CMPE 230 - Project 1 Documentation

Karahan Sarıtaş (2018400174)
Cahid Arda Öz (2019400132)

May 2, 2021

Contents

1 Problem Statement

We are asked to develop a translator called mylang2IR which translates a language called MyLang. Our translator is expected to generate low-level LLVM IR code that will compute and output appropriate statements. We should also prepare a makefile that compiles our source code and generates the executable mylang2ir file.

2 Solution

We splitted the problem to two parts: Pasing the MyLang code and printing the LLVM code. To solve the first part of the problem, we start by reading all the lines from file.my, then we remove the whitespaces and comment blocks.
We send these processed lines to the Parser object. Parser simply parses lines one by one. It then utilizes its Writer object to write the LLVM IR code.
The biggest problem in the project was parsing the expressions. Expressions include

multiplication, addition, subtraction and division operations. In addition, there can also be parenthesis, and choose() functions in an expression. To overcome this obstacle, we eliminate choose() functions by replacing them with the temporary variable ids (%t) to be used in LLVM. We then convert the expression from infix to postfix. This postfix is then used to print corresponding commands to LLVM code.

3 Implementation

Our implementation consists of two objects, namely Parser and Writer, and a main section. We keep our main section as small as possible and put the majority of the burden on Parser object. Parser has its own Writer object as a field. Writer is primarily used at the end of the Parser functions to store what is going to be printed out as a LLVM IR code.

3.1 Parser

Parser has different functions such as validateVariable(), validateNumber(), validateExpression(), syntaxError(), assignmentParser(), expressionParser(), replaceChoose() and read(). Apart from read() function, all functions are private and called inside the read().

1. read(vector¡string¿ lines) = It simply takes the lines of the file.my and examines them one by one. It divides them into categories like if, while blocks, print statements, assignment statements etc. It checks each line for certain syntax errors. It is responsible for the general flow of the code.

2. validateVariable(string var) = It takes a string and checks if it satisfies the conditions to be a variable. If it can be a variable, adds it to the set of variables so that we can print them out at the very beginning of our LLVM code.

3. validateNumber(string var) = It takes a string and checks if it satisfies the conditions to be a number.

4. validateExpression(string expr, vector¡string¿ tokens) = It performs necessary checks on the given expression to find out syntax errors. Along the procedure, it tokenizes them and adds them to a vector. expressionParser() utilizes this token vector. In fact, validateExpression() is only called within the expressionParser().

5. expressionParser(string expr) = At first, by sending the expression to the validateExpression() function, it makes sure that the expression does not have any syntax error. Then using the tokens, prepared by the same function, it turns the whole expression from infix to postfix. Writer takes this postfix expression (vector) and prepares corresponding LLVM code.

6. assignmentParser(string line) = It divides the line into two parts, LHS and RHS. LHS must be a variable and RHS must be either an expression or a variable. assignmentParser() sends them to necessary validate functions and checks if there are any syntax errors. Then it sends RHS to expressionParser() to be processed.

7. replaceChoose(string expr) = This function is excessively used to get rid of choose() functions from the lines. With the help of writer, it stores corresponding LLVM IR code and then replaces choose() with temporary variable ID.

8. syntaxError() = When a syntax error is found, syntaxError() is called. It simply prints the number of the line, where syntax error has occurred.

3.2 Writer

Writer object doesn't have much logic. It has a stringstream object and functions which are used to add lines of code to the stringstream object. Most of the functions are public since they are called by the Parser object. Writer also has counters for keeping track of the last instances of temporary variables and if/while bodies. As we parse lines of the input file, we store corresponding LLVM lines in a stringstream. After printing necessary allocate and store code for variables to the output file, remaining code in the stringstream is printed out.

1. writeToFile(string moduleID, unordered_set¡string¿ variableSet) = Used to create the output file when the parsing of the input file is complete. It starts by declaring global @printf and @choose functions, @print.str and moduleID. Then we start printing the @main section. We allocate variables and set their values to 0. This is followed by the content of the stringstream object. We then end the program by returning zero.

2. writeErrorToFile(string moduleID, int line) = This function is called when the Parser detects a syntax error. It prints the LLVM code that prints the syntax error message.

3. print(string toWrite) = This function is used to add a @printf call to the stringstream object.

