CSE 340 Principles of Programming Languages
Fall 2014
Programming Assignment #4

Abstract

The goal of this project is to give you some hands-on experience implementing a compiler for a small programming language. The compiler will be able to translate source code to a simple intermediate code. The execution of the program will be done after compilation by interpreting the generated intermediate code using a virtual machine provided to you.

1. Introduction

You will write a compiler that will read an input program and will translate it to a simple intermediate code. This includes: (a) developing a descendent recursive parser for a given grammar and (b) implementing, inside the parser, the process for intermediate code generation.

The source language and the intermediate language will be described during the lectures; and several examples of programs using the source language will be provided as well as a step-by-step description of the intermediate code generation process. You can assume that the source code used as input to test your assignment will not have lexical, syntactic, or semantic errors. Handling errors is not included in this assignment; however, bonus points are given if you decide to implement it. See section 10 for a description of bonus points' tasks.

The intermediate code will also be described during the lectures. A virtual machine (interpreter) is provided to you to execute the intermediate code. The virtual machine has a graphical user interface that allows you to execute the code line by line to follow up the execution process.

2. Advice

Some highlights for you:

- If you **attend the lectures** and complete the after-class activities that we will start doing during the lectures, you will be able to complete the code generation part of this assignment in few days.
- **Start working** in the developing of the parser for the new grammar and/or in the bonus points' tasks.
- Attend **office hours** with me, with the graduate TA, or with the undergraduate TA. You have three persons available to help you, take advantage of that.
- Attend the **programming workshops** that we will be running this and next week.
- Participate in the **discussion board**, join the discussions posting and answering questions.
- **Do not procrastinate!** Asking for help the last week or some days before the deadline will not work.

3. Language

The language to be used for writing source code is defined by the following grammar:

```
<PROGRAM>    → '{' <BODY> '}'
<BODY>    → {<PRINT>';'|<ASSIGNMENT>';'|<VARIABLE>';'|<WHILE>|<IF>|<SWITCH>|<RETURN>';'}
<ASSIGNMENT> → identifier '=' <EXPRESSION>
<VARIABLE>   → ('int'|'float'|'boolean'|'char'|'string'|'void')identifier
<WHILE>      → 'while' '(' <EXPRESSION>  ')' <PROGRAM>
<IF>         → 'if' '(' <EXPRESSION>  ')' <PROGRAM> ['else' <PROGRAM>]
<RETURN>     → 'return'
<PRINT>      → 'print' '(' <EXPRESSION> ')'
<EXPRESSION> → <X> {'|' <X>}
<X>          → <Y> {'&' <Y>}
<Y>          → ['!'] <R>
<R>          → <E> {('>'|'<'|'=='|'!=') <E>}
<E>          → <A> {('+'|'-') <A>}
<A>          → <B> {('*'|'/') <B>}
<B>          → ['-'] <C>
<C>          → integer | octal | hexadecimal | binary | true | false |
               string | char | float | identifier|'(' <EXPRESSION> ')'
<SWITCH>    → 'switch' '(' id ')' '{' <CASES> [<DEFAULT>] '}'
<CASES>     → ('case' (integer|octal|hexadecimal|binary) ':' <PROGRAM>)+
<DEFAULT>   → 'default' ':' <PROGRAM>
```

This grammar is similar **but not the same** than the grammar we used for assignment #2 and assignment #3. You are required to write a descendent recursive parser for this new grammar.

4. Source code example

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

```
{
  int a; int b; int c; int 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+10);
    }
  }
  return;
}
```

5. Analysis

As reviewed in class, the compiler has two stages: analysis and intermediate code generation. The analysis stage is composed of three phases: lexical analysis, syntactic analysis, and semantic analysis.

5.1. Lexical Analysis

During the lexical analysis a code written in our language is transformed to an array of tokens. To fulfill this phase you have two options: (a) you can reuse the Lexer you created in your assignment #1, modifying it to include the word "*default*" in the category "*keyword*"; or (b) you can use the Lexer that we are providing.

The Lexer that we are providing is located in the file lexer.jar and is published jointly with this document. To use the provided Lexer you should include the file lexer.jar in your project and import the classes Lexer and Token as follows (unless you have your project in a package called A4):

```
import A4.Lexer
import A4.Token
```

Then you can use the following code to get a vector of Tokens from a text:

```
String text;
// ...
Lexer lex = new Lexer (text);
lex.run();
Vector<Token> tokens = lex.getTokens();
```

You used this code in assignment #1, assignment #2, and assignment #3; thus, we expect that you are familiar with it: you read a file line by line, concatenate the lines using '\n' as a delimiter, and store them in a String variable; then, pass that variable to the constructor of the class Lexer as a parameter; then, recognize the tokens from the input by calling the method run(); finally, you can get those tokens calling the method getTokens(), which return a Vector of Token objects.

5.2. Syntactic Analysis

A requirement for this assignment is to show competency programming a recursive descendent parser. Thus, **the first outcome of this assignment is the implementation of a parser for the grammar described before**.

Some highlights of the parser are as follows:

- The grammar is similar to the grammar used in assignment #2. If you achieve successfully the implementation of assignment #2 you could reuse some of your code, if not, start from scratch a new parser. This is your second chance to show proficiency implementing a parser. **Attend the programming workshops; they will help you to have a parser done in a couple of hours.**

- It is not required to implement error handling or error recovery.
- It is not required to visualize the parse tree (as was requested in assignment #2).

5.3. Semantic Analysis

Semantic analysis is not required.

6. Execution

All statements are executed sequentially according to the order in which they appear. An exception is made for the body of <if>, <while>, and <switch> as explained below.

6.1. *If* statement

If statement has the standard behavior:

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

These behaviors apply recursively to a nested *if* statement

6.2. *While* statement

While statement has the standard behavior:

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

These behaviors apply recursively to a nested *while* statement.

