# Compiler for PASCALMAISPRESQUE

## Project Part 3 : LLVM Code Generation Report

Dave Pikop Pokam –> 506979
Mohammad SECUNDAR –> 504107

INFO-F403 - Introduction to language theory and compiling
MA1-SECUC
Université Libre de Bruxelles

## 1 Introduction

This report outlines the implementation and functionality of a compiler's code generation phase for a PASCALmaispresque-like language, focusing on the generation of LLVM Intermediate Representation (IR) from source code parsed into a parse tree.

## 2 Implementation

### 2.1 ParseTree Class

The ParseTree class represents the abstract syntax tree (AST) of the parsed program, with methods for converting the tree to LaTeX and TikZ representations, aiding in visualization and debugging.

Each node of the ParseTree is defined as an object of the class Symbol and contains either a list of children or not.

This class was implemented in the part 2 of the project.

## 3 Main Class

The Main class handles file reading, parsing, LLVM code generation, and saving the generated code to a file. It encapsulates the entire process from reading the source file to saving the generated LLVM IR code.

### 3.1 LLVM Code Generation

#### 3.1.1 Methods for begin...end generation

The ParseTree class is enhanced with private fields and methods to facilitate LLVM IR generation, focusing on crucial aspects like arithmetic and conditional expressions, as well as control structures and I/O operations. These methods closely mirror those in the Parser, each representing a non-terminal function in the language.

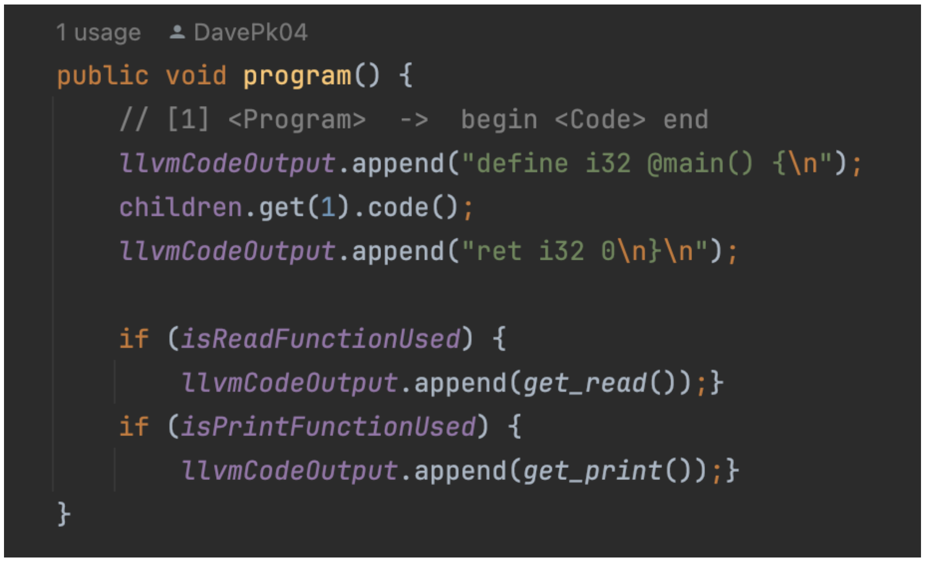
When a PascalMaisPresque program is input, LLVM code generation is initiated via the **program()** method, adhering to the following rule :

[1] <Program> -> begin <Code> end

This structure dictates that the Program node comprises three children : `begin`, `<Code>`, and `end`. The **program()** method systematically processes the parse tree, starting from the top, to generate the corresponding LLVM IR code.

## Illustration

The LLVM code generation process is visually represented below. All programs starting with begin and ending with end are transformed into an LLVM IR code structure beginning with **define i32 @main() {** and concluding with **ret i32 0 }**.



