grammar does not have or require a break statement (like C++ or Java) but the behavior of a SWITCH_STMT is the same as the *switch* statement in Java or C++ having a *break* statement at the end of each *case* statement.

5. *Print* statement

The statement

```
print a;
```

prints the value of variable **a** at the time of the execution of the *print* statement.

6. Source code example

The following code is a valid input for a parser implementing the grammar described above.

```
a, b, c, d;
{
  a = 1;
  b = a;
  c = a + b;
  d = c * 3;
  while (d > 1) {
```

```
    print d;
    d = d -1;
  }
  if (a < 10) {
    a = 10;
  }
  b = 10;
  while (b > 1) {
    if (b > 7) {
      print b;
    }
    b = b -1;
  }
  print b;
  c = 3;
  d = 2;
  switch (c) {
    case 1: {
       print c;
    }
    default: {
       if ( d != 1 ) {
         d = 1;
       }
       d = d * 2;
       c = c - d;
       print c;
    }
  }
}
```

7. How to generate the code

The intermediate code will be a subset of p-code (described in class). As a summary, the following paragraphs describe how it looks.

7.1. Handling simple assignments

Simple assignments have an id on the left-hand side and a value (integer or id) on the right-hand side. For instance,

```
(1) w = 1; // simple assignment
(2) x = w; // simple assignment
```

Line numbers are not part of the code; they were added for reference.

Also, an assignment can have expressions on the right-hand side. An expression has only one operator and its operands. For instance,

```
(3) y = w + 1;   // assignment with expression
(4) z = 1 *  b;  // assignment with expression
```

Line numbers are not part of the code; they were added for reference.

To execute an assignment, you need to get the values of the operands; apply the operator,

if any, to the operands; and assign the resulting value of the right-hand side to the id on the left-hand side. For literals (NUM), the value is the value of the number. For variables (ID), the value is the last value stored in the variable. Initially, all variables are initialized to 0. Multiple assignments are executed one after another.

For a simple assignment the intermediate code has the LIT (for literals) or LOD (for variables) instruction followed by an STO instruction. For instance, this is the intermediate code for lines (1) and (2):

```
LIT 1, 0      ;stores the value 1 in the register 0
STO w, 0      ;stores the value in the register 0 in w
LOD w, 0      ;stores the value of w in the register 0
STO x, 0      ;stores the value in the register 0 in x
```

For an assignment with expressions, two LIT or LOD instructions (one for each operand), the OPR instruction for the operator, and an STO instruction are required to complete the assignment. For instance, the following is the intermediate code for lines (3) and (4) above:

```
LOD w, 0
LIT 1, 0
OPR 2, 0      ;addition is operation 2.
STO y, 0      ;stores the value in the register 0 in y
LIT 1, 0
LOD b, 0
OPR 4, 0      ;multiplication is operation 4
STO z, 0
```

Review the code numbers for operators in the lecture's slides.

7.2. Handling *print* statements

The *print* statement is straightforward. Its intermediate code uses an LOD instruction followed by an OPR 21, 0 instruction. For instance,

```
print a;
```

will be translated to this:

```
LOD a, 0
OPR 21,0
```

7.3. Handling *if* and *while* statements

The structure for an *if* statement is as follows:

```
if ( a < b) {
   a = 1;
}
```

The condition of the *if* statement is an expression with two operands and only one

operator. To generate the intermediate code for an *if* statement, we need to generate code for the condition, use the JMC (jump conditional) instruction, and generate the intermediate code for the STMT_LIST that corresponds to the *if* statement. The JMC instruction is crucial to the execution of the *if* statement. If the condition evaluates to true then the statements in STMT_LIST are executed. The intermediate code for the *if* statement above is as shown below. Line numbers are not part of the code; they were added for reference.

```
(1) LOD a, 0
(2) LOD b, 0
(3) OPR 12,0                ;less than is operation 12
(4) JMC #e1, false
(5) LIT 1, 0
(6) STO a, 0
(7)
```

The label `#e1` should be added to the symbol table with the value 7.

While statements use an additional intermediate instruction, JMP (jump), to allow for executing the loop body multiple times. For instance,

```
while (a < b) {
   a = 1;
}
```

Translates to:

```
(1) LOD a, 0
(2) LOD b, 0
(3) OPR 12,0
(4) JMC #e1, false
(5) LIT 1, 0
(6) STO a, 0
(7) JMP #e2, 0
(8)
```

The label `#e1` should be added to the symbol table with the value 8 and the label `#e2` should be added with the value 1.