4. choose(string saveTo, vector¡string¿ operands) = This function is used to add a @choose call to the stringstream object.

5. writeExpression(vector¡string¿ postFix) = This function takes an expression in post-fix format and then calls functions to create the LLVM code corrosponding to the given post-fix.

6. arithmetic(string saveTo, string opcode, string operandOne, string operandTwo) = This function is used to add a arithmetic operations to the stringstream object.

7. brConditional(string condition, string blockOne, string blockTwo) = This function is used to add conditional branching points to the stringstream object.

8. brUnconditional(string block) = This function is used to add unconditional branching points to the stringstream object.

9. ret(string ret) = This function is used to add return commands to the stringstream object.

10. load(string loadTo, string loadFrom) = This function is used to add load commands to the stringstream object. It is used primarily used before expressions to load variables to temporary registers.

11. allocate(string varName) = Every variable in the program should be allocated in the beginning of the program. This function is used to add allocate commands to the stringstream object.

12. store(string varName, string value) = We need to use the store command to set initial values of every variable to 0 and to store the results of computations in variables. This function is used to add store commands to the stringstream object.

13. mountFile(string filePath) = This function is used open files.

14. dismountFile() = This function is used close files.

15. fileMounted() = This function is used to check whether a file is mounted.

4 Example

In this section, we will show how MyLang programs are converted to LLVM code with a couple examples

4.1 Example 1

```
1 a=4+5
2 print(a)
```

We start by reading every line and removing white spaces. In this example, this doesn't have any effect. We then look at line 0. We see that '-' character is present, which means that line 0 is an assignment statement. We call the assignmentParser and split this statement to two, left hand side (LHS) and right hand side (RHS). We check whether the LHS is an variable using the validateVariable() function. If so, we check whether the RHS is an expression using the validateExpression() function. By validating the RHS by parsing it, we also create the post-fix corrosponding to the RHS. We then use this post-fix to print the computations corresponding to RHS of line 0 by calling writeExpression(). We end the line 0 by storing the result to %_a which represents the variable a. Line 0 is completed and now we start looking at line 1.
In line 1, we see that the print keyword is used. We then look at the expression inside the parentheses of the print statement. In this example, we only have a variable, so a is loaded to a temporary variable and a print call is added to print the new temporary variable.
After parsing all lines, we then start printing the output file using the writeToFile() function. We define global variables and functions. Then print the content of the stringstream object to the main function and end the program.

4.2 Example 2

```
b = 5
while (b+1) {
    print(b)
    b = b - 1
}
```

Line 0 is parsed and processed just like the line 0 in example 1. In line 1, we see that the while keyword is used. To evaluate the expression inside the parentheses, we create a new branch called whcond1. Expression is then validated and converted to post-fix. This post-fix is converted to LLVM code via writeExpression() function and the results is saved to a temporary variable. We add a icmp call to create a one bit boolean value and use it in the conditional branch call. We either exit the while body (regbody1) or enter it (whbody1).
To evaluate the inside of the while body, we process the lines just like we did in example 1 until we see a closing bracket ('}'). We then leave the whbody1 by going back to whcond1.
Program ends when the whcond1 returns false and program jumps to regbody1.

5 Testing

We wanted to create more testcases to test our code as we made changes to it. To do this we first prepared the following BNF code:

```
<program> ::= <chunkList>
<chunkList> ::= <chunk> <chunkList> | <chunk>

<chunk> ::= <regChunk> | <ifChunk> | <whileChunk>
<regChunk> ::= <statementList>
<ifChunk> ::= "if(" <expr> "){\n" <statementList> "}\n"
<whileChunk> ::= "while(" <expr> "){\n" <statementList> "}\n"

<statementList> ::= <statement> <statementList> | <statement>
<statement> ::= (<var> "=" <expr> | "print(" <expr> ")") "\n"

<expr> ::= <term> "+" <expr>  |  <term> "-" <expr> | <term>

<term> ::= <factor> "*" <term> | <factor> "/" <term> | <factor>

<factor> ::= "(" <expr> ")" | <thing>

<thing> ::= <num> | <var> | "choose(" <expr> "," <expr> "," <expr> ","
    <expr> ")"

<num> ::= [1-9]
<var> ::= [A-G]
```

