FORTRESS
Introduction to language theory and compiling
Project – Part 3

Gilles GEERAERTS   Léonard BRICE   Sarah WINTER

November 29, 2022

Statement

For this third and last part of the project, we ask you to augment the recursive-descent LL(1) parser you have written during the first and second parts to let it *generate code* that corresponds to the semantics of the FORTRESS program that is being compiled (those semantics are informally provided in Appendix A). The output code must be LLVM intermediary language (LLVM IR), which you studied during the practicals. This code must be accepted by the `llvm-as` tool, so that is can be converted to machine code. It is **not allowed** to first compile to another language (*e.g.* C) and then use the language compiler to get LLVM IR code.

When generating the code for arithmetic and boolean expressions, pay extra attention to the associativity and priority of the operators.

Bonus

For this last part of the project, you can enrich FORTRESS with several features:

- syntactic sugar/simple extensions, *e.g.* `FOR` loops
- functions (you can for example introduce new keywords `BEGINPROG`/`ENDPROG` and allow them to appear multiple times)
- additional types: boolean , float numbers and strings, or even arrays, lists, ...
- recursive functions

They are sorted by increasing difficulty, but you can choose any, or all. If you would like to add a feature that is not in the list, tell us first. You can also provide compiling optimizations (*dead code elimination*, *inlining*, ...), but this is less rewarding for you[1] since such optimisations are most likely provided by LLVM.

You have entire freedom of implementation for these bonuses; in particular, you can enrich the keywords and syntax, as long as FORTRESS is a subset of your language (any FORTRESS program must compile correctly). You are however required to explain what you did and how you did it in your report (we are not supposed to guess how your program works, nor what bonus you implemented), and you must provide test files to demonstrate your additional features. If you have any questions, please send us an email.

This bonus can get you *up to five point.* However, if you obtain more than 100/100, it will *not* be passed on to your exam grade. Also, note that this is only a *bonus*: it is better to provide a working FORTRESS compiler than a buggy Mega-FORTRESS one.

[1]In terms of interest, not in terms of grading: this will give you as many bonus points as the previous ones.

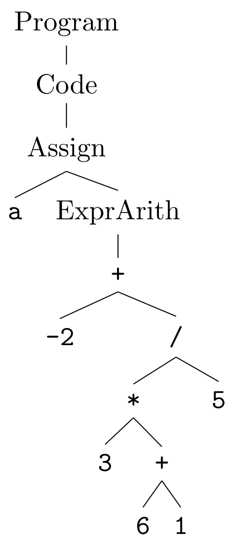
Guidelines

Those are only guidelines, and you have entire freedom of implementation for the code generator (except of course using compiling tools such as CUP).

To generate code from your parser, you can first modify it so that it builds an an Abstract Syntax Tree, either by doing it directly during the parsing or by post-processing the Parse Tree. Abstract Syntax Trees are meant to abstract away some non-terminals to obtain the very structure of the program. For instance, consider the program in Fig. 1a:

```
BEGIN
  a := -2 + 3 * (6 + 1) / 5
END
```

(a) A very simple program



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

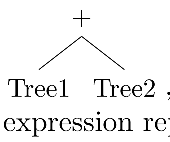
Figure 1: A simple program and its corresponding AST.

Its parse tree can then be converted to the AST of Figure 1b. Note that here, since `*` and `/` have the same priority, the parsing is done left-associatively, *i.e.* the arithmetic expression should be parenthesized as $-2 + (3 * (6 + 1))/5 = 2$ and not $-2 + 3 * ((6 + 1)/5) = 1$ (this matters because "`/`" is the *euclidean* division).

You should draw inspiration from this example to design an abstract representation of <If>, <While>, and the like. Note in particular that some non-terminals were omitted, since they have no semantic meaning; they are only dummy non-terminals used to simulate lists, *e.g.* <InstList>, enforce priorities of operators, or remove left-recursion.

Note that there is no formal requirements how simple the AST must be. If you are comfortable with the AST, then it is simple enough. If you prefer to directly work on the Parse Tree without simplification you may do so.

