
Project part 3: LLVM

Subject :
introduction to
language theory
and compiling

TRANGANIDAS Orestis
BOUGLAM Sara

Contents

1 Introduction.....	3
2 Background.....	4
2.1 AST.....	4
2.2 LLVM IR.....	4
3 Implementation.....	6
3.1 Overview.....	6
3.2 Phase 1.....	6
3.3 Phase 2.....	7
3.3.1 Bonus.....	8
3.3.2 Main class.....	8
4 Conclusion.....	9
Bibliography.....	9

1 Introduction

The LLVM compiler infrastructure project is a set of compiler and toolchain technologies which can be used to develop a front-end for any programming language and a back end for any instruction set architecture.

It achieves that with a three phase design whose major components are front-end, optimizer and back-end. The front-end parser the source code according to the production rules of the language, and creates an Abstract Syntax Tree(AST), to represent the source code.

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target.

The back end (also known as the code generator) then maps the code onto the target instruction set.

In this last part of the project the purpose was to finish the compiler program that had been created during the first 2 parts of the project in order to add the translation of the source code into a LLVM IR file which is the representation of the code in the compiler.

2 Background

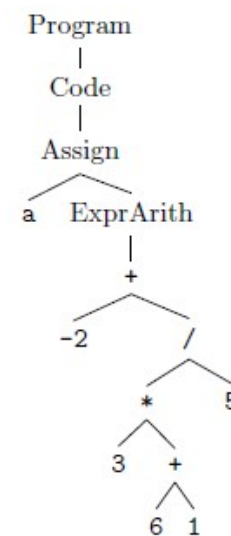
2.1 AST

The AST (Abstract syntax tree) is a tree representation of the abstract syntactic structure of the source code of a program. Each node represents a construct occurring in the code.

It omits unnecessary details from the inputted code such as parentheses and keywords that are always placed in specific spots according to the production rules of the used language.

```
BEGINPROG
  a := -2 + 3 * (6 + 1) / 5
ENDPROG
```

(a) A very simple program



(b) An abstract representation of the parse tree of the program in Fig. 1a.

Figure 1: AST of a simple program

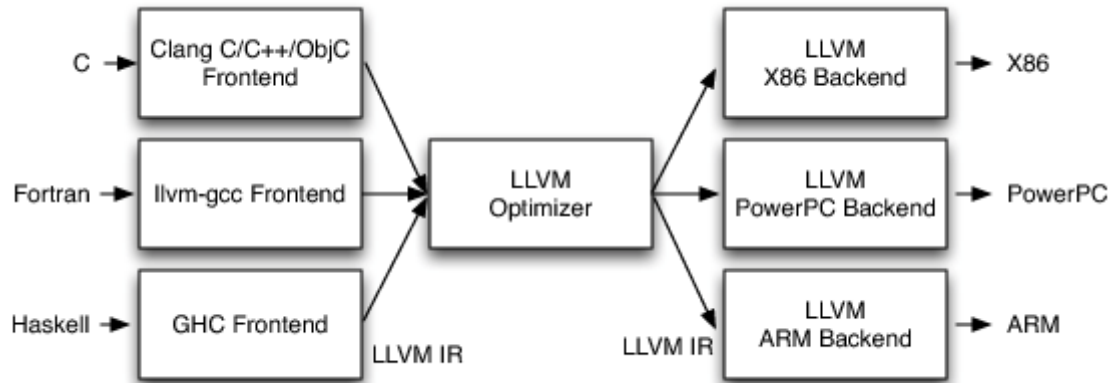
Its abstract nature allows it to demonstrate the functionality of the source code in a way that can be easily interpreted by an automaton, a property that we'll use during this part of the project.

2.2 LLVM IR

LLVM IR is a part of the LLVM compiler infrastructure project and more specifically it's the form it uses to represent code in the compiler.


LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations

In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code.



3 Implementation

3.1 Overview

The process of translating the code was divided into two main phases, the first phase required the generation of the AST which consisted of interpreting the  source code using the parser that was created during the previous part of the project, then the second phase relied on the results obtained from the first phase, using an automaton to read through the generated AST in order to generate the corresponding LLVM code.

3.2 Phase 1

During the previous part of the project the parser was able to generate a ParseTree based on the compiled code.

A ParseTree represents the syntactic structure of the source code. It contains all of the **LexicalUnits** that appear in the file arranged according to the production rules of the language based on which the source code was written. It consists of *Branch nodes* that represent a production rule and *Leaf nodes* that represent a symbol. The ParseTree is structured in such a way that if the *Leaf Nodes* are read from right to left the result will be a string that represents the original code.

In order to generate an AST we had to change the code so that it omitted unnecessary information such as *Leaf Nodes* that contain keywords with fixed places, parentheses and endline markers and *Branch Nodes* that exist only as a result of the LL(1) transformation process.

