

# IMP

## Introduction to language theory and compiling

### Project – Part 3

Gilles GEERAERTS

Marie VAN DEN BOGAARD

Léo EXIBARD

December 13, 2017

## 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 part to let it *generate code* that corresponds to the semantics of the IMP program that is being compiled (those semantics are informally provided in Appendix B). 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 it 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 expressions, pay extra attention to the associativity and priority of the operators. The modification of the grammar you have done during part 2 should help you in that respect.

## Bonus

For this last part of the project, you can enrich IMP with several features: syntactic sugar (*e.g.* parenthesized boolean expressions), variable scoping (*i.e.* handle programs such as `begin x := 1; for x = 0 to 5 do print(x) done; print(x) end`), functions, types (floats 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<sup>1</sup> since they are probably provided by LLVM.

You have entire freedom of implementation for this bonus; in particular, you can enrich the keywords and syntax, as long as IMP is a subset of your language (any IMP program must compile correctly, except for those containing additional reserved keywords). 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). If you have any questions, please send us an email.

This bonus can get you *up to one point*, which will be added to your overall project grade (*i.e.* it can compensate for points you lost on previous parts). However, if you obtain more than 8/8, 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 IMP compiler than a buggy SUPERIMP one.

---

<sup>1</sup>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 that you are not allowed to use CUP to generate the code itself, only to obtain the parse tree if it is difficult to get it directly from your parser).


To generate code from your parser, you can:

1. Modify it to make it generate a parse tree
2. From the parse tree, deduce an Abstract Syntax Tree, which abstracts away some non-terminals to obtain the very structure of the program. For instance, consider the following program:

```
begin
  a := 1 + 2 * (3 + 4) / 5
end
```

Be careful that here, since `*` and `/` have the same priority, the parsing is done according to the left associativity, *i.e.* the arithmetic expression should be parenthesized<sup>2</sup> as  $1 + (2 * (3 + 4)) / 5$ . Its parse tree, depicted in Figure 1, can be converted to the AST of Figure 2 (both figures can be found in Appendix A).

You should draw inspiration from this example to design an abstract representation of `<If>`, `<For>`, `<While>`, and the like. Note in particular that some non-terminals were omitted (*e.g.* `ProdExpr` and `InstTail`), since they have no semantic meaning: they are only dummy non-terminals respectively used to enforce priority and to simulate lists.

3. Walk the AST and generate code. For example, when walking a node  `Tree1` `Tree2`, 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]
%p3 = %n + %m
```

Of course, you can also directly generate the code from the parse tree (*i.e.* avoid step 2), or obtain the AST directly from the parser (*i.e.* avoid step 1), or even do everything at once, but this might be more complex: this decomposition separates the difficulties.

---

<sup>2</sup>If you removed ambiguity according to the method in the lecture notes or in the practicals, this should be the case.

<sup>3</sup>At this point, you can deduce the value of  $p$ .

## Requirements

You must hand in:

- A PDF report containing the necessary justifications, choices and hypotheses;
- The source code of your compiler;
- The IMP 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, otherwise the one we provided;
2. your parser, except if it does not work properly, in which case you may use ours;
3. an executable public class `Main` that reads the file given as argument and writes on the standard output stream the LLVM intermediary code. You can provide, **as options**<sup>4</sup>, the possibility to output it to a `.ll` file, and to interpret the code (for instance, `java -jar part3.jar inputFile.imp -o outputFile.ll` would output the LLVM IR code to `outputFile.ll`, and `java -jar part3.jar -exec inputFile.imp` would interpret the code).

The command for running your executable must be:

```
java -jar part3.jar inputFile
```

You will 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), and you will submit it on the Université Virtuelle before **December, 22<sup>nd</sup>**.

---

<sup>4</sup>It is important that, by default, your program outputs the LLVM IR code on `stdout` (even if you use a temporary file to store the LLVM IR code, in which case you just have to print it on screen).

## A Example of Abstract Syntax Tree

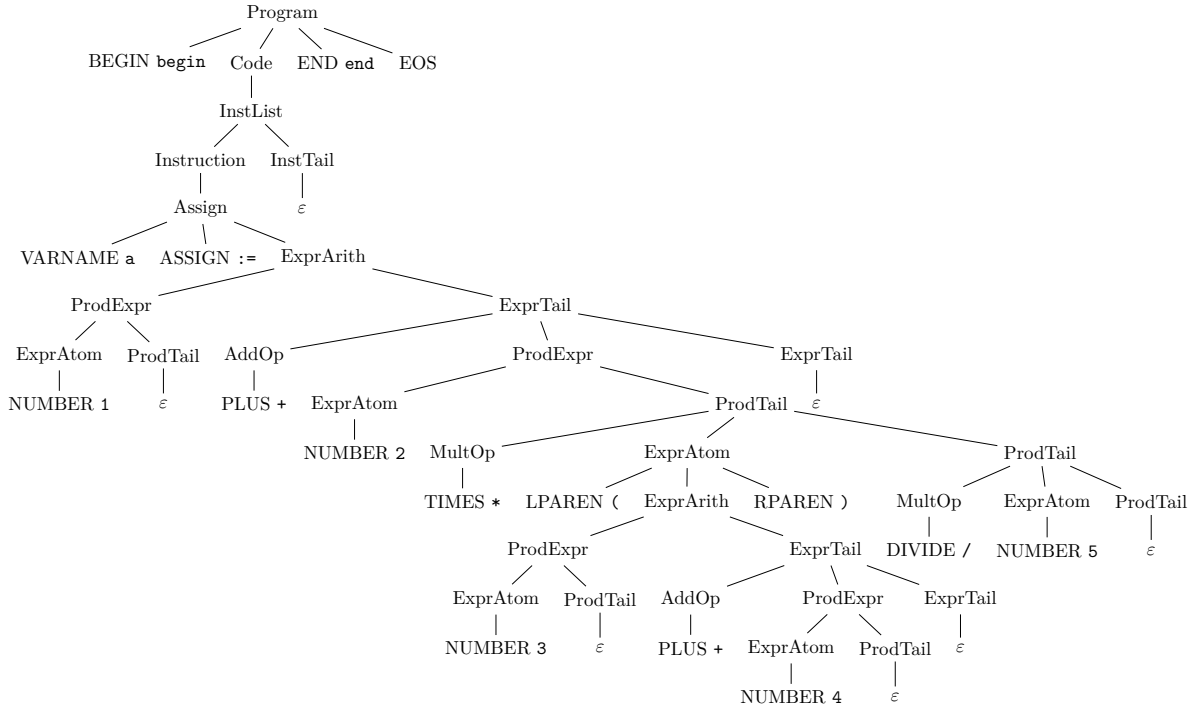


Figure 1: The parse tree of the program `begin a := 1 + 2 * (3 + 4) / 5 end.`

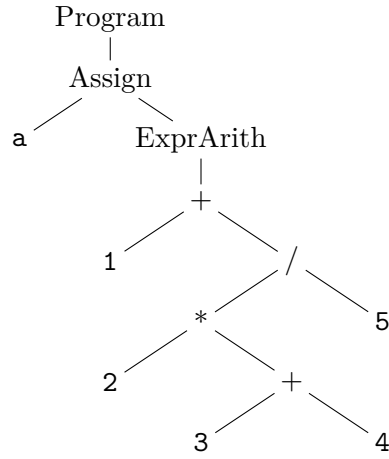


Figure 2: An abstract representation of the parse tree of Fig. 1 .

## B Informal semantics of the IMP 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  $\langle NT \rangle$  evaluates will be denoted by  $\llbracket NT \rrbracket$ , *e.g.*  $\llbracket \text{ExprArith} \rrbracket$ .

- The code represented by  $\langle \text{Program} \rangle$  should be the result of the processing of  $\langle \text{Code} \rangle$  (in other words, the **begin** and **end** markers are just markers).
- When  $\langle \text{Code} \rangle$  is empty, the program should do nothing. Otherwise, it should be the result of  $\langle \text{InstList} \rangle$ .
- $\langle \text{InstList} \rangle$  is a list of instructions  $\langle \text{Instruction} \rangle$ , which should be executed sequentially.
- $\langle \text{Assign} \rangle$ :  $[\text{VarName}] := \langle \text{ExprArith} \rangle$  means the program should store  $\llbracket \text{ExprArith} \rrbracket$  in the variable  $\text{VarName}$  (which should be stored in a memory location).
- $\langle \text{If} \rangle$ : **if**  $\langle \text{Cond} \rangle$  **then**  $\langle \text{Code} \rangle$  **endif** means that if the condition computed by  $\langle \text{Code} \rangle$  (*i.e.*  $\llbracket \text{Code} \rrbracket$ ) is true, then  $\langle \text{Code} \rangle$  should be executed, otherwise the program should go to the next instruction.
- $\langle \text{If} \rangle$ : **if**  $\langle \text{Cond} \rangle$  **then**  $\langle \text{Code1} \rangle$  **else**  $\langle \text{Code2} \rangle$  **endif** means that if  $\llbracket \text{Code} \rrbracket$  is true, then  $\langle \text{Code1} \rangle$  should be executed, otherwise  $\langle \text{Code2} \rangle$  should be executed instead.
- $\langle \text{While} \rangle$ : **while**  $\langle \text{Cond} \rangle$  **do**  $\langle \text{Code} \rangle$  **done** means that the program should test  $\langle \text{Cond} \rangle$ , then execute  $\langle \text{Code} \rangle$  if  $\llbracket \text{Cond} \rrbracket$  is true and then repeat, otherwise it should do nothing<sup>5</sup>.