7.4. Handling *switch* statement

You can handle the *switch* statement similarly to an *if* statement. See section 4.3 for more information.

8. Executing the intermediate code

After the intermediate code is written to a file, it needs to be executed using an interpreter. We are providing you with the interpreter to execute your intermediate code. It runs on the general machine and is included as the 'interpreter.c' file. It implements the symbol table as a map and the register as a stack.

The interpreter starts the execution in the first instruction (program counter equal to one, `pc = 1`). After an instruction is executed, the `pc` increases by one (`pc = pc+1`). The interpreter behavior is illustrated in the following pseudo-code:

```
pc_ = 1;
while (pc != NULL){
  switch (instruction[pc].name){
    case LIT: // put a literal (instruction[pc].param1) in the register
              pc = pc->next
    case LOD: // get the value (from the symbol_table) of
              // the variable instruction[pc].param1
               // put the value in the register
              pc = pc->next
    case STO: // get a value from the register
              // store the value in the variable instruction[pc].param1
              pc = pc->next
    case JMP: // get the value (from the symbol_table) of
              // the label instruction[pc].param1
              pc = value
    case JMC: // get a value from the register
              // compare the value with instruction[pc].param2
               // if they are equal
              // get the value (from the symbol_table) of
              // the label instruction[pc].param1
              pc = value
    case OPR  // get a value A from the register
              // get a value B from the register
              // evaluate A operation B
              // store the value in the register

  }
}
```

Notice that the instructions `JMP` and `JMC` modify `pc` to any value.

The interpreter receives as a parameter the name of the file that you generated containing your intermediate code and executes it. Only the results of *print* statements are shown on the screen. Optionally, the provided interpreter receives as a second parameter the label DEBUG. If the second parameter is included, it executes your intermediate code in a verbose mode. For instance,

The input file

```
a = 5;
while (a > 1) {
  a = a - 1;
  print a;
}
print a;
```

generates the following output file:

```
a
#e1=14, #e2=3
LIT 5, 0
STO a, 0
LOD a, 0
```

```
LIT 1, 0
OPR 11, 0
JMC #e1, false
LOD a, 0
LIT 1, 0
OPR 3, 0
STO a, 0
LOD a, 0
OPR 21, 0
JMP #e2, 0
LOD a, 0
OPR 21, 0
```

The first line of the output contains a list of comma-separated IDs. All IDs are automatically associated with a starting value of 0. The second line contains a list of comma-separated labels. Label names start with # and the symbol = is used to associate the label with its value. The output of the execution of the previous code is as follows:

```
$ interpreter.o output_file.txt
        4
        3
        2
        1
        1
```

If the verbose mode is enabled with the DEBUG parameter, the output is as follows:

```
$ interpreter.o output_file.txt DEBUG

* 1. Reading the file: output_file.txt.
* 2. Getting variables... done.
* 3. Getting labels... done.
* 4. Loading instructions... done.
* 5. Program running...
        1: LIT 5 0
        2: STO a 0
        3: LOD a 0
        4: LIT 1 0
        5: OPR 11 0
        6: JMC #e1 false
        7: LOD a 0
        8: LIT 1 0
        9: OPR 3 0
       10: STO a 0
       11: LOD a 0
       12: OPR 21 0
              >4<
       13: JMP #e2 0
        3: LOD a 0
        4: LIT 1 0
        5: OPR 11 0
        6: JMC #e1 false
        7: LOD a 0
        8: LIT 1 0
        9: OPR 3 0
       10: STO a 0
       11: LOD a 0
       12: OPR 21 0
              >3<
```

```
       13: JMP #e2 0 ;
        3: LOD a 0 ;
        4: LIT 1 0 ;
        5: OPR 11 0 ;
        6: JMP #e1 false ;
        7: LOD a 0 ;
        8: LIT 1 0 ;
        9: OPR 3 0 ;
       10: STO a 0 ;
       11: LOD a 0 ;
       12: OPR 21 0 ;
              >2<

       13: JMP #e2 0 ;
        3: LOD a 0 ;
        4: LIT 1 0 ;
        5: OPR 11 0 ;
        6: JMP #e1 false ;
        7: LOD a 0 ;
        8: LIT 1 0 ;
        9: OPR 3 0 ;
       10: STO a 0 ;
       11: LOD a 0 ;
       12: OPR 21 0 ;
              >1<
       13: JMP #e2 0 ;
        3: LOD a 0 ;
        4: LIT 1 0 ;
        5: OPR 11 0 ;
        6: JMP #e1 false ;
       14: LOD a 0 ;
       15: OPR 21 0 ;
              >1<
* 6. Program ends
```

