Artun Akdoğan
2020400327

Zeynep Gültuğ Aydemir
2019502276

CMPE 230 - SYSTEMS PROGRAMMING
Spring 2021
Project 1

For the project, we developed a translator for a language called MyLang, using LLVM. Our translator will produce the low-level intermediate representation(IR) code of the input files with “my” extension. The project can be divided into four main points: Lexer, Abstract Syntax Tree, Parser, and Code Generator.

Lexer

We started with implementing the lexer to tokenize inputs first. For each type of input, we prepared specific groups of tokens as an enum structure.

The function `int lex_tok (const char *text)` is designed to take file.my file as a parameter, divide the text into characters, read them one by one and group them into tokens until it reaches the end of the file. It checks out the numbers, variable names, comment lines, whitespaces, braces, operators and specific keywords of while, if, choose and print. Eventually, it returns their own specific tokens assigned at `enum block_type`. Sample blocks of codes are as follows:

**Assigned tokens (main.cpp, line 18)*

```
// Used for assigning tokens for specific inputs
enum block_type {
    num = -1,
    var = -2,
    _while = -3,
    _if = -4,
    _choose = -5,
    _print = -6,
    asg = -7,
    eof = -8
};
```

**lex_tok function checks whether the characters are variables or not. If so, it stores the name in `lex_str`. (main.cpp, line 78)*

```
    // Checks the non-numeric characters
    if (isalpha(text[pos])) {
        // Prepare variable name in lex_str if the consecutive
characters are alphanumeric
        do {
            lex_str += text[pos++];
        } while (isalnum(text[pos]));
```

Artun Akdoğan
2020400327

Zeynep Gültuğ Aydemir
2019502276

Abstract Syntax Tree

Then we implemented the Abstract Syntax Tree nodes in order to design the parser. The base class for all types of expressions is the Expr class. Other node classes are extends from the Expr class as one can be seen below:

** Number node class, expending the base class of Expr (expr.h, line 28)*

```
// NumExpr - Expression class for numeric values
class NumExpr : public Expr {
   public:
    NumExpr(string value) : Expr(value) {}
    virtual string codeGen() { return string(); }
};
```

There are 5 different nodes which are for numbers, choose function, variables, assignments, operators. They are all designed to generate the required IR code later on.

**AsgExpr node class, extending codeGen function. It returns the LLVM script of its assignment operation as string. (expr.cpp, line 58)*

```
string AsgExpr::codeGen() {
    return assignee->codeGen() + "store i32 " + assignee->tempNameGet()
+
          ", i32* " + varNameGet() + '\n';}
```

Parser

The parser is implemented in main.cpp. Basically, it runs until the end of the input file and calls the specific node classes according to the tokens it encounters, so that the program can generate the IR code according to their AST nodes.

**Sample switch cases from the parser function. According to the token of the character, its AST node is called. (main.cpp, line 159)*

```
        // If encountered with a variable, calls its expression
function.
       case var:
           top = new VarExpr(cur_str);
           break;
       // If encountered with a numeric expression, calls its
expression function.
       case num:
           top = new NumExpr(cur_str);
           break;
```

The function parseExpr returns the AST nodes. The function is going to be called later in the code part to generate the LLVM script according to the input.

** parseExpr is called to receive the nodes and create the required IR code.*

```
       // Reads through the token until it reaches the end of the file
```

```
        while (tok != eof) {
            switch (tok) {
                case asg:
                    exprPtr = new AsgExpr(cur_str,
parseExpr(text.c_str()));
                    gen.add_code(exprPtr->codeGen());
                    delete exprPtr;
                    break;
```

Code Generator

Then we handled the LLVM representation part in the generator.cpp file. We listed the required strings according to the IR structure. In order to generate a solid LLVM script, the code first detects the node and calls its own codeGen function, adding it to the final LLVM script. Then finally, it creates the file with ".ll" extension and writes the overall IR code to that file with `io.writeFile(gen.get_code());` line(main.cpp, line 423). Sample blocks from the code are as below:

**Switch case of assignment operations, calling its node's codeGen function and adding it to the IR code.*

```
            case asg:
                exprPtr = new AsgExpr(cur_str,
parseExpr(text.c_str()));
                gen.add_code(exprPtr->codeGen());
                delete exprPtr;
                break;
```

**add_code function, adding the node's specific code block to the LLVM script.*

```
void Generator::add_code(string text) { this->gen_code += text; }
```

**get_code function, concatenating the code blocks together and generating the final LLVM script.*

```
string Generator::get_code() {
    // Safe return the result.
    return this->beg_code + this->choose_func + this->st_code +
           this->init_args + zero_args + this->gen_code +
this->end_code;
}
```

**beg_code string, first part of the code containing the module name and the prototype for printf output statement.*

```
    const string beg_code =
        "; ModuleID = 'mylang2ir'\n"
        "declare i32 @printf(i8*, ...)\n"
        "@print.str = constant [4 x i8] c\"%d\\0A\\00\"\n\n";
```

Artun Akdoğan
2020400327

Zeynep Gültuğ Aydemir
2019502276

Running and Testing

The program can be run in LLVM version 3.3, with “mylang2ir” command plus a file with “.my” extension. When the filename is missing or file extension is not “.my”, the user will be prompted to “Please enter a filename!” or “Invalid File Extension” message and quit the process.

Description about MyLang language:

1. Variables are always integer variables and their default value will be 0, if they are not initialized.
2. Executable statements consist of one-line statements, while-loop, and if compound statements. No nested while-loops or if statements are allowed in the language.
3. One-line statements are either assignment statements or print statements that print the value of an expression.
4. There is a function “choose(expr1,expr2,expr3,expr4)” which:
   - ➔ returns expr2 if expr1 is equal to 0,
   - ➔ returns expr3 if expr1 is positive
   - ➔ returns expr4 if expr1 is negative.
5. The language supports only binary operand operations, which are multiplication, division, addition, and subtraction. (*, /, +, -) Unary minus operation is not supported but parentheses are allowed. Operator precedence needs to be implemented as in other programming languages, such as C or Java.
6. On a line, everything after the # sign is considered as comments.
7. If statement has the following format:

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

   If expr has a nonzero value, it means true. If expr has zero value, it means false. “Else” and nested if statements are not supported.
8. While loop will have the following format:

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

   If expr has a nonzero value, it means true. If expr has zero value, it means false. Nested while statements are not supported.There are no nested while statements.
9. print(id) statement, prints the value of variable id.
10. In case of a syntax error in line x, program produces the LLVM script, printing “Line x: syntax error”

Regarding these syntax rules and our implementation of the MyLangIR translator, the program runs and passes all the given test cases successfully.