CMPE230 - Project 1 Report

Hüseyin Türker Erdem - 2017400213

Halil Burak Pala - 2019400282

Overview

In this project, we are supposed to design a compiler which converts a specified language "mylang" to an LLVM intermediate representation code. In this compilation process, the compiler is supposed to detect syntax errors on the source code.

General Approach

While trying to find a solution for this task, we noticed that expressions can be anywhere in the source code. So, we had to discover a smart approach at this point to construct the whole project. Then we created methods for this purpose, which is explained in detail below. At this point, we slightly changed the general process to a try catch block and created our own exception, thus we can easily go through the code and debug. Finally, we fixed little bugs in the code and created **"makefile"**.

Methodology

To be more precise, we created a BNF structure as follows:

```
<expr> => <term><moreterms>
<moreterms> => "+"<term><moreterms>|"-"<term><moreterms>|""
<term> => <factor><morefactors>
<morefactors> => "*"<factor><morefactors>|"/"<factor><morefactors>|""
<factor> => <num>|<id>|"("<expr>")"|<choose>
<choose> => "choose("<expr1>"," <expr2>"," <expr3>"," <expr4>")"
```

Here, `<num>` and `<id>` are defined as typical integers and alphanumeric strings (except special keywords such as "while", "print", etc.) respectively. At this point, we decided not to create classes for each one of these types, instead we implemented boolean methods that return if the lexemes, which are given as a single parameter, construct that type of expression or not.

After definition of these methods, we started scanning the whole input file to determine how many variables will be stored in the whole code, thus we can allocate enough memory on LLVM. Then, the whole input source code is scanned again into a 2D-`ArrayList` of `String`'s. In this `ArrayList`, all the lines are kept as lexemes, i.e. each word or each special character now represents a single variable and each line is now stored as an `ArrayList` of these types. We did not create a new class for this purpose, instead used `String` class.

```
58
59          for(String line : lines){
60                  ++lineNum;
61              StringTokenizer tokenizer = new StringTokenizer(line, "=+-*/(){}#,", true);
62              while(tokenizer.hasMoreTokens()){
63                  String var = tokenizer.nextToken();
64                  m = r.matcher(var);
65                  // We control if the found variable candidate is one of our reserved words. If not so
66                  // we add it to our variables Set.
67                  if(m.find()){
68                      if(!var.equals("while") && !var.equals("if") && !var.equals("choose") && !var.equals("print")){
69                          variables.add(var);
70                      } else{
71                          var = tokenizer.nextToken();
72                          if(!var.equals("(")){
73                              output.println(printSyntaxError(lineNum));
74                              return;
75                          }
76                      }
77                  }
78              }
79          }
```

The code segment above checks characters on a line and seperates each lexeme as explained above. Here, we had to check if there is any variable named “while”, “if”, “choose” or “print”. If this is the case, the compiler is supposed to throw exception.

Now the tricky part begins. In a `for` loop, we used a tokenizer to seperate everything. For example, the lines below:

```
while(var1-var2){
    var1 = var2*2
    #this is comment
}
```

will be processed and converted to the `ArrayList` of `String`s as below:

```
[“while”,”(”,”var1”,”-”,”var2”,”)”,”}”]
[“var1”,”=”,”var2”,”*”,”2”]
[]
[“}”]
```

As one can see, each lexeme is seperated and comment lines are ignored in that `for` loop. Then, global `String irCode` is initialized so as to LLVM can process it. If the source code includes any choose funciton, we define it at the beginning of the `irCode`.

Final part in the main function is trying to convert source code into IR code. If the source code is impossible to be compiled somehow, then we throw exception and catch it. In this case, we replace the IR code with a new one and print that to the output file, which gives error message when run on LLVM. If there is not any syntax errors, then we regularly update `irCode` and print it to the output file.

The control flow of syntax errors is done by recursive methods, which are implemented based on the BNF structure explained above. These methods are the base of our code. We can easily check if an

`ArrayList` of lexemes describe an expression or not with `expression(ArrayList<String>,boolean,Queue<String>)` method.

First parameter is lexemes that is supposed to be processed.

Second parameter is a flag that controls if this function is called from a choose function. If so, then the process changes a little bit, explained below.