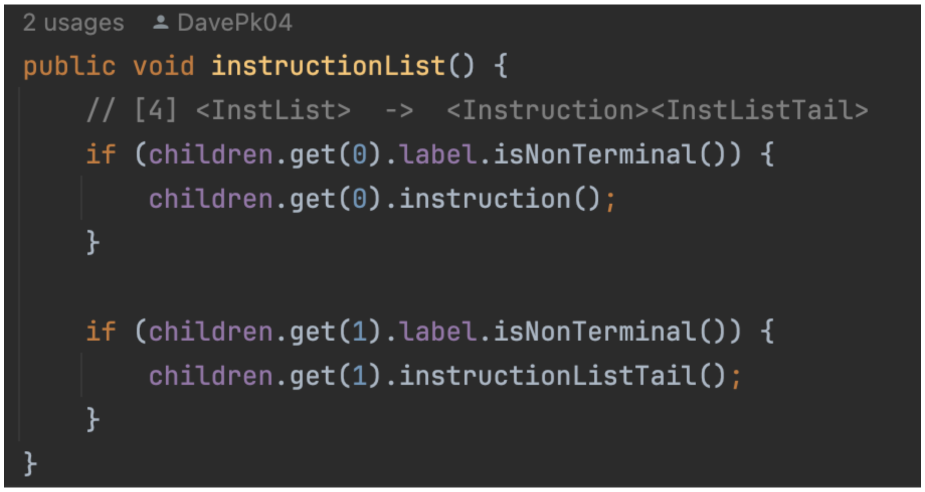
```
1 usage   DavePk04
public void program() {
    // [1] <Program>  ->  begin <Code> end
    llvmCodeOutput.append("define i32 @main() {\n");
    children.get(1).code();
    llvmCodeOutput.append("ret i32 0\n}\n");

    if (isReadFunctionUsed) {
        llvmCodeOutput.append(get_read());}
    if (isPrintFunctionUsed) {
        llvmCodeOutput.append(get_print());}
}
```

Figure 1 – LLVM code generator for Program

The non-terminal Code is invoked next. It's involved in two rules, but essentially, Code leads to InstList, and the **EPSILON** terminal is disregarded.

[2] <Code> -> <InstList>
[3] <Code> -> EPSILON

Therefore, the function `instructionList` is called.



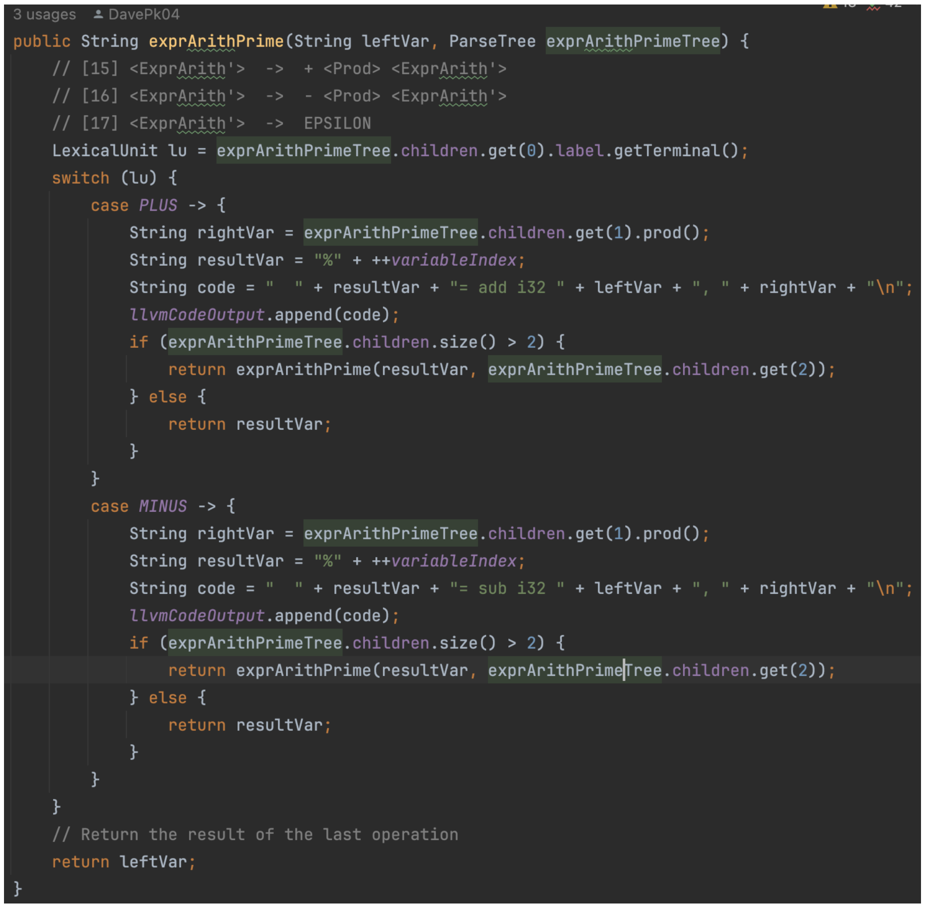
```
public void instructionList() {
    // [4] <InstList>  ->  <Instruction><InstListTail>
    if (children.get(0).label.isNonTerminal()) {
        children.get(0).instruction();
    }

    if (children.get(1).label.isNonTerminal()) {
        children.get(1).instructionListTail();
    }
}
```