That being done, you can walk the AST and generate code. For example, when walking a node



, you should first evaluate the expression represented by Tree1, then evaluate the expression represented by Tree2, then compute the result of the addition and store it in an unnamed variable (whose number should be deduced from the ones used in Tree1 and Tree2). It should give something of the form:

[sequence of instructions to evaluate Tree1]

[`%n` is assigned the result of Tree1]

[sequence of instructions to evaluate Tree2, unnamed variables start at $n + 1$]

[`%m` is assigned the result of Tree2]

`%p`[2] = `%n` + `%m`

Of course, you can also directly generate the code from the parse tree, or obtain the AST directly from the parser, or even do everything at once, but this might be more complex: this decomposition separates the difficulties.

Requirements

You must hand in:

- A PDF report containing the necessary justifications, choices and hypotheses;
- the source code of your compiler;
- the JAVADOC;
- the FORTRESS example files you used to test your compiler;
- all required files to evaluate your work.

You must structure your files in five folders:

- `doc` contains the JAVADOC and the PDF report;
- `test` contains all your example files;
- `dist` contains an executable JAR **that must be called part3.jar**;
- `src` contains your source files;
- `more` contains all other files.

Your implementation must contain:

1. your scanner if you were able to use it for parsing, or the one we provided;
2. your parser, except if it does not work properly, in which case you may use ours;[3]
3. an executable public class that reads the file given as argument and writes on the standard output stream the LLVM intermediary code. **Do not write any output from Part 1 or 2 on the standard output stream; only your LLVM code!**

The command for running your executable must be: `java -jar part3.jar inputFile`

You need to compress your folder (in the *zip* format—no *rar* or other format), **following the same naming convention as for Part 2**: `Part3_Surname1(_Surname2)?.zip` where `Surname1` and, if you are in a group, `Surname2` are the last names of the student(s) of the group (in alphabetical order). Exactly one team member needs to submit it on the Université Virtuelle before **December, 22nd**.

[2]At this point, you can deduce the value of p.
[3]It will be available on the UV

A Informal semantics of the Fortress language

We only provide an informal description of the semantics, since a formal one would needlessly complicate the matter for such a simple language. There is nothing surprising here, since those semantics are similar to the ones of everyday languages. The value to which a nonterminal <NT> evaluates will be denoted by $[\![\text{NT}]\!]$, *e.g.* $[\![\text{ExprArith}]\!]$.

- The code represented by <Program> should be the result of the processing of <Code> (in other words, the `BEGIN`, `END`, `ProgName` markers are just markers).
- <Code> is a list of instructions <Instruction>, which should be executed sequentially.
- <Assign>: [VarName] := <ExprArith> means the program should store $[\![\text{ExprArith}]\!]$ in the variable VarName (which should be stored in a memory location, not simply in an LLVM variable).
- <If>: `IF` <Cond> `THEN` <Code> `ENDIF` means that if the condition computed by <Cond> (*i.e.* $[\![\text{Cond}]\!]$) is true, then <Code> should be executed, otherwise the program should go to the next instruction.
- <If>: `IF` <Cond> `THEN` <Code1> `ELSE` <Code2> `END` means that if $[\![\text{Cond}]\!]$ is true, then <Code1> should be executed, otherwise <Code2> should be executed instead.
- <While>: `WHILE (` <Cond> `) DO` <Code> `END` means that the program should test <Cond>, then execute <Code> if $[\![\text{Cond}]\!]$ is true and then repeat, otherwise it should do nothing[4].
- <Print>: `PRINT(`[Varname]`)` should print the value of [Varname] to `stdout`.
- <Read>: `READ(`[Varname]`)` should read an integer from `stdin`, and store it in the corresponding [Varname].
- <ExprArith>: those are the semantics of usual arithmetic expressions written in infix notation, with the conventional precedence of operators (given in Part 2).

[4]Giving formal semantics to `WHILE` is actually very hard. Here is a hint of how it could be done: <While> can be unrolled as `IF` <Cond> `THEN` <Code> `WHILE` <Cond> `DO` <Code> `END END`.