We then used this BNF code to generate test cases. While generating these testcases, we excluded the "/" operator from the language to avoid division by zero errors. Following is an example from our pool of testcases:

```
if(8*(((choose(6*(1)*5-(F)*D*G,2*7+3-D-choose(G*A-G*E,B-choose(2+8,6,(F
    )+3,E),5*F*5+8,8)-5,E,6+8-E)*C+1-D)-A)-D)+A){
     G=(E+(B*C-(E+D*4)*C-G+E*9+9)+D*(9+8))+(F*4*choose(G,4*1+choose(6,G
```

```
    -8*D,E+2,(4-C+A*(3+B-(7*G))+5*A*(B))*3*G),8+G,3+8*E)-choose(4*8*(G-A
    -9+C)*9,B*(3*C+3*D-(E)*(E*G)),C,8+(7*E)*6))-(3*(F*G-1)-7)*B-D+E*C
3    print(((F-choose((1*1*6*9)*9-7-7*(choose(C+4,7-G,G,B+F)*5),A*(1)
    ,(2),(2*(3*6)*5-3+B-5-4*4*E)*choose(D*(6-G*(3-B*1-E))+B-A+G,(2)
    *6-7*5*(choose(G+D+E,5-6,D-4*A,B))*A*1*F-D+C*D*(6-7),6*C,(C+B+E*C)))
    -4+C-5)-3)-(E*4*4)*choose(choose((((C)))*8,(B)-C-1-E,D,E-3+5+G
    *5-2*2)+C+(((6*F-G*8-F)*3*E)),(8*G+A+E*(A*E+3))-((B*G*5-C*(6+4)*(F)+
    A)),((F+F)+1*A)+(F)+1*(9-E),1+G-8*(B)*choose(8,choose(G*7,(9-F*2-5)
    -4,5+7-1*4+F,8)+A-D*A,A-F-8*F+9,F)*(7-2))*8)
4    B=B*1
5    print(6*(4*A)*choose(9-5-C,(F+C),F*G+4-1,D+E-(5)*choose((C+3+choose
    (A+A,(3-6*4)+B,9-choose(1,3+1,C+A,choose(7*8,(B*D*2),B,2*(5*B+C))
    *9+7+6)-(A+8-4),7)*E-2)*C+A*choose(4,2,B*G-2-B,(F-3-D+C)+6-G)+1,B*C+
    E-B+B,(9-(choose(4-E,4,C+F*G*E-A,3-G*6*G)*E))+6+(5+4)*D-D,7*(B)-F)
    +4+G)-D+B*E+(B*2*2)*F)
6    G=(6*(5-F-D*7)-(3*C)*(E)*7)
7    print(choose(choose(A,(A-C*2*F+(D-5+A)*G)*5-8*9,((7))-3*((C*F)-3-5)
    +D*G,((A*(D+9))*choose((6)*((8)*B-7+(D*C+8+1)*4),E-((7*D-1*4-B)+(C*D
    +B))*C-G+A-4,((5-C-6*G+5)*E)*1+8,((B-4*B)*(3-8)*G+(C*F+9+4))*(1))*D
    +2*E)),(3*(((6-A-E*G)*C)*(5*1)-1-8-D*(3*C)))+(C)+((B)*2+B),(7-E-(E-B
    )+(2)-1)*B*((D-2-4-7*D*G))+((7+7)-F*G)*G+(choose(2,G,F,5*8)),2))
8  }
9  C = G-(B*9-6-D*choose(D,G+B,B,A)*B*G+F*3-9*G)-D
10 while(C){
11     C = C -1
12     print((((2-(C*6*C+C)))+A*C-A*4*(9*5*3))+6*3)
13     A=choose(5*((choose(B*(G+5+3)+E,5+9-C,choose(D-A-G,A+1+B,4-2,((
    choose((E+2*B*C+D+2)*F,7*8-E*G*C*(A+C-B-A),2,G*1+8*A*G+(E*5)+C-2)*4-
    F+F)))*B+6*9,6*E+(F)*F-1)+(F-7)-(3-1+8+(1))*7*1)-B+4),(2*(1*choose((
    choose(5+(A+D)+A*6+B-F,G*6,F-C,(C)))-(C*2-5)-D+(7-C),((A*choose
    (2+(4-5+9-7)+A*E,B+2-E+6,4,7*choose(5,A+2-D*(8)-F,8+F,G*C*D-2+2)