Figure 2 – LLVM code generator for Instlist

### 3.1.2 Methods for arithmetic operations

The exprArithPrime method in the ParseTree class handles arithmetic operations, specifically addition and subtraction. It receives a left variable **leftVar** and a subtree **exprArithPrimeTree**. Based on the lexical unit (either PLUS or MINUS), it calculates the result of adding or subtracting the right variable **rightVar** from leftVar. This result is stored in resultVar, and the LLVM IR code for the operation is appended to llvmCodeOutput. If the subtree has more children, the method recursively processes them ; otherwise, it returns the result. This approach allows for the evaluation of complex arithmetic expressions in the input program.

## Illustration



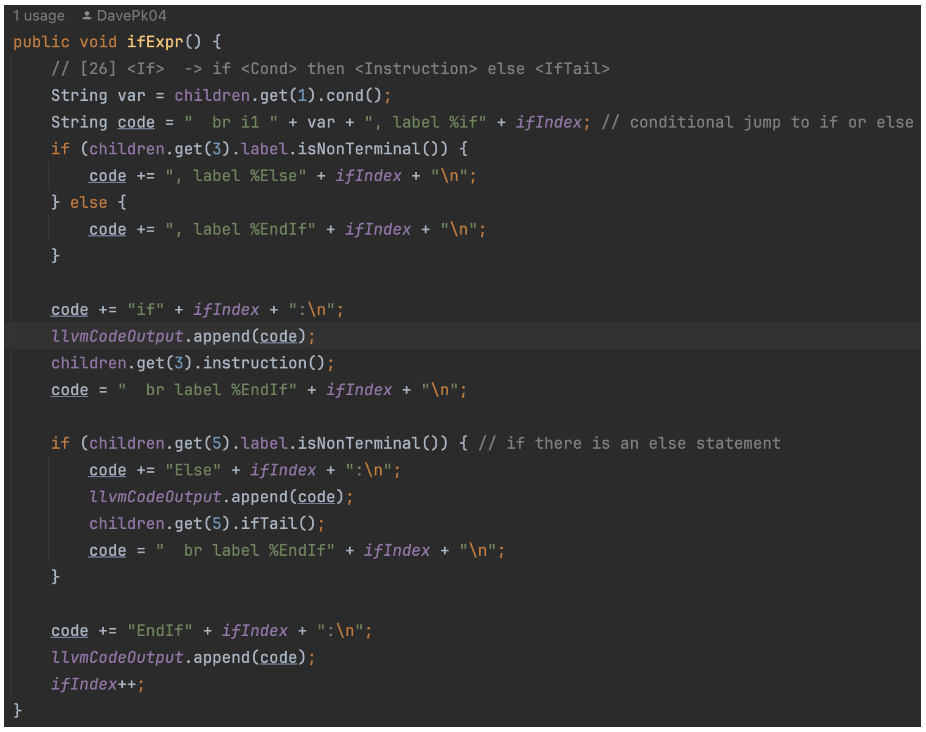
```java
public String exprArithPrime(String leftVar, ParseTree exprArithPrimeTree) {
    // [15] <ExprArith'>  ->  + <Prod> <ExprArith'>
    // [16] <ExprArith'>  ->  - <Prod> <ExprArith'>
    // [17] <ExprArith'>  ->  EPSILON
    LexicalUnit lu = exprArithPrimeTree.children.get(0).label.getTerminal();
    switch (lu) {
        case PLUS -> {
            String rightVar = exprArithPrimeTree.children.get(1).prod();
            String resultVar = "%" + ++variableIndex;
            String code = "  " + resultVar + "= add i32 " + leftVar + ", " + rightVar + "\n";
            llvmCodeOutput.append(code);
            if (exprArithPrimeTree.children.size() > 2) {
                return exprArithPrime(resultVar, exprArithPrimeTree.children.get(2));
            } else {
                return resultVar;
            }
        }
        case MINUS -> {
            String rightVar = exprArithPrimeTree.children.get(1).prod();
            String resultVar = "%" + ++variableIndex;
            String code = "  " + resultVar + "= sub i32 " + leftVar + ", " + rightVar + "\n";
            llvmCodeOutput.append(code);
            if (exprArithPrimeTree.children.size() > 2) {
                return exprArithPrime(resultVar, exprArithPrimeTree.children.get(2));
            } else {
                return resultVar;
            }
        }
    }
    // Return the result of the last operation
    return leftVar;
}
```

