Lab 7 FLCD – Parser
Pop Daniel Avram + Popa Alex Ovidiu
936/1

Solution repository:
https://github.com/alexovidiupopa/flcd/tree/main/parser

Problem statement:
Implement a parser algorithm using the ll(1) parsing algorithm.
The representation of the parsing tree (output) will be the table (using father and sibling relation).

Input example:
(G1 + Sequence + Output)

g1.txt:
N = { S, A, B, C }
E = { (,), +, *, int }
S = S
P = {
    S -> A B,
    A -> (S) | int C,
    B -> + S | E,
    C -> * A | E
}

seq.txt:
(
int
)
+
int

tree.out

```
1 | S | None | None
2 | A | 1 | None
3 | B | 1 | 2
4 | ( | 2 | None
5 | S | 2 | 4
6 | ) | 2 | 5
7 | A | 5 | None
8 | B | 5 | 7
9 | int | 7 | None
10 | C | 7 | 9
11 | E | 10 | None
12 | E | 8 | None
13 | + | 3 | None
14 | S | 3 | 13
15 | A | 14 | None
16 | B | 14 | 15
17 | int | 15 | None
18 | C | 15 | 17
19 | E | 18 | None
20 | E | 16 | None
>>
```

Parsing output: 1 2 1 3 7 5 4 1 3 7 5

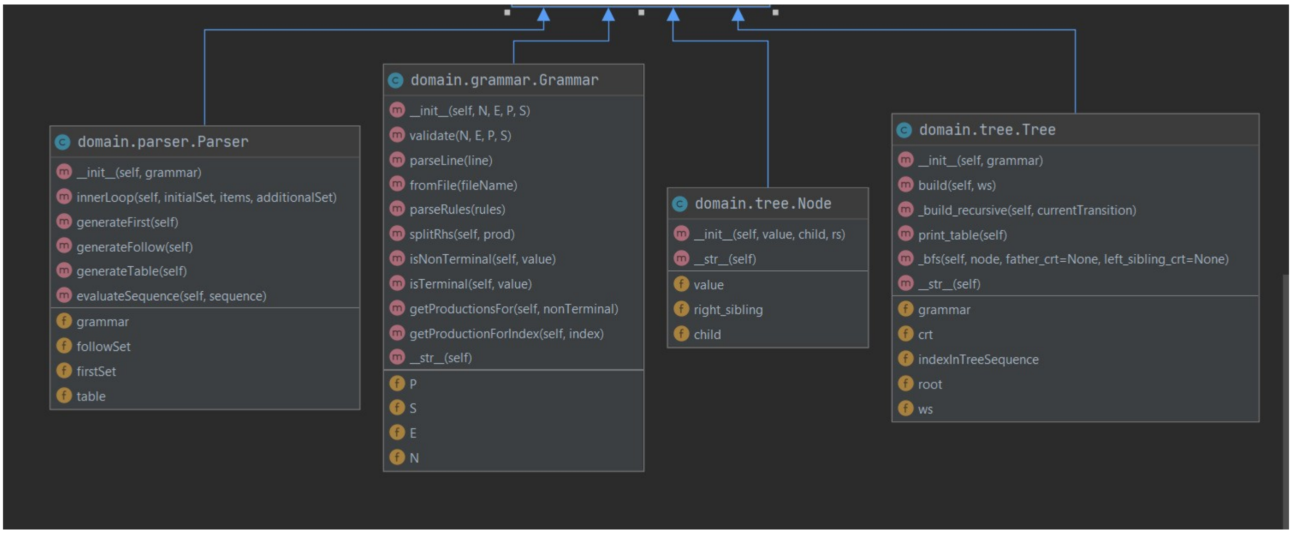
More complex example:
g2.txt:
N = { program, declaration, type, typeTemp, cmpdstmt, stmtlist, stmt, stmtTemp, simplstmt, structstmt, ifstmt, tempIf, forstmt, forheader, whilestmt, assignstmt, arithmetic1, arithmetic2, multiply1, multiply2, expression, IndexedIdentifier, iostmt, condition, relation }
E = { go, number, array, string, {, }, ;, +, -, *, /, (,), while, for, if, else, cin, cout, <<, >>, id, const, lt, lte, is, dif, gte, gt, eq }
S = program
P = {
program -> go cmpdstmt,
declaration -> type id,
type -> string | number typeTemp,
typeTemp -> E | array [const],
cmpdstmt -> { stmtlist },
stmtlist -> stmt stmtTemp,
stmtTemp -> E | stmtlist,
stmt -> simplstmt ; | structstmt,
simplstmt -> assignstmt | iostmt | declaration,
structstmt -> cmpdstmt | ifstmt | whilestmt | forstmt,
ifstmt -> if condition stmt tempIf,
tempIf -> E | else stmt,
forstmt -> for forheader stmt,
forheader -> (number assignstmt ; condition ; assignstmt),
whilestmt -> while condition stmt,
assignstmt -> id eq expression,
expression -> arithmetic2 arithmetic1,
arithmetic1 -> + arithmetic2 arithmetic1 | - arithmetic2 arithmetic1 | E,
arithmetic2 -> multiply2 multiply1,
multiply1 -> * multiply2 multiply1 | / multiply2 multiply1 | E,
multiply2 -> (expression) | id | const,
IndexedIdentifier -> id [const],
iostmt -> cin >> id | cout << id,
condition -> (id relation const),
relation -> lt | lte | is | dif | gte | gt
}

pif.txt:
go
{
number
id
;
id
eq
const
;
if
(
id

gt
const
)
cout
<<
id
;
}

Parsing output:
1 7 8 11 15 2 4 5 10 8 11 13 26 27 31 37 34 30 10 8 12 17 20 41 47 11 14 40 21 9

Solution:



The **Grammar** class has a field for each (N, E, P, S) set of the grammar, namely *terminals*, *non terminals*, *productions* and *a starting symbol*.
The set of productions P is kept as a list of tuples, of the type (startingSymbol, dest), both strings.
In the Grammar class, most of the methods are for file parsing, however getProductionsFor returns a list for all productions for the specific nonTerminal, for example *(S, aA), (S, Epsilon)*. Since we are implementing the LL(1) algorithm, we also implemented the first and follow algorithms.
The **first** algorithm builds a set for each non-terminal that contains all terminals from which we can start a sequence, starting from that given non-terminal.
The **follow** builds a set for each non-terminal basically returns the “first of what’s after”, namely all the non-terminals into which we can proceed from the given non-terminal.
Having these 2 sets built for each non-terminal (and for terminals also, but those are trivial), we proceed to build the **LL(1) parse table**. We follow the rules given in the lecture: we build a table that has as rows all non-terminals + terminals, and as rows, all terminals, plus the “$” sign in both rows and columns. We then follow the rules given in Ms. Motogna’s seventh lecture, slide 9.
Having the parse table built, the next step is **parsing** a given input sequence with

the LL(1) parsing algorithm, following the push/pop rules from slide 13 in same lecture 7. This will build an output which we subsequently **recursively build a tree** in a *depth first search* manner – we start from the root, and then we take care of its first child and then right sibling.
The last step is iterating through the tree in a *breadth first* manner and **printing** the obtained data.