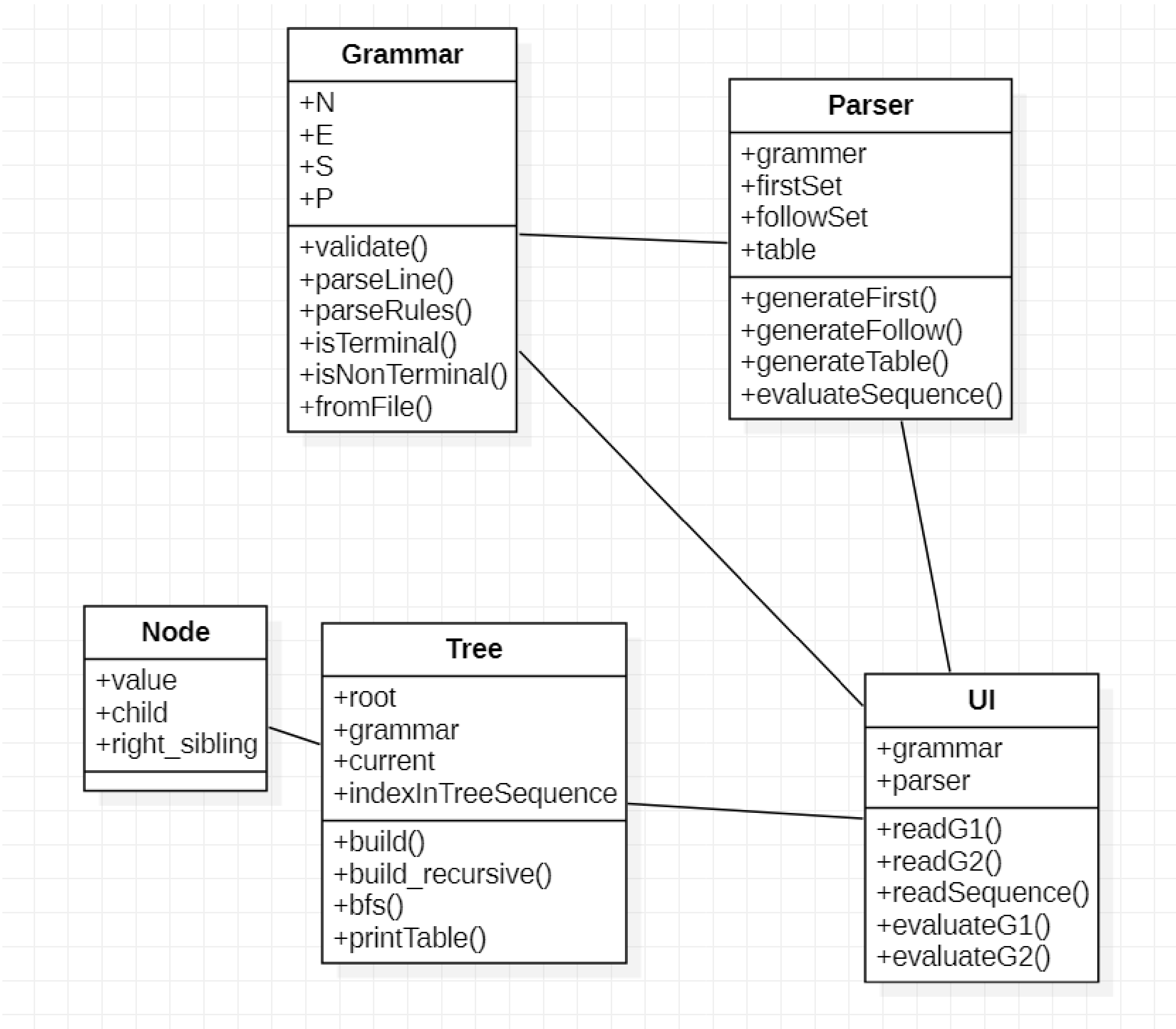
Parser Documentation (lab5-8)

Implement a parser using ll(1) parsing algoritm.Use a table as representation of the parsing tree.

Source code: https://github.com/IuliaPapureanu/FLCD



Example 1:

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
}
```

treeOutput1:

TREE

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

Example2

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
}
```

treeOutput2:

TREE

1 | program | None | None

2 | go | 1 | None

3 | cmpdstmt | 1 | 2

4 | { | 3 | None

5 | stmtlist | 3 | 4

6 | stmt | 5 | None

7 | stmtTemp | 5 | 6

8 | simplstmt | 6 | None

9 | ; | 6 | 8

10 | declaration | 8 | None

11 | type | 10 | None

12 | id | 10 | 11

13 | number | 11 | None

14 | typeTemp | 11 | 13

15 | E | 14 | None

16 | stmtlist | 7 | None

17 | stmt | 16 | None

18 | stmtTemp | 16 | 17

19 | simplstmt | 17 | None

20 | ; | 17 | 19

21 | assignstmt | 19 | None

22 | id | 21 | None

23 | eq | 21 | 22

24 | expression | 21 | 23

25 | arithmetic2 | 24 | None

26 | arithmetic1 | 24 | 25

27 | multiply2 | 25 | None

28 | multiply1 | 25 | 27

29 | const | 27 | None

30 | E | 28 | None

31 | E | 26 | None

32 | stmtlist | 18 | None

33 | stmt | 32 | None

34 | stmtTemp | 32 | 33

35 | structstmt | 33 | None

36 | ifstmt | 35 | None

37 | if | 36 | None

38 | condition | 36 | 37

39 | stmt | 36 | 38

40 | tempIf | 36 | 39

41 | (| 38 | None

42 | id | 38 | 41

43 | relation | 38 | 42

44 | const | 38 | 43

45 |) | 38 | 44

46 | gt | 43 | None

47 | simplstmt | 39 | None

48 | ; | 39 | 47

49 | iostmt | 47 | None

50 | cout | 49 | None

51 | << | 49 | 50

52 | id | 49 | 51

53 | E | 40 | None

54 | E | 34 | None

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 nonterminal.

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:

Rules LL(1) table

1. $M(A,a) = (\alpha, i), \forall a \in FIRST(\alpha), a \neq \epsilon, A \rightarrow \alpha$ production in P with number i
   $M(A,b) = (\alpha, i)$, if $\epsilon \in FIRST(\alpha), \forall b \in FOLLOW(A), A \rightarrow \alpha$ production in P with number i

2. $M(a,a) = pop, \forall a \in \Sigma$;
3. $M(\$,\$) = acc$;
4. M(x,a)=err (error) otherwise .

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.

Moves

1. Push – put in stack
   $(ux, A\alpha\$, \pi) \vdash (ux, \beta\alpha\$, \pi i)$, if $M(A,u) = (\beta, i)$;
   (pop A and push symbols of β)
2. Pop – take off from stack (from both stacks)
   $(ux, a\alpha\$, \pi) \vdash (x, \alpha\$, \pi)$, if M(a,u)=pop
3. Accept
   $(\$, \$, \pi) \vdash acc$
4. Error - otherwise

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