INFO-F-403

PART 2:
PARSER

EL MIRI Hamza
000479603

28/11/2022

Contents

1 Introduction

Building from Part 1 of the project, the aim is to write a LL(1) parser for the *FORTRESS* language. This document will present the grammar of the language in LL(1) form, go over the necessary computations to build the parser and finally its implementation. The working language will be *Java 17* and test examples will be provided.

2 Transformation of the grammar

Before doing anything, the ambiguity of the language must be removed. First, the grammar must be rewritten to take into account the associativity of the operators (see rules 13 -> 24 and 31 -> 34 in table 1). This step introduces ambiguity in the language in the form of left recursion, thus the grammar was transformed to remove it. Unproductive and unreachable symbols have to be removed since they are of no use to the parser. Fortunately, there were none in the grammar. Finally, the grammar was refactored (see rules 25 -> 27). Additionally, the original rules 2 - 5 were rewritten as rules 2 - 6 in table 1 preserving the grammar.

3 First¹ and Follow¹ sets

See table 2. The computation was done systematically since the algorithms are very straightforward to implement (see Class CFG).

The program can output the First¹ and Follow¹ sets when executed with the following command:

```
java -jar dist/part2.jar -s [file]
```

4 LL(1) action table

You can use the following command to print the action table:

```
java -jar dist/part2.jar -t [file]
```

It is a direct implementation of the algorithm seen during the practicals.

5 The implementation

5.1 The CFG class

This is the class that will represent the language. It takes a path to a grammar file as argument and parses it into a HashMap where the key is the variable and the value is an array containing the rules as strings, removes unproductive and unreachable symbols and computes the First[1] and Follow[1] sets. The file for the FORTRESS grammar is located in the folder `more/`
The syntax of the grammar file is as follows:

- Variables are enclosed in < ... > and cannot contain spaces or <>
- The left hand side of the rule is followed by − > to indicate that this variable produces a rule
- If a variable produces multiple rules, the first one must be on the same line as the variable while subsequent rules can contain empty spaces as a left hand side for ease of readability.
- Each token in a rule must be separated by a space (for ease of parsing)
- "e" is reserved as the ε character

While parsing the grammar file, the class will keep track of the order of the rules which is necessary to output the modified grammar after the cleanup operations since their implementation is not guaranteed to preserve the order of insertion even in a LinkedHashMap. Additionally, in order to identify the variables in the rules during the computations, a match against the RegExp pattern "<[^]+?>" is performed. This will be faster and simpler than iterating through the KeySet of the grammar HashMap.
The ActionTable class will use the First[1] and Follow[1] sets computed by this class, therefore, the ActionTable class requires the results of the computation to keep track of which rule produced which terminal token in the sets. This is why an auxiliary class Terminal was needed. It allows CFG to communicate to ActionTable which rule was used for the generation of each terminal symbol.

5.2 The ActionTable class

A very simple and straightforward implementation. It takes the First[1] and Follow[1] sets computed by CFG as parameters and computes the action table. To save on memory, the table is represented as a HashMap where the key is a string with the format *"v,t"* to parallel a *M[v,t]* lookup in a real table. This

method does not require memory allocation for empty cells in a matrix and is simple to implement. If the key does not exist, then $M[v,t]$ is not a valid transition from a non-terminal v with look-ahead t.

If the provided grammar is not LL(1), a conflict in the table will be detected and the user will be informed of the problematic rule.

5.3 The Symbol class

No modifications were made outside of a method to convert the data to LaTex format for the ParseTree.

5.4 The ParseTree class

A method to convert the node into LaTex was added in addition to a constructor for non-terminal nodes.

5.5 The Parser class

It is a recursive descent implementation. It incrementally builds the leftmost derivation sequence as well as the ParseTree within each recursive call.

Before parsing the file, it first makes a call to the LexicalAnalyzer from part 1 to make sure that the program respects the syntax rules.

5.6 Parsing Errors

When the Parser class detects an error, it will raise a ParserException error that informs the user of the token that caused it at line:column as well as a way to fix it if applicable.

6 Test Files

Multiple test files were written to test multiple user errors. They can be found under `test/parser`. For example:

- **Factorial.fs** is a correct program. Parsing it should yield its leftmost derivation: 1 2 10 30 4 2 6 11 12 17 21 32 24 20 16 4 2 7 25 12 17 21 31 24 20 13 17 21 32 24 20 16 2 8 28 12 17 21 31 24 20 13 17 21 32 24 20 16 2 6 11 12 17 21 31 22 31 24 20 16 4 2 6 11 12 17 21 31 24 19 21 32 24 20 16 4 3 4 3 27 2 6 11 12 17 21 32 24 20 16 4 3 4 2 9 29 4 3.