9. Requirements

1. Write a compiler that generates intermediate code from source code.
2. Use the provided interpreter to execute the intermediate code. You can assume that there are no syntax or semantic errors in the source code used as input.
3. Language: You should use C or C++ for this assignment. Any language other than C or C++ is not allowed for this project.
4. Platform: As with previous projects, the reference platform is the general machine.

10. Submission

Submit your code on Blackboard by the deadline. Submission by email or other forms are NOT accepted. Your submission should be a ZIP file containing a folder named **"assignment"** with your code (all files) and a makefile.

11. Grading

Make sure you test your code extensively with input programs that contain all statements and combinations of them. Share your test cases on the Discussion board and test your compilers using the test cases shared by your classmates.

12 Bonus: replaces any project grade for projects 1 or 2

Support the following grammar:

```
PROGRAM       → VAR_SECTION BODY
VAR_SECTION   → 'VAR' INT_VAR_DECL ARRAY_VAR_DECL
INT_VAR_DECL  → ID_LIST ';'
ARRAY_VAR_DECL → ID_LIST ':' 'ARRAY' '[' num ']' ';'
ID_LIST       → id ',' ID_LIST | id
BODY          → '{' STMT_LIST '}'
STMT_LIST     → STMT STMT_LIST | STMT
STMT          → ASSIGN_STMT | PRINT_SMT | WHILE_STMT | IF_STMT | SWITCH_STMT
ASSIGN_STMT   → VAR_ACCESS '=' EXPR ';'
VAR_ACCESS    → ID | ID '[' EXPR ']'
EXPR          → TERM ('+' | '-') EXPR
EXPR          → TERM
TERM          → FACTOR ('*' | '/') TERM
TERM          → FACTOR
FACTOR        → '(' EXPR ')'
FACTOR        → num
FACTOR        → VAR_ACCESS
PRINT_STMT    → print VAR_ACCESS ';'
WHILE_STMT    → 'WHILE' CONDITION BODY
IF_STMT       → 'IF' CONDITION BODY
CONDITION     → EXPR RELOP EXPR
RELOP         → '>' | '<' | '!='
SWITCH_STMT   → 'SWITCH' VAR_ACCESS '{' CASE_LIST '}'
SWITCH_STMT   → 'SWITCH' VAR_ACCESS '{' CASE_LIST DEFAULT_CASE '}'
CASE_LIST     → CASE CASE_LIST | CASE
CASE          → 'CASE' num ':' BODY
DEFAULT_CASE  → 'DEFAULT' ':' BODY
```

The "[" is used for arrays and the "{" is used for body. Assume that all arrays are integer arrays and are indexed from 0 to *size* - 1, where *size* is the size of the array specified in the VAR_SECTION after the `array` keyword and between "[" and "]".

The interpreter that we have provided for the regular assignment will not be enough for the bonus, so you will need to modify it to support arrays. Submit all code files for the bonus project (including the modified interpreter.c).

13. Bonus: Grading

You should submit the bonus assignment in a separate folder from the main submission. It should be in a folder named **"bonus"** and should be included in the same ZIP file as your main submission.

Appendix A. Grammar for Assignment #4

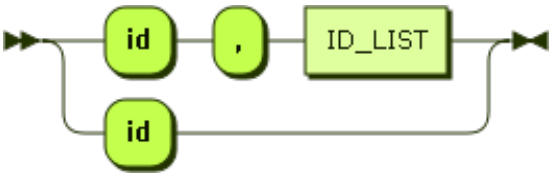
PROGRAM:


```
PROGRAM  → VAR_SECTION BODY
```

VAR_SECTION:


```
VAR_SECTION → ID_LIST ';'
```

ID_LIST:

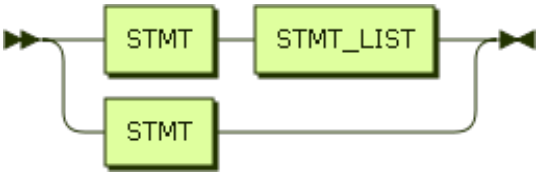



```
ID_LIST  → id ',' ID_LIST | id
```