Third parameter stores the postfix state of the given lexemes. Initially, an empty queue is passed to the method and if the method returns true (given lexemes describe an expression), then empty queue is filled with proper lexemes and processed to update `irCode`. If the second parameter is given true into the method, then the postfix do not include the expressions of choose seperately, instead includes choose and its expressions as a whole. This is done for calculating postfix expressions in a proper way.

```
1115    /**
1116     * Checks whether given list of lexemes obeys the syntax of <expression> in MyLang.
1117     * @param lexemes A list of lexemes.
1118     * @param fromChooseCall Whether this function is called from a choose function. If it is so
1119     * the lexemes are not added to postfix.
1120     * @param postfix Postfix representation of an expression which can be updated by
1121     * this function.
1122     * @return Whether given list of lexemes obeys the syntax of <expression> in MyLang.
1123     */
1124    public static boolean expression(ArrayList<String> lexemes, boolean fromChooseCall, Queue<String> postfix){
1125        // Expression needs to be in the form '<term> <moreterms>'. We need to seperate them.
1126        ArrayList<String> termCandidate = new ArrayList<String>();
1127        ArrayList<String> moretermsCandidate = new ArrayList<String>();
1128        int i = 0;
1129        String lexeme;
1130        while(i < lexemes.size()){
1131            // To do this seperation, we need to add everything until a +, - or ( to our
1132            // term candidate.
1133            lexeme = lexemes.get(i);
1134            if(!lexeme.equals("+") && !lexeme.equals("-") && !lexeme.equals("(")){
1135                termCandidate.add(lexeme);
1136            }
1137
1138            // There can be something between the parantheses. In that case, we should
1139            // take everything between this parantheses and add them to our term candidate.
1140            else if(lexeme.equals("(")){
1141                // This process is done by help of a stack. Everytime we see an opening paranthesis,
1142                // we add some mark to our stack. Everytime we see a closing paranthesis, we pop this
1143                // mark. When the stack becomes empty, that means no paranthesis left.
1144                termCandidate.add(lexeme);
1145                Stack<String> stack = new Stack<String>();
1146                stack.push("x");
1147                try{
1148                    while(!stack.empty()){
1149                        lexeme = lexemes.get(i+1);
1150                        if(lexeme.equals("(")){
1151                            stack.push("x");
1152                        } else if(lexeme.equals(")")){
1153                            stack.pop();
1154                        }
1155                        i++;
1156                        termCandidate.add(lexeme);
1157                        if(i >= lexemes.size()) return false; // There can be a case that some closing paranthesis
1158                        // is missed. If we do not check that case, infinite loop occurs. If we reach the end of
1159                        // our lexemes but still have something in our stack, that means some closing paranthesis is
1160                        // missed.
1161                    }
1162                }catch(Exception e){
1163                    return false;
1164                }
1165            }
1166            else break;
1167            i++;
1168        }
1169        // Everything left in lexemes is added to the moreterms candidate.
1170        while(i < lexemes.size()){
1171            lexeme = lexemes.get(i);
1172            moretermsCandidate.add(lexeme);
1173            i++;
1174        }
1175        // If these obey the corresonding rules, return true;
1176        return term(termCandidate, fromChooseCall, postfix) && moreterms(moretermsCandidate, fromChooseCall, postfix);
1177
1178    }
```

We also check `<term>`, `<moreterms>` and the other functions in a similar way.

IR code is updated at `printIrCodeOfExpression(...)` by using a postfix state of a expression. At this point, we utilized Stack to produce IR code.

Problems

We encountered several problems throughout the project. The first problem is to decide the general flow of the code, because we had to check nested choose functions, expressions as conditions

at `while` and `if` blocks, etc. We solved this issue by defining the BNF and using recursion. In this manner, the only thing we had to control was that if an `ArrayList` of `String` describes an expression or not, which was our second and hardest problem.

For the second problem, Burak came up with a clever idea, which makes the process of evaluation from top-most level on BNF (`<expr>`) to the deepest one(`<num>` and `<id>`). While processing the lexemes, we could easily evaluate the postfix and update the IR code.

Another (but relatively much easier) problem for our perspective was fixing the bugs about executing IR code. Each time we encounter a new error on the output of IR code, we had to understand the reason of the error. The errors were generally based on our IR code but sometimes we stuck at some point, where we get an error because of a property of LLVM.

Further Improvements and Conclusion

If we had enough time, we could have changed the methods a little bit, so we could just call a single method that updates IR code automatically, otherwise returns false. Similar to this change, we could design the BNF on Java a little bit different.

On the other hand, to improve project, errors would be printed with a message that explains why there is a syntax exception, which we have already implemented but not used.

In conclusion, we learned how compilers work and convert complex programming languages to low-level codes. Now, we know that compiled languages do this job in order to check each line to detect any errors and create the low-level representation, each time we change and compile the source code. If compilation process has never existed, programmers would have to implement each system using 0’s and 1’s.