Introduction to language theory and compiling
Project – Part 2

Gilles Geeraerts Léonard Brice Sarah Winter

December 13, 2022

This report gives an overview of the typical mistakes that were made, and explains the different choices you had for the implementation. It also aims to show you what a typical report should look like in terms of structure and how to explain your choices. Obviously, you are not supposed to give a list of potential mistakes in your report.

1 Introduction

The Fortress language is a simple imperative language which provides basic constructors: variable names, instructions to read and print them, `while` loops, comperators, etc. This second part of the project consisted in implementing a recursive-descent, LL(1) parser for Fortress.

2 Preprocessing

You first had to modify the grammar so as to make it LL(1):

2.1 Removal of unproductive and/or unreachable variables

There were no unproductive nor unreachable variables, but to prove it you had to execute the algorithm given in the course.

2.2 Disambiguise the grammar

Precedence The case of arithmetic expressions is practically the same as the one seen during the practicals. The novelty lies in the unary minus, which enjoys highest precedence. Adding the derivation ¡A¿ → MINUS ¡A¿ (where ¡A¿ denotes your most atomic terminal in ¡ExprArith¿) implements this, because it ensures that the unary minus is "glued" to its corresponding expression.

Left-factoring Here, you could blindly apply the algorithms provided during the lessons. You could however be slightly smarter regarding the "else": here, the grammar can be both left- and right-factored, which translates into adding a `MaybeElse` non-terminal.

All the above modifications are implemented in the `Fortress_Grammar.txt` file in the `more/` directory.

2.3 First, Follow and the action table

The First and Follow sets, as well as the action table, can be computed by using the command `java -jar dist/ProcessGrammar.jar -pat more/Fortress_Grammar.txt` (`pat` stands for "print action table"). You can also write it to `filename.txt` by calling `java -jar dist/ProcessGrammar.jar -wat filename.txt more/Fortress_Grammar.txt`

Please note that the algorithm provided to compute the First and Follow sets, as well as the action table, may not be completely correct. So, if you want to use it to generate an action table for your own grammar treat the result with caution.

3 A few words about the implementation

3.1 Grammar processing

We implemented a simple grammar parser using `JFlex` (the language is simple enough to avoid using more complex tools). The First, Follow and Action Table computation, as well as their printing, are implemented in the `Grammar.java` class. Such implementation is meant to be as close as possible to the algorithm described in the Practical Session 6. The corresponding executable is implemented in the `ProcessGrammar.java` class. Again, we do not guarentee that this implementation is error free.

Note that the `Pair.java` is simply a quick implementation of the pair type, to be able to `Map` pairs to something (instead of doing a `Map` of `Map`s).

3.2 Modification of the `Symbol` class

To get more flexibility about the manipulation of terminals and variables, we chose to heavily modify the `Symbol` class, and split it into several classes: the `Terminal` and `NonTerminal` classes, which are simply enums of (non-)terminals, the `Symbol` class which is the union of `Terminal` and `NonTerminal`, and the `Token` class which plays the role of the old `Symbol` class. `NonTerminal` and `Token` are unioned under the `TreeLabel` class, which represents labels of parse tree nodes (cf the `ParseTree` class). Please note that this is certainly not the best way to do it in terms of code quality.

3.3 Computation of the parse tree

The parser keeps track of the current token — by current we mean here the one that is right after the reading head: the parser will never have to look at the token currently pointed by the reading head. It can look at it (this is the meaning of the

`switch(current.getType())` construct), and can advance the reading head using the `match(term)` function, which checks that the current token is indeed the expected terminal, consumes it and moves to the next token (this is exactly the Match operation seen in class).

The parse tree is computed during the parsing. It is represented using the `ParseTree` class, which represents a tree as a root label and the list of its children which are themselves parse trees. This corresponds to the classical, recursive definition of labeled trees.

Thus, there is one function per non-terminal of the grammar, and each function returns the parse tree rooted in the corresponding non-terminal. For instance, assume the parser has input `READ(num)` and is currently calling `read()` (we omit the calls to `program()` etc for clarity). Then, the `read()` function will return the tree depicted in Figure 1

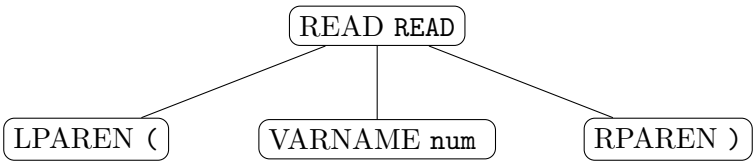

Figure 1: The output of `read()` on input `READ(num)`

When there are more than one possible rule to apply, the parser looks at the next token (`switch(current.getType())`) and decides accordingly.

At the end of the parsing, the program has built the entire parse tree, and the `main` function (in `WriteTree.java`) outputs it to the `.tex` file given as argument.

Note that our parser does not output the numbers of the rules during the parsing. This is because you won't need them for Part 3, and it should be quite straightforward to add this feature anyway: we wrote the corresponding rule as a comment everytime (e.g. `// Program -> BEGIN PROGNAME Code END` at l.42 of `Parser.java`), so just add `System.out.println(number of the rule)`.

3.4 Parsing errors

Parsing errors are implemented in the `ParseException` class. It outputs the read token, the token it expected, and, when useful, a list of tokens it would have expected. Such tokens are simply the ones which have a non-empty cell in the action table in the corresponding non-terminal line. If the error happens while trying to parse a non-terminal (and not during a match), it additionally outputs the corresponding non-terminal.

4 Conclusion

The implementation we provide is not meant to be perfect. However, it should suffice for Part 3 if you did not manage to implement a working parser during Part 2.