BODY:

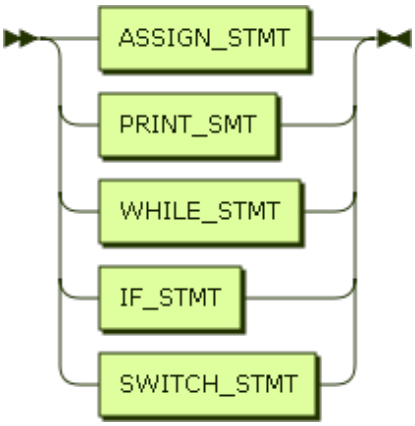

```
BODY     → '{' STMT_LIST '}'
```

STMT_LIST:



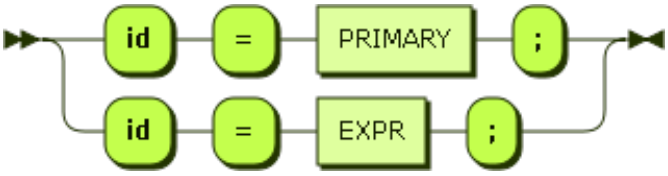

```
STMT_LIST → STMT STMT_LIST | STMT
```

STMT:



STMT → ASSIGN_STMT | PRINT_SMT | WHILE_STMT | IF_STMT | SWITCH_STMT

ASSIGN_STMT:

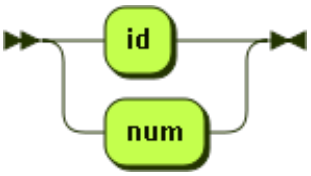


ASSIGN_STMT → id '=' PRIMARY ';' | id '=' EXPR ';'

EXPR:

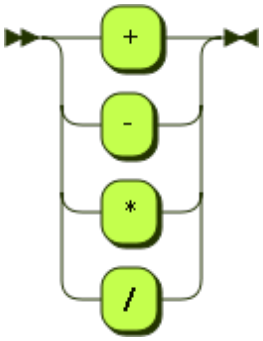
EXPR → PRIMARY OP PRIMARY

PRIMARY:



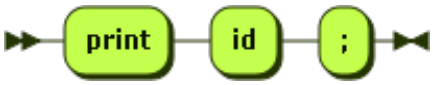
PRIMARY ::= id | num

OP:



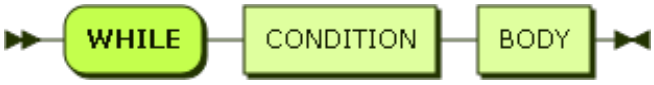
OP → '+' | '-' | '*' | '/'

PRINT_STMT:



PRINT_STMT → print id ';'

WHILE_STMT:



WHILE_STMT → 'WHILE' CONDITION BODY

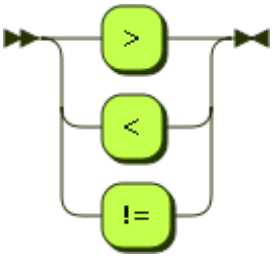
IF_STMT:

IF_STMT → 'IF' CONDITION BODY

CONDITION:

CONDITION → PRIMARY RELOP PRIMARY

RELOP:

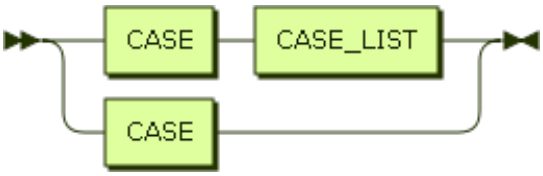


RELOP → '>'| '<' | '!='

SWITCH_STMT:

SWITCH_STMT → 'SWITCH' id '{' CASE_LIST '}' | 'SWITCH' id '{' CASE_LIST DEFAULT_CASE '}'

CASE_LIST:

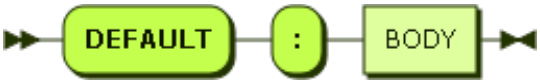


CASE_LIST → CASE CASE_LIST | CASE

CASE:

CASE → case num ':' BODY

DEFAULT_CASE:



DEFAULT_CASE → 'DEFAULT' ':' BODY