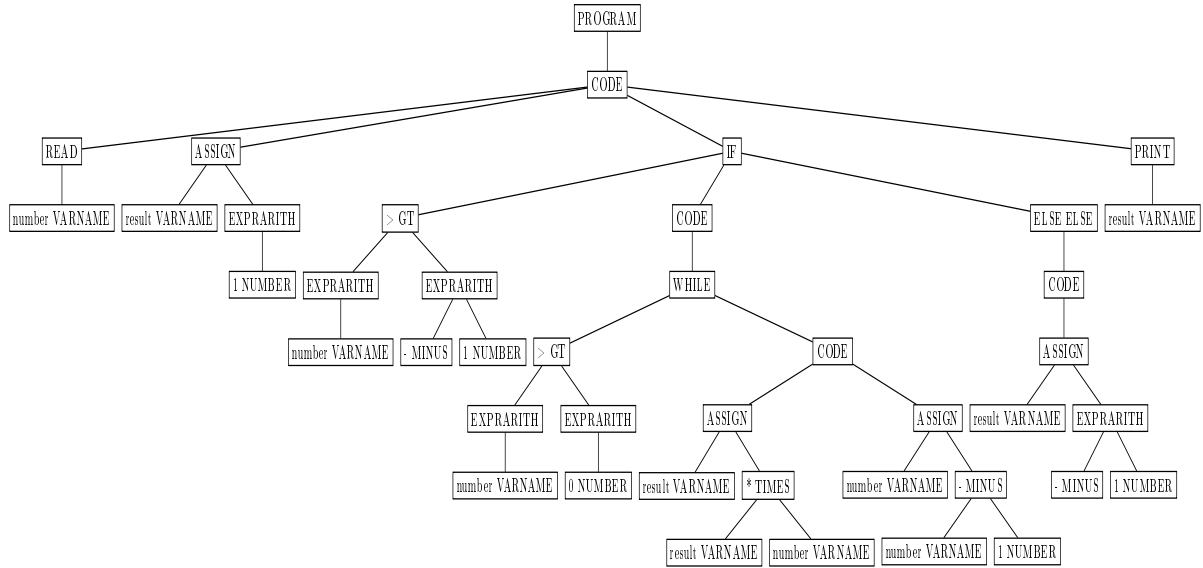


Figure 2: AST of the sample Factorial code

3.3 Phase 2

This phase consists of creating a program that reads the AST that was generated from the first phase. This program consists of functions that are able to recognize the production rules of the source code.

Each function takes as a parameter the node of the AST that it has to read and then calls the functions that represent the next parts of the rule with the node's children as parameters.

For each command of the source code the program creates an equivalent operation composed of multiple lines in LLVM and then saves it.

```

31
32     public Operation PROGRAM(ParseTree node) {
33         // calling the predefined read and write before the main
34         String equation = this.read_write.readWrite();
35         this.stack.add(new Operation(equation, ""));
36         // Beginning of the program
37         equation = "define i32 @main() {\r\n" +
38                 "entry: \r\n";
39         this.stack.add(new Operation(equation, " "));
40
41         try { this.CODE(node.getChild(0));
42             // End of the program
43             equation = " ret i32 0\r\n" +
44                     "}\r\n";
45             this.stack.add( new Operation(equation, ""));
46             return null ;
47
48         }catch( IndexOutOfBoundsException e ) {
49             System.out.print("CALLING NON EXISTING CHILD");
50             return null;
51         }
52     }
53

```

Figure 3: Excerpt from the *LlvmGenerator* class

Observation:

In the excerpt above, the `<program>` function is displayed, receives as a parameter the root node, creates the beginning and end of the LLVM code and then calls the `<code>` function with the child of the current node.

3.3.1 Bonus

- Output: the option to save the resulting LLVM code into an *.ll* file

3.3.2 Main class

The main class simply, puts all the pieces together: receiving the following arguments from the command line:

1. **-o:** stands for the file in which to write the LLVM IR code (*.ll* extension is expected, else an error is returned).
2. **inputFile.fs:** program file containing the input code (*.fs* extension is expected)

Afterward, it parses the program and generates the LLVM IR code.

4 Conclusion

In Summary, translating source code to a different programming language requires generating a AST which gives us the functionality of the original code in a form that can be easily understood by a machine and then using a finite automaton to read the generated AST and write the corresponding code in the second programming language.

With this last part of the project we have created a complete compiler that is able to read a file written in the Fortran programming language, parse it and then translate it in an assembly type language that can be used to create machine code and be executed in any type of machine.

Bibliography

1. Reinhard Wilhelm, Helmut Seidl, Sebastian Hack. *Compiler Design* .
2. GILLESGEERAERTS, GUILLERMO A. PÉREZ. *Introduction to language theory and compiling*.
3. LLVM. The Architecture of open source applications . [Online]
<http://www.aosabook.org/en/llvm.html>
4. Abstract Syntax Tree. Wikipedia . [Online]
https://en.wikipedia.org/wiki/Abstract_syntax_tree
5. Parse Tree. *Wikipedia* [Online] https://en.wikipedia.org/wiki/Parse_tree
6. *Introduction to language theory and compiling* .