- **parserAccidentalCOMMA.fs** contains a COMMA after ELSE where it doesn't belong. Parsing it should inform the user of the error: *Unexpected token at 4:8 –> COMMA*

- **parserNestedOperations.fs** tests operation priority. parsing it should yield : 1 2 6 11 12 17 21 33 12 17 21 33 12 17 21 32 24 18 21 32 24 20 16 22 33 12 17 21 32 24 19 21 31 23 32 24 19 21 33 12 17 21 32 24 18 21 32 24 20 16 24 20 16 24 20 14 17 21 33 12 17 21 32 23 33 12 17 21 31 24 18 21 32 24 20 16 24 20 16 24 20 16 24 20 16 4 3. And its ParseTree at figure 1

(1) <Program> → BEGIN [ProgName] <Code> END
(2) <Code> → <Instruction> <InstNext>
(3) → ε
(4) <InstNext> → COMMA <Code>
(5) → ε
(6) <Instruction> → <Assign>
(7) → <If>
(8) → <While>
(9) → <Print>
(10) → <Read>
(11) <Assign> → [VarName] := <Cond>
(12) <Cond> → <ExprArith> <SimpleCond>
(13) <SimpleCond> → > <ExprArith> <SimpleCond>
(14) → < <ExprArith> <SimpleCond>
(15) → = <ExprArith> <SimpleCond>
(16) → ε
(17) <ExprArith> → <Product> <ExprArith'>
(18) <ExprArith'> → + <Product> <ExprArith'>
(19) → - <Product> <ExprArith'>
(20) → ε
(21) <Product> → <Atom> <Product'>
(22) <Product'> → * <Atom> <Product'>
(23) → / <Atom> <Product'>
(24) → ε
(25) <If> → IF <Cond> THEN <Code> <IFseq>
(26) <IFseq> → END
(27) → ELSE <Code> END
(28) <While> → WHILE <Cond> DO <Code> END
(29) <Print> → PRINT ([VarName])
(30) <Read> → READ ([VarName])
(31) <Atom> → [VarName]
(32) → [Number]
(33) → (<Cond>)
(34) → - <Atom>

Table 1: FORTRESS grammar

	First	Follow
<Program>	BEGIN	
<Code>	READ PRINT [VarName] ε IF WHILE	ELSE END
<InstNext>	, ε	ELSE END
<Instruction>	PRINT READ [VarName] WHILE IF	, ELSE END
<Assign>	[VarName]	, ELSE END
<Cond>	[VarName] (- [Number]	, ELSE) END THEN
<SimpleCond>	[VarName] (- [Number]	, ELSE) END THEN
<ExprArith>	[VarName] (- [Number]	, ELSE) END THEN <= >
<ExprArith'>	ε + -	, ELSE) END THEN <= >
<Product>	[VarName] (- [Number]	, ELSE) END THEN <= > + -
<Product'>	ε * /	, ELSE) END THEN <= > + -
<If>	IF	, ELSE END
<IfSeq>	ELSE END	, ELSE END
<While>	WHILE	, ELSE END
<Print>	PRINT	, ELSE END
<Read>	READ	, ELSE END
<Atom>	[VarName] (- [Number]	, ELSE) END THEN <= > + - * /

Table 2: First and Follow sets

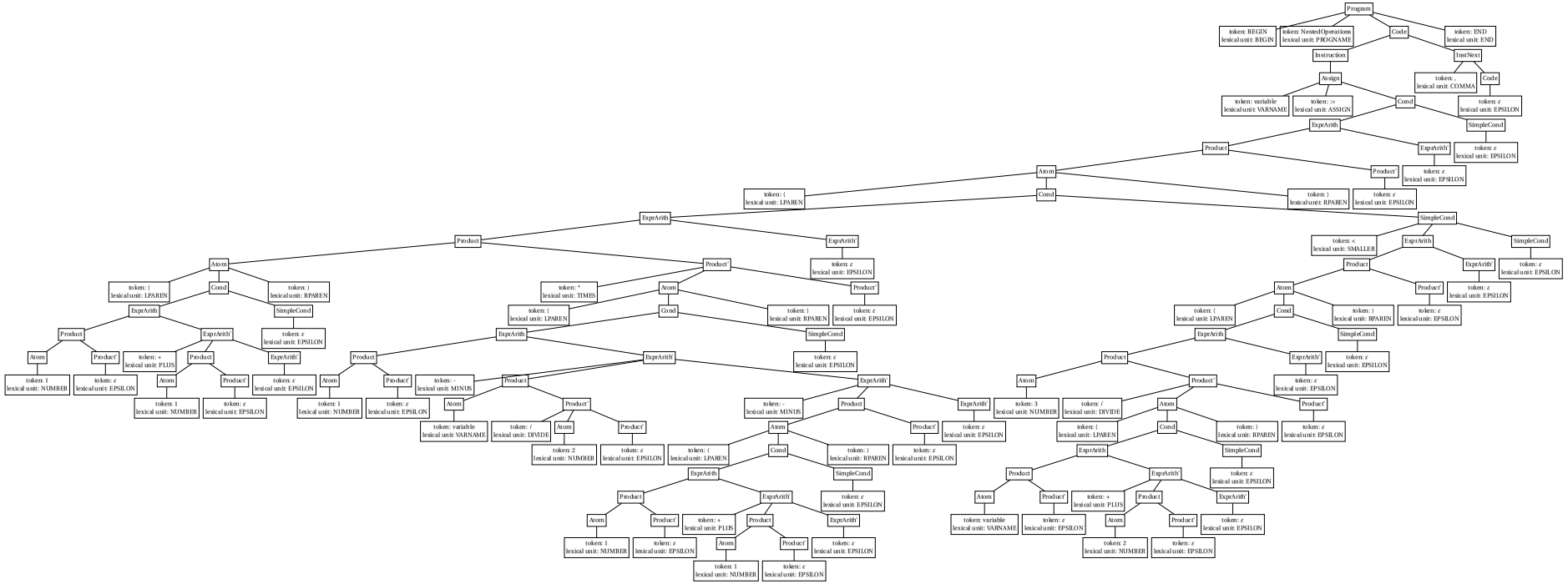

Figure 1: parserNestedOperations.fs