CMPE 230: Systems Programming
Homework 1 Report

Aral Dörtoğul
2018402108

May 2, 2021

Contents

Listings

1 Introduction

In this homework, a new language called `myLang` is defined and a translator called `mylang2ir` is developed for the language. `mylang2ir` gets `myLang` source code as input and through synthetical/lexical analysis, it generates it's corresponding LLVM intermediate representation, which is usually used to generate code for various target architectures. The translator is implemented using Java and the output of the translator is for LLVM version 3.3.

Figure 1: An illustration of `myLang2ir`'s duty

`myLang` is a basic language with the following features:

1. Variables in `myLang` are integer variables, and their default value is 0 (i.e. they are not initialized).
2. The executable statement in `myLang` can only be one of these:
   - one-line statements
   - while-loop
   - if-compound statement
3. A one-line statement is either an assignment statement or print statement. Print statement prints the value of its expression to the console.
4. In `myLang`, a special function `choose(expr1, expr2, expr3, expr4)` is defined. It returns `expr2` if `expr1` is equal to 0, `expr3` if `expr1` is positive, or `expr4` if `expr1` is negative.
5. Everything after the `#` character is regarded as comments.
6. Print statement has the following syntax:

   ```
   print(expr)
   ```

7. If statement has the following syntax:

   ```
   if (expr) {
    ...
    ...
   }
   ```

8. While statement has the following syntax:

   ```
   while (expr) {
    ...
   ```

```
  ...
}
```

9. The condition expression of the if and while blocks are considered to be `true` if the expression is equal to 0. If it is not equal to 0, it means the expression is `false`.
10. There can not be nested if/while blocks in `myLang`.

2 Implementation

All the things that the translator should do can be divided into four parts:

2.1 Lexing

The lexer (also called the scanner or tokenizer) is the phase that breaks up a string (the source code) into a list of tokens. A token is the smallest meaningful unit in the language. Variable names, keywords, constants, and seperators like braces are all examples of tokens.

All the possible tokens of `myLang` is listed below:

- Special keywords:
  - `if`
  - `while`
  - `choose`
  - `print`
- Operators:
  - Assignment operator '='
  - Addition '+'
  - Subtraction '-'
  - Multiplication '*'
  - Division '/'
- Seperators:
  - Left parentheses '('
  - Right parentheses ')'
  - Left curly braces '{'
  - Right curly braces '}'

  - Comma ‘,’
- Identifiers (Variables)
- Literals (Integer)

In order to represent these tokens in the scripts, a class called `Token` is defined:

```
public class Token {
  // SPECIAL KEYWORDS
  public static final int _if = -11;
  public static final int _while = -12;
  public static final int _choose = -13;
  public static final int _print = -14;

  // OPERATORS
  public static final int _assgn = -21;
  public static final int _add = -22;
  public static final int _sub = -23;
  public static final int _mult = -24;
  public static final int _div = -25;

  // SEPERATORS
  public static final int _lpar = -31;
  public static final int _rpar = -32;
  public static final int _lcurl = -33;
  public static final int _rcurl = -34;
  public static final int _comma = -35;

  // IDENTIFIERS
  public static final int _variable = -41;
  public static final int _tempvar = -42;

  // LITERALS
  public static final int _integer = -51;

  // the type of the token
  public int type;

  // the value of the token (null if its not a variable, integer, tempvar)
  public String value;

  public Token(int type, String value) {
    this.type = type;
    this.value = value;
  }
  public Token(int type) {
    this.type = type;
    this.value = null;
  }
  ...
}
```

Listing 1: `Token` class

Each `Token` object has a `type` (initiliazed to one of the public static constants defined in the class) and `value`. If the `Token` object is a variable, its value is the name of the variable, if it is an integer literal, its value is the String representation of the integer value. If it’s type is not one of these, `value` is `null`.

