PASCALmaispresque
Introduction to language theory and compiling
Project – Part 3

Gilles Geeraerts Mathieu Sassolas Léonard Brice

Info-F403 – 2023-2024

Work description

For this third and last part of the project, we ask you to augment the recursive-descent LL(1) parser you have written[1] during the first and second parts to let it *generate code* that corresponds to the semantics of the PASCALmaispresque 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.

Expected outcome

You must hand in all files required to compile and evaluate your code, as well as the proper documentation, including a **PDF report**. This must be structured into five folders as follows, and a `Makefile` must be provided.

`src/` Contains all source files required to evaluate and compile your work:

- the JFlex source file `LexicalAnalyzer.flex` for the scanner;
- all necessary java files, either generated (`LexicalAnalyzer.java`), provided (and maybe modified, such as `LexicalUnit.java`, `Symbol.java`...), or written on your own.
- a `Main.java` 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!**

`doc/` Contains the JAVADOC and the PDF report.

The PDF report should present your work, with all the necessary justifications, choices and hypotheses, as well as descriptions of your example files. Such report will be particularly useful to get you partial credit if your tool has bugs.

[1] If you are not satisfied with your own parser you can use the correction provided on the UV.

`test/` Contains all your example PASCALmaispresque files. You must provide enough examples to convince of the correctness of your compiler on the whole PASCALmaispresque language.

`dist/` Contains an executable JAR called `part3.jar`. The command for running your executable should therefore be: `java -jar part3.jar [FILE]`. Remember that it should only output the LLVM code and nothing else!

`more/` Contains all other files.

`Makefile` At the root of your project (i.e. not in any of the aforementioned folders). There is an example Makefile on the UV.

You will compress your root folder (in the *zip* format—no *rar* or other format), **which is named according to the following regexp**: `Part3_Surname1(_Surname2)?.zip`, where `Surname1` and, if you are in a group, `Surname2` are the last names of the student(s) (in alphabetical order); you are allowed to work in a group of maximum two students.

The *zip* file shall be submitted on Université Virtuelle before **December 21st, 2023, 23:59**, Brussels Grand-Place time.

Bonus

For this last part of the project, you can enrich PASCALmaispresque 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[2] since such optimizations 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 PASCALmaispresque is a subset of your language (any PASCALmaispresque 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 PASCALmaispresque compiler than a buggy Mega-PASCALmaispresque one.

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

A Informal semantics of the PASCALMAISPRESQUE 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 ⟦NT⟧, *e.g.* ⟦ExprArith⟧.

- The code represented by <Program> should be the result of the processing of <Code> (in other words, the `begin`, and `end` markers are just markers).
- <Code> is a `...`-separated list of instructions <Instruction>, which should be executed sequentially.
- <Assign>: [VarName] `:=` <ExprArith> means the program should store ⟦ExprArith⟧ in the variable VarName (which should be stored in a memory location, not simply in an LLVM variable).
- <If>: `if` <Cond> `then` <Instruction> `else` means that if the condition computed by <Cond> (*i.e.* ⟦Cond⟧) is true, then <Instruction> should be executed, otherwise the program should go to the next instruction.
- <If>: `if` <Cond> `then` <Instruction1> `else` <Instruction2> means that if ⟦Cond⟧ is true, then <Instruction1> should be executed, otherwise <Instruction2> should be executed instead.
- <While>: `while` <Cond> `do` <Instruction> means that the program should test <Cond>, then execute <Code> if ⟦Cond⟧ is true and then repeat, otherwise it should do nothing[3].
- <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 Table 1).
- <Cond>: those are the semantics of usual boolean expressions with only operators *and* and *or*, with the conventional precedence of operators (given in Table 1).

Operators	**Associativity**
`-` (unary)	right
`*`, `/`	left
`+`, `-` (binary)	left
`<`, `=`	left
`and`	left
`or`	left

Table 1: Priority and associativity of the PASCALMAISPRESQUE operators (operators are sorted in decreasing order of priority). Note the difference between *unary* and *binary* minus (`-`).

[3]Giving formal semantics to `while` is actually not trivial.