FIGURE 3 – LLVM code generator for ExprArithPrime

### 3.1.3 Methods for conditional statements

The **ifExpr** method in the ParseTree class handles the `if-else` control structure. It starts by evaluating the condition `<Cond>` and generating LLVM IR code for a conditional jump, determining whether to execute the `if` or `else` block. The ifIndex is used to label the branches uniquely. The method then processes the `<Instruction>` child for the `if` part and, if present, the `<IfTail>` child for the `else` part. Each branch ends with a jump to a common end label `EndIf`. This implementation allows for the execution of either the 'if' or 'else' block based on the condition, adhering to the structure defined by the input program.

## Illustration



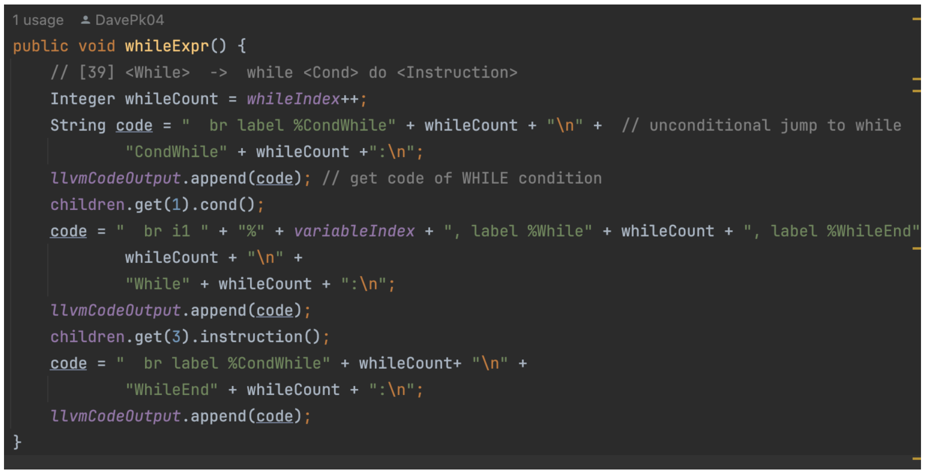
```
1 usage   DavePk04
public void ifExpr() {
    // [26] <If>  -> if <Cond> then <Instruction> else <IfTail>
    String var = children.get(1).cond();
    String code = "  br i1 " + var + ", label %if" + ifIndex; // conditional jump to if or else
    if (children.get(3).label.isNonTerminal()) {
        code += ", label %Else" + ifIndex + "\n";
    } else {
        code += ", label %EndIf" + ifIndex + "\n";
    }

    code += "if" + ifIndex + ":\n";
    llvmCodeOutput.append(code);
    children.get(3).instruction();
    code = "  br label %EndIf" + ifIndex + "\n";

    if (children.get(5).label.isNonTerminal()) { // if there is an else statement
        code += "Else" + ifIndex + ":\n";
        llvmCodeOutput.append(code);
        children.get(5).ifTail();
        code = "  br label %EndIf" + ifIndex + "\n";
    }

    code += "EndIf" + ifIndex + ":\n";
    llvmCodeOutput.append(code);
    ifIndex++;
}
```