In `mylang2ir`, source code is tokenized line by line using a method called `ArrayList<Token> Token.lex(String input)`:

```
public static ArrayList<Token> lex(String input) throws SyntaxErrorException {
  ArrayList<Token> result = new ArrayList<Token>();
  int i = 0;
  while(i < input.length()) {
    switch(input.charAt(i)) {
    case '(': result.add(new Token(Token._lpar)); i++;
      break;
    case ')': result.add(new Token(Token._rpar)); i++;
      break;
    case '{': result.add(new Token(Token._lcurl)); i++;
      break;
    case '}': result.add(new Token(Token._rcurl)); i++;
      break;
    case '+': result.add(new Token(Token._add)); i++;
      break;
    case '-': result.add(new Token(Token._sub)); i++;
      break;
    case '*': result.add(new Token(Token._mult)); i++;
      break;
    case '/': result.add(new Token(Token._div)); i++;
      break;
    case '=': result.add(new Token(Token._assgn)); i++;
      break;
    case ',': result.add(new Token(Token._comma)); i++;
      break;
    default:
      if(Character.isWhitespace(input.charAt(i))) i++;
      else {
        if (Character.isLetter(input.charAt(i)) || input.charAt(i) == '_') {
          String variableName = "" + input.charAt(i);
          i++;
          while(i != input.length() && (Character.isLetterOrDigit(input.charAt(i))
     || input.charAt(i) == '_')) {
            variableName += input.charAt(i);
            i++;
          }
          switch (variableName) {
          case "if": result.add(new Token(Token._if)); break;
          case "while": result.add(new Token(Token._while)); break;
          case "choose": result.add(new Token(Token._choose)); break;
          case "print": result.add(new Token(Token._print)); break;
          default:
            result.add(new Token(Token._variable, "v_" + variableName));
          }

        } else if (Character.isDigit(input.charAt(i))) {
          String NumVal = "" + input.charAt(i);
          i++;
          while (i != input.length() && Character.isDigit(input.charAt(i))) {
            NumVal += input.charAt(i);
            i++;
          }
          result.add(new Token(Token._integer, NumVal));
        } else if (input.charAt(i) == '#')    // Anything after a '#' is
    considered to be a comment.
          return result;
        else
          throw new SyntaxErrorException(); // Unknown token
      }
```

```
58        break;
59      }
60    }
61    return result;
62  }
```

Listing 2: `lex()`, the lexer of the program

One thing to note about this method is that when a variable name is encountered, it is tokenized with the name `v_<variableName>`. This way, variables and temporary variables cannot be confused in LLVM-IR code because all the variables start with `v_` and all the temporary variables in the LLVM-IR start with `t`. For instance, if a variable `t1` is defined in .my script, it can not be confused with a temporary variable called `t1` using this technique.

In order to make things simpler about the `choose` function, a class called `Choose` is defined, which is a subclass (child) of the class `Token`. This class has additional `Arraylist<Token>` lists for its 4 arguments, which holds the tokens of the arguments.

In addition to `lex()`, an additional method called `tokenizeChoose()` is used to combine all the tokens of a `choose` function (`choose` keyword, parantheses, commas, argument tokens) into a single `Choose` token. This method is called after tokenziing an input String. The method is as follows:

```
1  private static Choose tokenizeChoose(ListIterator<Token> itr) throws
       SyntaxErrorException {
2    Choose result = new Choose(Token._choose);
3    if (itr.hasNext()) {
4      Token first = itr.next();
5      if (first.type != Token._lpar) throw new SyntaxErrorException(); // "choose"
       is not followed by '('
6      itr.remove();
7    } else throw new SyntaxErrorException(); // "choose" is followed by nothing.
8    int commaCount = 0, openLeftParentheses = 1;
9
10   while(itr.hasNext()) {
11     Token next = itr.next();
12     itr.remove();
13     if (next.type == Token._comma) {
14       commaCount++;
15       if (commaCount > 3) throw new SyntaxErrorException(); // There are more than
        3 commas inside choose.
16       continue;
17     }
18     if (next.type == Token._rpar) {
19       if (--openLeftParentheses == 0) break;
20     } else if (next.type == Token._lpar)
21       openLeftParentheses++;
22     if (next.type == Token._choose) {
23       Choose choose = tokenizeChoose(itr);
24       result.tokens_of_arg.get(commaCount).add(choose);
25     } else
26       result.tokens_of_arg.get(commaCount).add(next);
27   }
28   if (openLeftParentheses != 0 || result.tokens_of_arg.get(0).isEmpty() || result.
       tokens_of_arg.get(1).isEmpty() ||  result.tokens_of_arg.get(2).isEmpty() ||
       result.tokens_of_arg.get(3).isEmpty())
```

```
29      throw new SyntaxErrorException();
30    return result;
31  }
```

Listing 3: `tokenizeChoose()` method, a lexer special for `Choose` token

With these functions, we obtain simple lists of tokens that are ready to be parsed.

2.2 Parsing

After acquiring the tokens, they are divided into grammatical parts and identified using parsing. In `myLang`, executable statements can be assignment, while, if, print or closing curly braces statement. For each new line of input, a method called `ParseLine` is called:

```
1  private static char ParseLine(ArrayList<Token> tokens) throws SyntaxErrorException
       {
2    if(!tokens.isEmpty()) {
3      Token initial = tokens.get(0);
4      if (initial.type == Token._variable)    // Assignment line
5        return parseAssignment(tokens);
6
7      else if (initial.type == Token._if)     // If line
8        return parseIf(tokens);
9
10     else if (initial.type == Token._while)    // While line
11       return parseWhile(tokens);
12
13     else if (initial.type == Token._print)    // Print line
14       return parsePrint(tokens);
15
16     else if (initial.type == Token._rcurl) {  // While/If closing line
17       if (!curlyBracesOpen || tokens.size() > 1) throw new SyntaxErrorException();
        // If there is no open curly braces or the closing curly braces line
       continues with other tokens.
18       return '}';
19
20     } else throw new SyntaxErrorException(); // Statements of other forms
21   } else    // Empty line
22     return 'e';
23 }
```

Listing 4: `ParseLine()`, the general parser of `mylang2ir`

This method parses `myLang` lines by looking at the first token. The type of the first token is the key factor of determining the type of statement. If it is a variable, then the line must be an assignment statement. Similarly, if it is `if`, `while`, or `print`, the statement must be the corresponding statement types.

