# CMPE230 Project 1: mylang2IR

Burak Ferit Aktan: 2019400183
Ilgaz Er: 2019400018

## General Approach

The first project for CMPE230 was the creation of a compiler from a language called “mylang” to LLVM IR, or simply LLVM, an intermediate representation form that can be compiled for different languages or interpreted. “mylang” has a single type that corresponds to int, while and if statements as flow control and an assign operator. Expressions consist of the four arithmetic operators (+, -, *, /), parantheses and a choose function that is similar in functionality to C and Java’s ternary operator. Variables are automatically declared to zero and expressions have no side effects.

For the compiler design, two methods were considered: a procedural programming approach where the source code is directly transformed into LLVM, and an object oriented approach where a parse tree is generated from the source code and the LLVM code is generated from the parse tree. The object oriented approach was decided on due to factors such as ease of writing and debugging, benefits of modularity in teamwork and the expected educational value of designing an object oriented program.

Our approach is based on three principles:

1. Separation of parsing and LLVM generation
2. Separation of both parsing and LLVM generation logic into separate classes for different mylang structures
3. Keeping each node unaware about the properties of its children, parents and siblings.

## Parse Tree Generation

The parsing logic is loosely based on the following BNF representation of mylang:

<var> → [a-zA-Z][a-zA-Z0-9]*
<num> → [0-9]+

<choose-term> → choose(<stmt>, <stmt>, <stmt>, <stmt>)
<term> → <num> | <num> ( * | / ) <term> | (<term>) | (<stmt>) | <choose-term>
<expr> → <term> | <term> ( + | - ) <expr>

<assign-stmt> → <var> = <expr>
<print-stmt> → print(<expr>)
<if-stmt> → if(<expr>){<stmt-list>}
<while-stmt> → while(<expr>){<stmt-list>}

<stmt> → <assign-stmt> | <print-stmt> | <if-stmt> | <while-stmt>

<stmt-list> → <stmt> | <stmt> \n <stmt-list>
The actual code is structured around three base classes: `Statement`, `Expression` and `Value`. `Value` extends `Expression` as a solitary value serves as an expression by itself. These base classes are responsible for parsing a sentence or phrase and returning the appropriate subclass.

Each subclass of `Statement` and `Value` exposes its regex pattern in a constant called `PATTERN`. The `Statement` and `Value` classes use this for checking if a given expression belongs to that particular subclass. Delegating the definition of patterns to each subclass increases readability and allows for easily adding in more types of statements. Below, you may find the pattern for `IfStatement`:

```
public static final Pattern PATTERN =
Pattern.compile("\\h*print\\h*\\((.+)\\)\\h*");
```

As nesting flow control statements is explicitly disallowed in the specification, a boolean `canHaveFlow` is used to transmit the information necessary for checking the presence of nested if and when statements. The restriction provided by the specification requires the breaching the principle of no information. It is important to note that upon the removal of this feature, the compiler would be able to compile nested control flow statements.

The parsing logic in `Expression` is implemented in a while loop that iterates over the sentence character by character, detects values, constants, special keywords, and operators and then transforms these tokens into a tree of expressions using a stack and similar logic to an infix to postfix converter. The four arithmetic operations, their corresponding operators and their order of operations are defined in the Enum `Operation`, which allows for the addition of other arithmetic operations with ease. The only expression keyword “choose” is hard coded in the while loop, which decreases the complexity of the parser at the expense of extendability.

## LLVM IR Generation

Converting our parse tree into LLVM code is done by `getLLVM` methods in all nodes in the parse tree. These `getLLVM` methods call the `getLLVM` methods of their children, combines their results and appends to that the LLVM code belonging to the current node. The result of expressions are assigned to a temporary LLVM variable at the end of the LLVM code of this particular node and this temporary variable is made available to the parent node with the `getResult` method.

The easiest nodes to convert to LLVM code were the `Number` and `Variable` expressions, which both extend the `Value` class. They are the leaf nodes of our parse tree, so they do not make any `getLLVM` calls.

**Finding LLVM Code of a Number**

LLVM code of a Number instance is nothing, the integer constant is just passed on in `getResult`.

**Finding LLVM Code of a Variable**

LLVM code of a Variable instance is just loading that pointer (mylang variables are kept as pointers in LLVM code) to a temporary variable. This temporary variable is passed on in `getResult`, just as any other expression.

**Finding LLVM Code of a Choose Expression**

The LLVM code of a choose expression contains 2 conditions to be checked and 3 cases whose results may be returned according the this conditions.

Case 1 is the expression whose result should be returned if Expression equals zero.

Case 2 is the expression whose result should be returned if Expression is greater than 0.