6.3. *Switch* statement

Switch statement has the standard behavior:

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* statement in the program 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* statement in the <program> 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* statement in the <program> 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 behaviors apply recursively to a nested *switch* statement.

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* statement is the same as the *switch* statement in Java or C++ having a *break* statement at the end of each *case* statement.

6.4. *Print* statement

The statement

```
print (...);
```

prints the value of any expression at the time of the execution of the *print* statement.

6.5. *Assignment* statement

The language supports the use of assignment to store values in variables. The expressions in the right side of the assignment support logical, relational, and arithmetical operators and their precedence as described in the grammar.

6.6. Variable declaration

The language also supports declaration of variables. Only one variable can be declared each time. The types supported for variables include: int, float, boolean, char, string, and void. The semantic for the use of types will be similar to the one used in Java and described in the assignment #3. No semantic validation is required, all code used to test your compiler is going to be correct.

7. How to generate the code

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

7.1. Variables

Variable declarations are included in the intermediate code as comma-separated lines, involving: type, variable name, scope (global is the only scope we will be using), and initial value (0, “”, or ‘’). For instance:

```
a, int, global, 0
```

7.2. Handling assignments

Simple assignments have an identifier on the left-hand side and a value or an identifier on the right-hand side of the = symbol. Assignments with expressions have an identifier on the left-hand side and an expression on the right-hand side. For instance,

```
(1) w = 1; // simple assignment
(2) x = w; // simple assignment
(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 identifier on the left-hand side. For literals, the value is the value of the number. For variables, 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) above:

```
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 all operations in the lecture’s slides. Examples of the code generation process for expressions with multiple operands and operators are provided in the lecture’s slides.

7.3. 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     ; print is operation 21
```

A print statement with an expression as a parameter, such as the following:

```
print (a + b);
```

will be translated to this:

```
LOD a, 0
LOD b, 0
OPR 2, 0     ; addition is operation 2
OPR 21,0     ; print is operation 21
```

7.4. 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 a boolean expression. 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 statements that correspond to the body of 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 the if's body 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. An example of the *if* statement with its *else* statement is described in the lecture's slides.

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
```

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.

7.5. Handling *switch* statement

You can handle the *switch* statement similarly to an *if* statement. See section 6.3 for more information.

7.6. Handling *return* statement

The *return* statement is straightforward. Its intermediate code is an OPR 1, 0 instruction.

8. Executing the intermediate code

After the intermediate code is generated, it needs to be executed using an interpreter. We are providing you with the interpreter to execute your intermediate code. It is included as the 'interpreter.jar' file. It implements the symbol table as a hastable 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` is increased by one (`pc = pc+1`). Notice that the instructions `JMP` and `JMC` modify `pc` to any value. 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

  }
}
```

The interpreter loads an intermediate code file, generated by the compiler, and executes it. Only the results of *print* statements are shown on the screen. For instance, the input file:

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

compiles to the following code:

```
a,int, global, 0
#e1, label, 14
#e2, label, 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 part of the output contains a list of variables. All variables are automatically associated with a starting value of 0 or empty. The second and third lines contain labels. Label names start with # and they are associated with a value. The output of the execution of the previous code is as follows:

```
        4
        3
        2
        1
        1
```

A verbose description of how that output is created is as follows:

```
* 1. Loading variables and labels... done.
* 2. Loading instructions... done.
* 3. 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: 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 ;
      >2<

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 ;
        >1<

13: JMP #e2 0 ;
 3: LOD a 0 ;
 4: LIT 1 0 ;
 5: OPR 11 0 ;
 6: JMC #e1 false ;
14: LOD a 0 ;
15: OPR 21 0 ;
      >1<
```

```
* 4. Program ends
```

9. Submission

Submit your code on Blackboard by the deadline. Submission by email or other forms are NOT accepted.

Create a ZIP file, using the following naming convention: **Firstname_Lastname_P4.zip**. This file should contain **two folders: (a) assignment, and (b) bonus**. On each folder put your Java source code for:

- Gui.java – We are providing you a graphical user interface, similar to the one used for previous projects. This GUI include 4 tabs, you can delete the first 3 tabs if you are not planning to work in the bonus points; see section 10 to learn about bonus point.
- Parser.java - The class in which you should implement your parser.
- CodeGenerator.java - The class in which the code generation process should be implemented.
- Other java files, such as Lexer.java and Token.java, if you decide to use your own Lexer.

Include a compiled version of your application named

- CSE340.jar.

The code in the folder "assignment" will be used to grade your assignment. The code in the folder "bonus" will be used to grade the bonus points. Thus, code related to bonus points does not jeopardize your grade for the assignment 4.

10. Bonus points

The following tasks provide bonus points. You can do both or only one of them as your choice.

10.1 *SWITCH* statement

The code generation process for all statements, except the *SWITCH* statement, will be described step by step in the following lectures. Attending the lectures will give you all what you need to work on it. Generating intermediate code for SWITCH statement is not required as part of the assignment; but figuring out the intermediate code for the *SWITCH* statement is a task that you can work on and it gives you 10% bonus points. Section 6.3 can give you an idea about how to proceed.

10.2 Previous assignments

10% bonus points will be given if you:

- Include the code generation process in the graphical user interface used in assignment #3, which has tabs for showing the Tokens table, the parser tree, and the symbol table.
- Include in your compiler (parser of assignment #4) the error handling and error recovery process as it was implemented in assignment #2.
- Include semantic analysis, as it was implemented in assignment #3.

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 compiler using the test cases shared by your classmates. Test cases will be provided in the lecture slides and we will be discussing them in class.

If you have any question, contact us (me or the TAs) to clarify them. And remember to actively participate in the discussion board.