It is crucial to explain how expressions are handled. An expression is defined as an infix string of operands (variable, temporary variable, or choose token) and operations (+, -, *, /, =). In order to evaluate expressions, they are transformed into postfix notation (using the method `infixToPostfix()`). Postfix notation is very useful because

parentheses are removed from expressions without affecting the operator precedence and the notation itself is easier to convert to LLVM-IR code.

For more details on parsing (such as how a `choose` function or a while loop is parsed), please check the source code which is full of regular and JavaDoc comments on methods.

2.3 Syntax Error Handling

Syntax checking is done both in lexing and parsing parts. The situations where syntax error is detected is written in JavaDoc comments of the methods in detail. For example, during lexing, if an unknown token is encountered, it is regarded as a syntax error (e.g. `'&'` is not defined). When a syntax error is detected at line `n`, an exception called `SyntaxErrorException` is thrown immediately and `mylang2ir` writes the LLVM code which prints out `"Line n:  syntax error"`.

```
1  private static void SyntaxError(String output_file_name) throws
       FileNotFoundException {
2    PrintStream output = new PrintStream(new File(output_file_name));
3
4    output.println("; ModuleID = 'mylang2ir'");
5    output.println("declare i32 @printf(i8*, ...)");
6    output.println("@print.str = constant [23 x i8] c\"Line %d: syntax error\\0A
       \\00\"");
7    output.println();
8    output.println("define i32 @main() {");
9    output.println("\tcall i32 (i8*, ...)* @printf(i8* getelementptr ([23 x i8]*
       @print.str, i32 0, i32 0), i32 " + lineCount + " )");
10   output.println("\tret i32 0");
11   output.println("}");
12   output.close();
13 }
```

Listing 5: `SyntaxError()`, the output file generator when a syntax error is detected

2.4 LLVM-IR Code and Output File Generation

While parsing, the corresponding LLVM-IR code is generated and stored in ArrayLists of Strings at the same time. Since LLVM-IR uses static single assignment (SSA) based representation, temporary variables are required and this translator creates temp. variables in the form of: `'t' + <temp var count>`. If there is no syntax error, these IR statements are printed using `printIR()` method:

```
1  private static void printIR(String output_file_name) throws FileNotFoundException
       {
2    PrintStream output = new PrintStream(new File(output_file_name));
3    output.println("; ModuleID = 'mylang2ir'");
4    output.println("declare i32 @printf(i8*, ...)");
5    output.println("@print.str = constant [4 x i8] c\"%d\\0A\\00\"");
6    output.println();
7    output.println("define i32 @main() {");
8
9    if (!IRvariabledeclaration.isEmpty()) {
10     for (String line : IRvariabledeclaration)
```

```
11        output.println(line);
12      output.println();
13    }
14    if (!IRvariableinit.isEmpty()) {
15      for (String line : IRvariableinit)
16        output.println(line);
17      output.println();
18    }
19    for (String line : IRstatements)
20      output.println(line);
21
22    output.println("\tret i32 0");
23    output.println("}");
24    output.close();
25  }
```

Listing 6: `printIR()`, the output file generator

3 Conclusion and Future Work

With this homework, I implemented a compiler for a language from scratch, and I gained insight about how compilers work in general (lexing, parsing etc.). Even though the result is satisfactory, it can be improved further.

First, all the token lists are defined as `ArrayList<Token>`, and sometimes these lists are passed from a method to another or occassionally list items are added (subtracted) to (from) the lists. This pass-by-value process and the list operations may cause the time complexity of `mylang2ir` to increase drastically when compiling large files with long list of tokens. Instead of using `ArrayLists`, `LinkedLists` could have been preferred in some circumstances as a substitute. Time complexity wasn't an important concern in this homework because the translator is going to be used for small files only (at least in the near future). However, it can definitely be improved.

Next, the syntax errors detection mechanism is implemented such that it can not detect a type of syntax error (e.g. forgotten `'}'` after `if`) if it is not checked explicitly. So, this may lead to some undetected syntax errors if a specific type of error is not checked throughout the translation process. While implementing the error detection, I tried my best to think of all the syntax error possibilities, but still there may be some forgotten occassions.

To conclude, I enjoyed implementing `myLang2ir` and it certainly had a postiive impact on my systems programming skills.