- $\langle \text{For} \rangle$ : **for**  $[\text{Varname}]$  **from**  $\langle \text{ExprArith1} \rangle$  **by**  $\langle \text{ExprArith2} \rangle$  **to**  $\langle \text{ExprArith3} \rangle$  **do**  $\langle \text{Code} \rangle$  **done** means that the program should initialize the variable named  $\text{VarName}$  to  $\llbracket \text{ExprArith1} \rrbracket$ , then execute  $\langle \text{Code} \rangle$  and increment  $\text{VarName}$  by  $\llbracket \text{ExprArith2} \rrbracket$  (which can be negative), and so on until  $\text{VarName}$  exceeds  $\llbracket \text{ExprArith3} \rrbracket$ . If  $\llbracket \text{ExprArith} \rrbracket = 0$ , the loop should go infinitely. If  $\llbracket \text{ExprArith2} \rrbracket > 0$  and  $\llbracket \text{ExprArith3} \rrbracket < \llbracket \text{ExprArith1} \rrbracket$ , or conversely if  $\llbracket \text{ExprArith2} \rrbracket < 0$  and  $\llbracket \text{ExprArith3} \rrbracket > \llbracket \text{ExprArith1} \rrbracket$ , the loop should not execute at all, and the program should go to the next instruction.
- $\langle \text{For} \rangle$ : **for**  $[\text{Varname}]$  **from**  $\langle \text{ExprArith1} \rangle$  **to**  $\langle \text{ExprArith3} \rangle$  **do**  $\langle \text{Code} \rangle$  **done** is equivalent to **for**  $[\text{Varname}]$  **from**  $\langle \text{ExprArith1} \rangle$  **by** 1 **to**  $\langle \text{ExprArith3} \rangle$  **do**  $\langle \text{Code} \rangle$  **done**
- $\langle \text{Print} \rangle$ : **print**( $[\text{VarName}]$ ) should print the value of  $\text{VarName}$  to **stdout**.
- $\langle \text{Read} \rangle$ : **read**( $[\text{VarName}]$ ) should read an integer from **stdin** and store it in  $\text{VarName}$ .
- $\langle \text{ExprArith} \rangle$ : those are the semantics of usual arithmetic expressions written in infix notation, with the conventional precedence of operators (given in Part 2).
- $\langle \text{Cond} \rangle$ : those are the semantics of usual boolean expressions with only operators *and*, *or* and *not*, with the conventional precedence of operators (given in Part 2).

---

<sup>5</sup>Giving formal semantics to **while** is actually very hard. For completeness, here is a hint of how it could be done:  $\langle \text{While} \rangle$  can be unrolled as **if**  $\langle \text{Cond} \rangle$  **then**  $\langle \text{Code} \rangle$ ; **while**  $\langle \text{Cond} \rangle$  **do**  $\langle \text{Code} \rangle$  **done** **endif**.