    -3+5)-D+1))*G-(D*7+2-9-G),7*E*(F)+(4*7-D-F)-1-F,A-(1+3*C)*(F*7*9*A
    -5*E+A)*G+G+E)*2+choose(2,B*1-8,B-4,B+choose(B+E+1*A,(C+E*6)-B*2+G
    +8-(B),2*F*2,3))+(G+E-E)-B*B*4-A)-G)*(E-D*A*2)*(((((2-8-2+9-8)+9*2*
    choose((F*9)-A,C,E,6))+(F-B)+3-(G+7)*7)))+B*8-E,E+((7*7-3)+B)*((3-D)
    +E)*choose(((7*6*B-E-A)-5)-(7+8+2),F*(5*E+B-6-C*4)*C,5+choose((4+F),
    B*1+B,5*8-1*G-choose(9,F,8-A,9)*(1)*1,2+B+6)+2,2+(C+8)+3*E*D+D+G)-D
    ,((4*B+choose(9,2+B+5,A+(6+4)*1,9-G*B-choose(E*3*D*8*(F-6),G-G*A*7,C
    ,2-C)))*1+1*(7)*1))*6
14     print(choose(8+A,(5)*(5)*(6+8-3*1),6*((F+6*9*choose(F,((5-B+A-B+5-G
    )),(9-B+G*2+5+5),5-B*9)))*C*B-C*G*choose((9*(8))+F,(B*(6-F+D+7*(4*9+
    B))-7+D-(A))*9+D,B*(A-choose(4,(D)-5,(5)-C-8,(6+6-8+G+8*4*7-C+3)+7))
    -2*(2*B-(5)-B*B+D*6),(B+G)-6+8)+D+9,(4*B-1*(choose(G+C,C,1+D-choose
    (8*D,3-(C)-7*F,C*choose(E,4-(2*1)+E+C,(8*7+C)*A+B,(F)-A-2)*A*B,(C-A)
    +G)*9,C+F))-C)-B)*(6*1*E-4*4)-((((F)))*D)-1)
15     A=G*choose(9-1-G-(E)*1*choose(3,E*G-B+E,A*(8),F-2),G*5*(B-(9+(2*6-C
    *(F)+8))+(F)-choose(B+D+(D+8+E)*E,9,C,E)-D*B)+8*1,(((D+A*B)*(4-(1)*C
    )+G-E-9)*E-7*C-2+9*G-A*7)*(D+4*5+(3+E)*1*B*3)-6*choose((6*D),G-E
    -2*1,choose(A*5*9+7,E-(A+D)*(B-1-4-5)-D,D+8,B+(A)-6-D)-B*A,(5))*A
    *1+5-(6)+choose(G,D,(3)+G*A*(4),F+G+D*8),E*(G*3)-4+8*A)
16     print(((F*((5-4+B*F)+E))*F)+choose(2-choose(((3-D)),(D),choose((E
    *7)+F,A*(6-D),(G*B),3),7)*4,7*(choose((7-D+A)*choose((B+7+A*2+E+3*8)
    +C,9*((8)+(5-4-E)+D+A)*(6),C*(7+1-choose(B*3,8+4,3,6-B*D+9)*F-C)+(D)
    +F*(A*G)+G,1*(6)),9+2*(F)*A,E-choose(B-F+A+5+6,choose(8*4,F+choose(A
    *C+6,4-4,4,6-B)-7-E,A,4*A)*A*6+D*4-(2*E+8),choose(E+9*7+4,(D*D),(E)
```

```
    *(D*C+B+5*E*5-E),G),B)*(6*E)*C,8)*4),(choose(A-B*4+6,3,5-(E)*(7)+2,A
    )*(1+1+2))*((B+(1*C+choose(C,E*E,5-1,6)))-(1-8-8)*8*(C-B*D)),F))
}
```

6 Conclusion

Looking at the testcases provided to us and created by us, we conclude that our program solves the given problem. There are some improvements that come to mind:

1. We could have used a macro for determining whether any debug information gets printed to console instead of commenting out debugging lines.
2. We could have used a BNF oriented approach to parsing the MyLang codes.