Université Libre de Bruxelles

FORTR-S compiler

INFO-F403 - Part 3

Baudru Julien

December 19, 2020

Contents

1 Introduction **2**

2 User guide **2**

3 Structure **2**
3.1 Overall structure 2
3.2 Implementation 3

4 Difficulties encountered **3**

5 Examples **4**
5.1 Example N°1 4
5.2 Example N°2 4
5.3 Example N°3 5
5.4 Example N°4 6

1 Introduction

The main goal of this last part of the project is to improve the parser created in the previous part so that it can generate code corresponding to the semantics of the FORTR-S language. The code produced by the compiler during this last phase will be an LLVM (Low Level Virtual Machine) type language that can be read by the *llvm-as* tool.

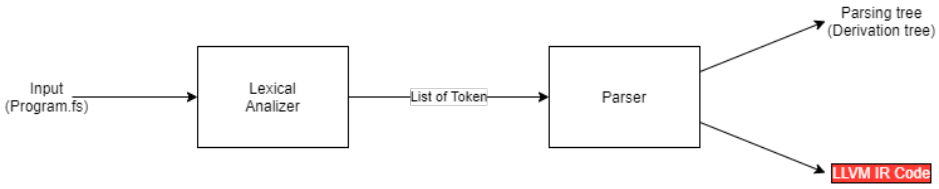

Figure 1: Compilation steps

2 User guide

The compiler presented in these pages can create a type *.ll* file from a *Fortr-S* program.
The command to generate the file of type *.ll* is the following :

```
java -jar dist/part3.jar test/<input_file>.fs -o <output_file>.ll
```

By default, the contents of the .ll file will be displayed on the standard output when the following command is entered by the user :

```
java -jar dist/part3.jar test/<input_file>.fs
```

Note that you can always use the command of the second part to generate the *Latex* file containing the derivation tree, as a reminder the command is the following :

```
java -jar dist/Part3.jar -wt more/tree.tex test/Factorial.fs
```

3 Structure

3.1 Overall structure

The structure of the project is the following :

```
part3:
|
|---dist
|     |---part3.jar
|
|---doc
|     |---Javadoc
|     |     |...
|     |---Report.pdf
|
|---more
|     |---tree.tex
|
|---src
|     |---CodeGenerator.java
|     |---LexicalAnalizer.java
|     |---LexicalAnalyzer.flex
|     |---LexicalUnit.java
|     |---Main.java
|     |---Parser.java
|     |---ParseTree.java
|     |---Symbol.java
|
|---test
      |---Factorial.fs
      |---TestOp.fs
```

The only notable difference from the structure of the previous section is the addition of a *CodeGenerator* class. The usefulness of this class is explained in the following point.

3.2 Implementation

This third part is based on the previous part and not on the correction available on the *Université Virtuelle*. The only difference from the previous part is the addition of a new class named *CodeGenerator*. As its name indicates, the purpose of this class is to generate the LLVM code. To do so, this class uses the derivation tree.

The first step is to walk (in order) through the previously created derivation tree. The tree traversal will therefore be done in the direction in which the instructions appear in the program, as a reminder the logic of the traversal made here is presented on the diagram below.

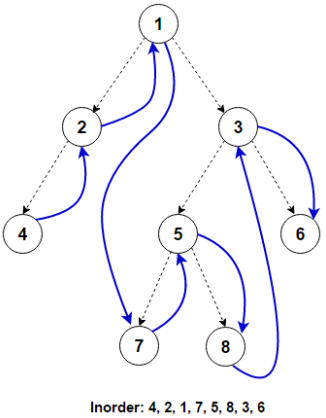

Figure 2: Inorder tree traversal

An important implementation choice to mention is the fact that the compiler presented here does not use an *ABS* (c.f. *Abstract Syntax Tree*) to create the *llvm* code. Thus, the *llvm* code is directly generated from the derivation tree, this choice was made to save time when designing this program.

4 Difficulties encountered

Among the various difficulties encountered, those that were the most interesting to solve were the following.

Firstly, the management of the memory linked to the variable. The difficulty for this part is to always save the new values of these variables (c.f. *store*) and to fetch their value when needed (c.f. *load*).This is especially the case when arithmetic expressions use variables, as in the example below.

```
BEGINPROG Example
x := 1
y := x + 8
ENDPROG
```

Thus the corresponding *llvm* code in this type of case will be the following :

```
define i32 @main() {
  entry:
  %x = alloca i32
  store i32 1, i32* %x
  %y = alloca i32
  %0 = load i32, i32* %x
  %1 = add i32 %0, 8
  store i32 %1, i32* %y
  ret i32 0
}
```

Then, the possibility to have several *if* statement or *while* statement in the program. To solve this problem, all you need is an if and while counter to be able to distinguish the labels of the different conditional blocks and loop blocks, that's why we find an *ifcounter* and *whilecounter* variable in the *CodeGenerator* class. An example of such a case is given here 5.2. The same principle is applied to the *cond* label within the if and while statements.

5 Examples

The following set of examples is intended to show that the transition from *Fortr-S* to *LLVM* works as expected.

