Lab 5-8

Jeler Andrei-Iulian & Hoha Mihai

934

Github link: https://github.com/SummerRolls99/FLCD/tree/main/lab%20-%20parser

Statement: Implement a parser algorithm

Lab 5

One of the following parsing methods will be chosen (assigned by teaching staff):

1.a. recursive descendent

The representation of the parsing tree (output) will be (decided by the team):

2.c. table (using father and sibling relation) (max grade = 10)

Lab 6

PART 2: Deliverables

1. Algorithm corresponding to parsing tables (if needed) and parsing strategy

2. Class ParserOutput - DS and operations corresponding to choice 2.c (required operations: transform parsing tree into representation; print DS to screen and to file)\

Lab 7

PART 3: **Deliverables**

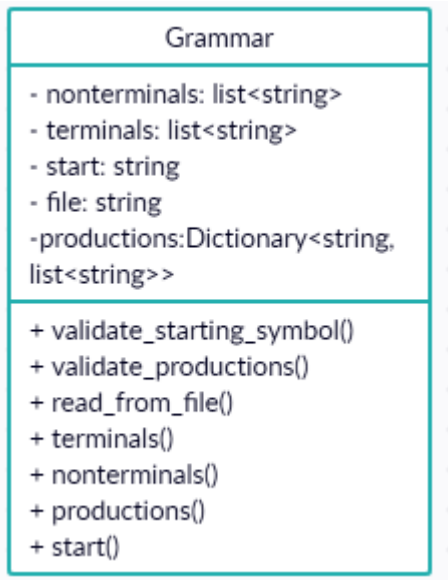
3. Source code
4. Run the program and generate: out1.txt (result of parsing if the input was g1.txt); out2.txt (result of parsing if the input was g2.txt)

Messages: if conflict exists; if syntax error exists (specify location if possible)

5. Code review

Lab 5 - Grammar

Class diagram:



Class structure:

The grammar class stores the necessary information as follows:

- The nonterminals are stored as a list of strings
- The terminals are stored as a list of strings
- The starting symbol is stored as a string
- The productions are stored as a dictionary that has as key the left hand side, and as value a list which has elements lists of string corresponding to each value in the right hand side

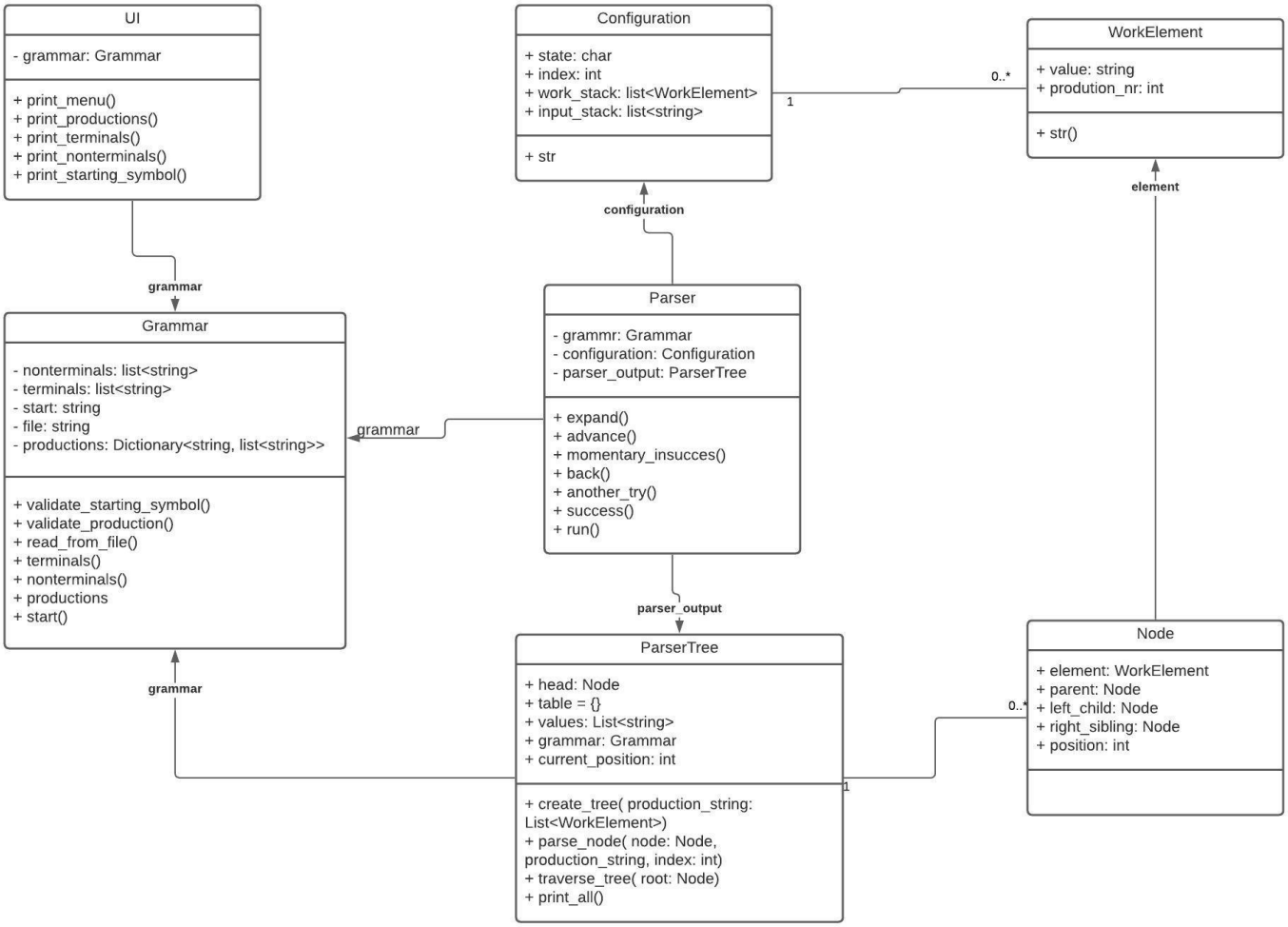
File structure:

The grammar is stored in the file as follows:

- First line: list of nonterminals
- Second line: list of terminals
- Third line: starting symbol
- Rest of file: the productions as follows: each line has a production, the lhs and rhs are separated by '->' and each possible value of the production is separated by '|'

Lab 6-8 Parser + ParserTree

Class diagram: (https://lucid.app/lucidchart/invitations/accept/4b299fa3-ac85-471b-a14f-76a585ffee9f) - link to the LucidChart document



Configuration:

(s, i, α, β) :

- s - the current state – represented as an enum
- i - the current index
- α - the working stack – represented as a list of WorkElements (contains a value and a production_nr in case it is a non_terminal)
- β - the input stack - represented as a list of strings

Parser:

Class structure:

The class contains the following attributes:

- Grammar: The grammar specific to the parser
- Configuration: The current configuration of the parsing
- Parser_output: The parserTree used for the output

Method:

- Run – wraps the descendent recursice algorithm , using the correct method corresponding to the move that is required. It stops when the current configuration is equivalent to an error or success.
- Specific methods for each move:
    - State is normal:
        - If the head of the input stack is a nonterminal: **EXPAND**
        - If the head of the input stack is a terminal and it is equal to the current symbol in the input: **ADVANCE**
        - If the head of the input stack is a terminal and it is equal to the current symbol in the input: **MOMENTARY INSUCCESS**
    - State is back:
        - If the head of the input stack is a nonterminal: **ANOTHER TRY**
        - If the head of the input stack is a terminal: **BACK**

Parser Output:

The ParserTree class stores the following information:

- Table: a dictionary which has as key the position, and as value the node itself
- Values: a list of strings corresponding to the values
- Grammar: the grammar of the parser

Method create_tree receives as parameter the list of productions string and constructs the table using parse_node recursively. Parse_node gets the first production string and constructs the rest of the tree extracting the corresponding production, adding the production as right siblings, and create another level for the nonterminals.

Output file structure:

The output file has the following structure:

- Starts with the corresponding tree
- Parent: list with the index of the parent of each node (or –1 if it the root)
- Left children: list with the index of the left child of each node (or –1 if it does not exist)
- Right siblings: list with the index of the right sibling of each node (or –1 if it does not exist)

Grammars (input files):

G1:

```
S
a b c
S
S -> a S b S | a S | c
```

G2:

Nonterminals:

```
	program_stmt array_values compound_stmt stmt_list stmt simple_stmt
complex_stmt IO_stmt write_expressions decl_stmt type primary_types
array_types number non_digit assign_stmt expression operator term factor
return_stmt if_stmt loop_stmt for_stmt while_stmt condition
relational_operator conditional_operator elif_stmt NZidentifier
NZEidentifier for_first
Terminals:
```

```
	+ - * / % = < > <= >= == != and or not { } [ ] ( ) ; , \n \t read
write if else elif while for integer string char bool true false program {
} return identifier constant
Start:
```

```
	program_stmt
Productions
```

```
	program_stmt -> program compound_stmt
compound_stmt -> { stmt_list }
stmt_list -> stmt | stmt stmt_list
stmt -> simple_stmt | complex_stmt
simple_stmt -> decl_stmt | assign_stmt ; | return_stmt ; | IO_stmt ;
complex_stmt -> if_stmt | loop_stmt
IO_stmt -> read ( identifier ) | write ( expression write_expressions
write_expressions -> , expression write_expressions | )
decl_stmt -> type identifier NZidentifier | type identifier = expression
NZEidentifier | type identifier = { constant array_values
array_values -> , constant array_values | } ;
NZidentifier -> , identifier NZidentifier | ;
NZEidentifier -> , identifier = expression NZEidentifier | ;
type -> primary_types | array_types
```

```
primary_types -> integer | char | string | bool
array_types -> primary_types [ constant ]
assign_stmt -> identifier = expression
expression -> term operator expression | term
operator -> + | -
term -> factor * term | factor / term | factor
factor -> ( expression ) | identifier | identifier [ expression ] |
constant
return_stmt -> return expression
if_stmt -> if ( condition ) compound_stmt | if ( condition ) compound_stmt
elif_stmt
elif_stmt -> elif ( condition ) compound_stmt elif_stmt | elif ( condition
) compound_stmt | else compound_stmt
loop_stmt -> for_stmt | while_stmt
for_stmt -> for ( for_first condition ; assign_stmt ) compound_stmt | for
( for_first condition ) compound_stmt
for_first -> decl_stmt | assign_stmt ;
while_stmt -> while ( condition ) compound_stmt
condition -> expression relational_operator expression
conditional_operator condition | not expression relational_operator
expression conditional_operator condition | expression relational_operator
expression | not expression relational_operator expression
relational_operator -> > | < | >= | <= | == | !=
conditional_operator -> and | or
```

Output for grammar G1 with input: aacbc

```
S#1
  |--a
  |--S#2
  |   |--a
  |   \--S#3
  |     \--c
  |--b
  \--S#3
      \--c

Fathers: [-1, 0, 0, 2, 2, 4, 0, 0, 7]
Left children: [1, -1, 3, -1, 5, -1, -1, 8, -1]
Right siblings = [-1, 2, 6, 4, -1, -1, 7, -1, -1]
```