FIGURE 4 – LLVM code generator for ifExpr

### 3.1.4 Methods for Loop Control Structure

The `whileExpr` method in the `ParseTree` class handles the 'while' loop structure in LLVM IR generation. It begins with an unconditional jump to **%CondWhile*N***, incrementing `whileIndex` for uniqueness. The condition (`<Cond>`) code is appended next. If true, it proceeds to **%While*N*** to execute the `<Instruction>`, then loops back for condition re-evaluation. Otherwise, it jumps to **%WhileEnd*N***, concluding the loop. This process enables executing code repeatedly while a specified condition remains true.

### Illustration



```
1 usage  DavePk04
public void whileExpr() {
    // [39] <While>  ->  while <Cond> do <Instruction>
    Integer whileCount = whileIndex++;
    String code = "  br label %CondWhile" + whileCount + "\n" +  // unconditional jump to while
            "CondWhile" + whileCount +":\n";
    llvmCodeOutput.append(code); // get code of WHILE condition
    children.get(1).cond();
    code = "  br i1 " + "%" + variableIndex + ", label %While" + whileCount + ", label %WhileEnd"
            whileCount + "\n" +
            "While" + whileCount + ":\n";
    llvmCodeOutput.append(code);
    children.get(3).instruction();
    code = "  br label %CondWhile" + whileCount+ "\n" +
            "WhileEnd" + whileCount + ":\n";
    llvmCodeOutput.append(code);
}
```

FIGURE 5 – LLVM code generator for whileExpr

## 4 Tests

The testing phase was crucial in validating the compiler's functionality. Tests covered a wide range of programming constructs, including assignments, conditional statements, loops, arithmetic operations, and input/output functionalities.

Upon generating the LLVM IR code, it is writing in the file with extension **.ll** and stored in **more/results** folder.
Then, we can run this file by using this command :

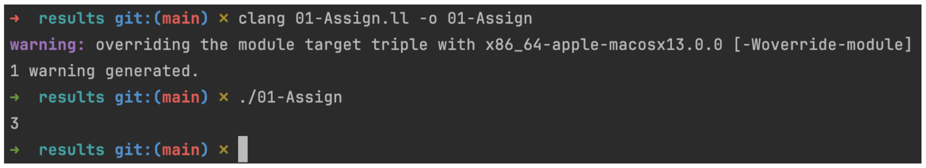
```
clang <input_file_name>.ll -o <output_file_name>
```

### 4.1 Test for Assign.pmp

The input program is :

```
begin
  x := 1+2...
  print(x)
end
```

### Result



```
➜  results git:(main) ✗ clang 01-Assign.ll -o 01-Assign
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./01-Assign
3
➜  results git:(main) ✗
```

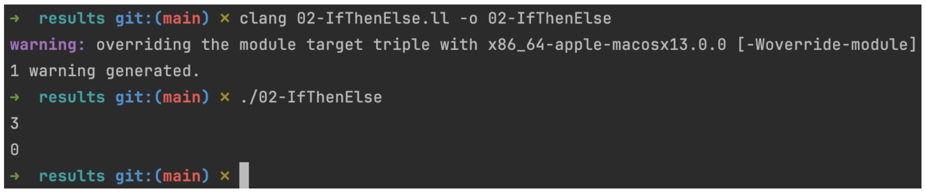
FIGURE 6 – Assign.pmp running

## 4.2 Test for IfThenElse.pmp

The input program is :

```
begin
  read(x)...
  if 1 = 1 then
    x := 0
  else
    x := 1...
  print(x)
end
```

### Result



```
➜  results git:(main) ✗ clang 02-IfThenElse.ll -o 02-IfThenElse
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./02-IfThenElse
3
0
➜  results git:(main) ✗
```