Case 3 is the expression whose result should be returned if Expression is less than 0.

Condition 1 is checks whether expression 1 equals to zero or not. If it is true, the code branches to Case 1. If it is false, the code branches to condition 2, which checks whether Expression 1 is greater than 0 or not. If it is greater the code branches to Case 2 and if not, it branches to Case 3. Note that the expressions in Case 1, 2 and 3 are evaluated lazily, i.e. they are only calculated if they are the expressions that will be returned.

All these cases assign their results to a pointer variable. At the end of all three cases is a jump to the end label of choose, where the result is loaded to a temporary variable.

**Finding LLVM Code of an Arithmetic Expression**

Every arithmetic expression has a right term, operator and a left term.

First, the LLVM code of the right and left terms are generated. Then, by using the LLVM instruction for the specific operation, (as defined in `enum Operation`) the operation is performed. The result is assigned to the result temporary variable of this Arithmetic Expression.

As an example:

Suppose that the result variable of left term is `%t1`, the result variable of right term is `%t2` and the operation is `ADD`. The LLVM instruction corresponding to `ADD` is `add`. Let the result variable of this arithmetic expression be `%t3`.

```
%t3 = add i32 %t1, %t2
```

**Finding LLVM Code of an If Statement**

LLVM code of an If Statement consists of 2 parts and there is an end label at the end.

First part checks whether condition is equal to zero or not. If it is zero, the code jumps to the end label and if not, it jumps to the LLVM code of the list of statements.

Second part is LLVM code of the Statement List to be executed if condition isn't zero. Branches to end.

**Finding LLVM Code of a While Statement**

Logic is nearly same with if statement.

The LLVM code of a While Statement consists of 2 parts and there is an end label at the end.

First part checks whether condition is equal to zero or not. If it is zero, the code jumps to the end label and if not, it jumps to the LLVM code of the list of statements.

Second part is LLVM code of the Statement List to be executed if condition isn't zero. At the end, it jumps back to the condition.

**Finding LLVM Code of a Print Statement**

This was directly taken from the project description.

**Finding LLVM Code of an Assign Statement**

At left side of an assign statement there is a variable and right side there is an Expression. We first find LLVM code of that Expression.

Then load the result temporary variable of that expression to pointer variable (left hand side)

**Finding LLVM Code of a Statement List**

Statement List consists of Statements. The LLVM code corresponding to each statement in the statement list is generated and then combined sequentially.

**Finding LLVM Code of the Program**

Beginning of the LLVM code was provided in the project description.(define i32 @main ...) Afterwards, all the variables that are accessed throughout the mylang code are allocated and initialized to 0. Following that, the LLVM code of the Statement List of the Program is appended. The end of the LLVM code was provided in the project description.(ret i32 0 …)

**What If Parse Tree Couldn't Generated**

In case of syntax errors, an LLVM program that prints: "Line x: Syntax error" is written to the output file. This string is printed character by character using LLVM printf calls.

## Analysis of Approach

As much of our approach is shaped by our three principles, many of the benefits and drawbacks of our code are a direct consequence of them.

The separation of parsing and LLVM generation logic was a clear advantage in terms of teamwork, as it made for a clean separation of tasks between different team members and allowed each team member to dedicate the majority of his efforts towards his area.

The clearly defined scope of each component along with the tree structure and the opaqueness of each node allowed for less edge cases, which meant less bugs from the beginning. It also made the process of finding bugs in  both the parser and LLVM generator

much easier. It is our belief that the ease of debugging has more than made up for the chief drawback of our approach, increased investment of time in the more structured object-oriented code.

Another drawback of our approach is the lack of distinction between syntax and semantics. Each phrase is parsed and transformed from a string immediately into an object that reflects its semantics, which made for some edge cases that were difficult to handle. It also makes implementing different types and especially operator overloading much harder. This was a deliberate choice however, as the specification for mylang includes a single type and the gains in edge case handling would be dominated by the increased time investment and complexity that a separating syntax and semantics would bring.

## Conclusion

Our code is modular, easy to debug, and Object oriented, and most importantly it passes all of the test cases. However, there are some things that could be improved if we had more time:

While converting the parse tree to LLVM code, each node calls the `getLLVM` method of its children and these methods return large Strings. A lot of large Strings are produced and copied, increases time complexity. As the strings returned by child nodes are not modified by parent nodes but only combined with the parent's LLVM code, each node could write its LLVM code directly to the file, instead of passing strings of ever-increasing length to its parent.

Also, the syntax error was printed character by character. With more time for gaining a deeper understanding of the print function, this could be replaced with code that prints the whole error string at once.

To conclude, this project was a valuable experience for us and an example of successful teamwork achieved through the separation of duties.