5.1 Example N°1

As a first example let's take the first example given in the second part of this project, the simplistic code of this program was the following :

```
BEGINPROG TestOp
x := 3 + 7 * 7
y := 10 * 5 - 7
ENDPROG
```

The corresponding LLVM code that is generated by the compiler is the following :

```
define i32 @main() {
  entry:
  %x = alloca i32
  %0 = mul i32 7, 7
  %1 = add i32 %0, 3
  store i32 %1, i32* %x
  %y = alloca i32
  %2 = mul i32 10, 5
  %3 = sub i32 %2, 7
  store i32 %3, i32* %y
  ret i32 0
}
```

Note : The code presented here is in the *test* folder of this project, it has the name *TestOp.fs*.

5.2 Example N°2

The example below is intended to show that the arithmetic expressions within the conditions work as expected.The *Fortr-S* code for this example is the following :

```
BEGINPROG Test3
// The program's ouput is 1
input := 5
x := 1
res := 0
IF ((input + 1) > (x - 1)) THEN
    res := 1
ELSE
    res := -1
ENDIF
ENDPROG
```

The corresponding LLVM code that is generated by the compiler is the following :

```
define i32 @main() {
  entry:
  %input = alloca i32
  store i32 5, i32* %input
  %x = alloca i32
  store i32 1, i32* %x
  %res = alloca i32
  store i32 0, i32* %res
  %0 = load i32, i32* %input
  %1 = add i32 %0, 1
  %2 = load i32, i32* %x
  %3 = sub i32 %2, 1
  br label %ifentry0
  ifentry0:
    %cond0 = icmp slt i32 %1, %3
    br i1 %cond0, label %iftrue0, label %iffalse0
  iftrue0:
    store i32 1, i32* %res
    br label %ifend0
  iffalse0:
    store i32 -1, i32* %res
    br label %ifend0
  ifend0:
  ret i32 0
}
```

Note : The code presented here is in the *test* folder of this project, it has the name *Test3.fs*.

5.3 Example N°3

In a similar way to the previous part of this project, the example below is a very simple program, it asks the user for a number and if this number is positive then it prints 1, otherwise it prints -1.

```
BEGINPROG Example
/* Prints 1 if the input is a postive value,
otherwise prints -1.
*/
READ(input)
res := 0
IF (input > 0) THEN //Positive value
    res := 1
ELSE                //Negative value
    res := -1
ENDIF
PRINT(res)
ENDPROG
```

The corresponding LLVM code that is generated by the compiler is the following :

```
@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32
     0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...) #1

@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32
    0, i32 0), i32* %1)
  %3 = load i32, i32* %1, align 4
  ret i32 %3
}

declare i32 @scanf(i8*, ...) #1

define i32 @main() {
  entry:
  %input = alloca i32
  %0 = call i32 @readInt()
  store i32 %0, i32* %input
  %res = alloca i32
  store i32 0, i32* %res
  br label %ifentry0
  ifentry0:
    %1 = load i32, i32* %res
    %2 = load i32, i32* %input
    %cond0 = icmp slt i32 %1, %2
    br i1 %cond0, label %iftrue0, label %iffalse0
  iftrue0:
    store i32 1, i32* %res
    br label %ifend0
  iffalse0:
    store i32 -1, i32* %res
    br label %ifend0
  ifend0:
  %3 = load i32, i32* %res
  call void @println(i32 %3)
  ret i32 0
}
```

Note : The code presented here is in the *test* folder of this project, it has the name *Test.fs*.

5.4 Example N°4

Unlike parts 1 and 2 of this project, the last example is a program displaying the result of the factorial calculation for the entered number.

```
BEGINPROG Factorial

/* Compute the factorial of a number
   If the input number is negative, print -1. */

READ(number)              // Read a number from user input
result := 1

IF (number > -1) THEN
  WHILE ( number > 0 ) DO
    result := result * number
    number := number - 1   // decrease number
  ENDWHILE
ELSE                      // The input number is negative
  result := -1
ENDIF

PRINT(result)
ENDPROG
```

The corresponding LLVM code that is generated by the compiler is the following :

```
@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32
     0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...) #1

@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32
    0, i32 0), i32* %1)
  %3 = load i32, i32* %1, align 4
  ret i32 %3
}

declare i32 @scanf(i8*, ...) #1

define i32 @main() {
  entry:
  %number = alloca i32
  %0 = call i32 @readInt()
  store i32 %0, i32* %number
  %result = alloca i32
  store i32 1, i32* %result
  br label %ifentry0
  ifentry0:
    %1 = load i32, i32* %number
    %cond0 = icmp slt i32 -1, %1
    br i1 %cond0, label %iftrue0, label %iffalse0
  iftrue0:
  br label %whileentry0
  whileentry0:
    %2 = load i32, i32* %number
    %cond1 = icmp slt i32 0, %2
    br i1 %cond1, label %whilenext0, label %whileend0
  whilenext0:
    %3 = load i32, i32* %number
    %4 = load i32, i32* %result
    %5 = mul i32 %4, %3
    store i32 %5, i32* %result
    %6 = load i32, i32* %number
    %7 = sub i32 %6, 1
```

```
49      store i32 %7, i32* %number
50      br label %whileentry0
51    whileend0:
52      br label %ifend0
53    iffalse0:
54      store i32 -1, i32* %result
55      br label %ifend0
56    ifend0:
57    %8 = load i32, i32* %result
58    call void @println(i32 %8)
59    ret i32 0
60  }
```

Note : The code presented here is in the *test* folder of this project, it has the name *Fibonacci.fs*.