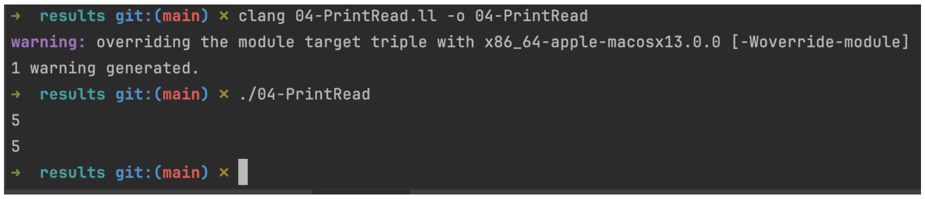
FIGURE 7 – IfThenElse.pmp running

## 4.3 Test for PrintRead.pmp

The input program is :

```
begin
  read(a) ...
  print(a)
end
```

## Result



```
➜  results git:(main) ✗ clang 04-PrintRead.ll -o 04-PrintRead
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./04-PrintRead
5
5
➜  results git:(main) ✗
```

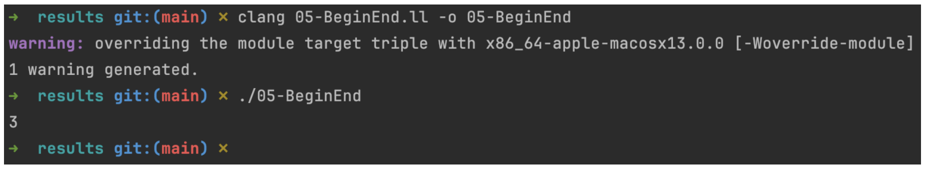
FIGURE 8 – PrintRead.pmp running

### 4.4 Test for BeginEnd.pmp

The input program is :

```
begin
  begin
    z := 3...
    print(z)
  end
end
```

## Result



```
➜  results git:(main) ✗ clang 05-BeginEnd.ll -o 05-BeginEnd
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./05-BeginEnd
3
➜  results git:(main) ✗
```

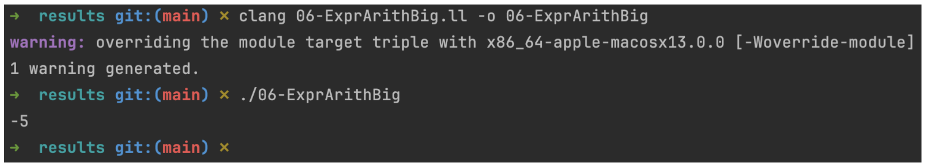
FIGURE 9 – PrintRead.pmp running

### 4.5 Test for ExprArithBig.pmp

The input program is :

```
begin
  x := 0...
  x := ((x+3)*-2)/4...
  print(x)
end
```

### Result



```
➜  results git:(main) ✗ clang 06-ExprArithBig.ll -o 06-ExprArithBig
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./06-ExprArithBig
-5
➜  results git:(main) ✗
```

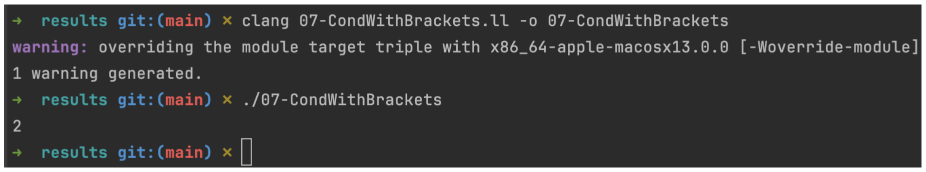
FIGURE 10 – ExprArithBig.pmp running

## 4.6 Test for CondWithBrackets.pmp

The input program is :

```
begin
  x := 1...
  y := 1...
  z := 4...
  while {x = 1 or {y < 2}} and 3 < z do
    z := 2...
  print(z)
end
```

### Result



```
➜  results git:(main) ✗ clang 07-CondWithBrackets.ll -o 07-CondWithBrackets
warning: overriding the module target triple with x86_64-apple-macosx13.0.0 [-Woverride-module]
1 warning generated.
➜  results git:(main) ✗ ./07-CondWithBrackets
2
➜  results git:(main) ✗
```

FIGURE 11 – CondWithBrackets.pmp running

## 5 Conclusion

The project demonstrates a comprehensive understanding of compiler design in the context of LLVM IR code generation, translating high-level language features into LLVM IR effectively.