Recursive Descendant Parser

Giurgiu Matei-Alexandru + Andrei Gabor

Github link: https://github.com/giurgiumatei/Formal-Languages-and-Compilers-Design/tree/main/Lab12%20Parser%20Full

- ## Statement:

Implement a parser using the recursive descendent algorithm.

- ## Simple example:

G1.txt:

```
S
a b c
S
S a$S$b$S
S a$S
S c$
```

Productions are split by the symbol “$”.

The productions for non-terminal S will be:

S -> a S b S

S -> a S

S -> c

- ## Grammar class:

Properties:

1. Filename -> string;
2. Grammar -> list of lists (representation of objects);
3. N -> list of non terminals;
4. E -> list of terminals;
5. S -> starting symbol (as a list of one string);
6. P -> productions kept as a dictionary with a string as key and value a list, that contains lists of symbols in the right hand side of the production;

Methods:

1. read_grammar() -> will read grammar from text file;
2. represent_productions() -> will construct the P dictionary;
3. get_non_terminals();
4. get_terminals();
5. get_start_symbol();
6. get_productions();
7. get_productions_for_non_terminal();
8. print_productions_for_non_terminal();

- Example on grammar:

G2.txt:

```
program dec_list declaration id_list type_1 vector_decl type stmt_list stmt simple_stmt
assignmnt expression operation term elem_vector io_stmt struct_stmt while_stmt if_stmt
for_stmt condition relation unary_operation
29 30 24 0 1 25 47 48 43 49 45 19 20 6 2 3 4 5 14 15 13 41 42 39 23 32 33 31 34 35 36
38 11 12 7 9 10 8 40 16 50
program
program 29$dec_list$stmt_list$30
dec_list declaration$24
dec_list declaration$24$dec_list
declaration type$id_list
id_list 0
id_list 0$25$id_list
type_1 47
type_1 48
type_1 43
type_1 49
vector_decl type_1$45$19$1$20
type type_1
type vector_decl
stmt_list stmt
stmt_list stmt$stmt_list
stmt simple_stmt$24
stmt struct_stmt
simple_stmt assignmnt
simple_stmt io_stmt
assignmnt 0$6$expression
assignmnt elem_vector$6$expression
expression term
expression term$operation$expression
expression unary_operation$expression
unary_operation 16
unary_operation 50
operation 2
operation 3
operation 4
operation 5
operation 14
operation 15
```

```
operation 13
term 0
term 1
term elem_vector
elem_vector 0$19$1$20
elem_vector 0$19$0$20
io_stmt 41$id_list
io_stmt 42$id_list
struct_stmt while_stmt
struct_stmt if_stmt
struct_stmt for_stmt
while_stmt 39$condition$23$stmt_list$40
if_stmt 31$condition$23@stmt_list$33
if_stmt 31$condition$23$stmt_list$32$23$stmt_list$33
for_stmt 34$0$35$1$36$1$23$stmt_list$38
for_stmt 34$0$35$1$36$0$23$stmt_list$38
condition expression$relation$expression
relation 11
relation 12
relation 7
relation 9
relation 10
relation 8
```

PIF.out:

```
token|ST_pos
29 | -1
43 | -1
0 | 0
25 | -1
0 | 1
25 | -1
0 | 2
24 | -1
43 | -1
45 | -1
19 | -1
1 | 3
20 | -1
0 | 4
24 | -1
41 | -1
0 | 0
24 | -1
0 | 2
6 | -1
1 | 5
24 | -1
34 | -1
0 | 6
35 | -1
1 | 7
36 | -1
0 | 0
23 | -1
41 | -1
0 | 1
24 | -1
0 | 4
19 | -1
0 | 6
20 | -1
6 | -1
0 | 1
```

```
24 | -1
0 | 2
6 | -1
0 | 2
2 | -1
0 | 1
24 | -1
38 | -1
42 | -1
0 | 2
24 | -1
30 | -1
```

- Parser class:

Properties:

1. grammar -> Grammar;
2. sequence -> list of codes;
3. out_file -> string;
4. working_stack -> list acting as a stack;
5. input stack -> list acting as a stack;
6. state -> string;
7. index -> int;
8. tree -> list;

Methods:

1. read_sequence() -> will build the list of codes from the file;
2. get_situation() -> will append to the file the current state, index, the content of the working stack and the content of the input stack;
3. init_output_file() -> will create the output file;
4. write_in_output_file() -> will append a message in the output file;
5. expand();
6. advance();
7. momentary_insuccess();
8. back();
9. success();
10. another_try();

11. print_working() -> will print the working stack and append it to the output file;
12. run() -> the main function, will check if the sequence is accepted;
13. create_parsing_tree();
14. get_length_depth();
15. write_parsing_tree();

- Algorithm:

We have an initial configuration and we define some moves to get to the final configuration. A configuration is of the model (s, i, α, β) where s is the state of the parsing, i is the position of current symbol in the input sequence, α is the working stack that stores the way the parse is built and β is the input stack, that is part of the tree to be built.

The state s can be:

1. q -> normal state;
2. b -> back state;
3. f -> finals state that signifies success;
4. e -> error state that signifies insuccess;

The moves can be:

1. Expand -> when the head of input stack is nonterminal. We pop the nonterminal from the input stack, then we put it to the top of the working stack and then we put to the top of the input stack a new production for the nonterminal.
2. Advance -> when the head of input stack is a terminal = current symbol from input, we pop the terminal from the input stack and put it at the top of the working stack. After that we increment the index.
3. Momentary insuccess -> when the head of the input stack is a terminal != current symbol from input. We set the current state to the back state “b”.
4. Back -> when the head of working stack is a terminal. We pop the terminal from the working stack and put it at the top of the input stack. After that we decrement the index.

5. Another try -> when the head of the working stack is a nonterminal. We pop the nonterminal from the working stack and then we have three cases. We have to keep in mind that an element in the working stack is a tuple of the form (nonterminal, production number).
   a) If the production number + 1 is smaller than the number of productions for the nonterminal, we set the state to “q”, we construct the new tuple that consists of the nonterminal and the new production number and put it on the top of the stack. After that, we delete the production at the end of the input stack and we put the new production at the top of the input stack.
   b) If the index is 1 and the nonterminal is the starting symbol, we set the state to “e”;
   c) Otherwise we delete the production at the end of the input stack and we put the nonterminal at the top of the input stack.
6. Success -> we set the state to “f”.

- Parsing Tree:

The parsing tree is represented as a table of the form (index, info, parent, left_sibling).

Example on G2:

```
Parsing tree:
index info parent  left_sibling
0  program  -1  -1
1  29  0  2
2  dec_list  -1  14
3  declaration  2  9
4  type  3  7
5  type_1  4  -1
6  43  5  -1
7  id_list  -1  -1
8  0  7  9
9  25  -1  10
10  id_list  -1  -1
11  0  10  12
12  25  -1  13
13  id_list  -1  -1
14  0  13  15
15  24  -1  -1
16  dec_list  -1  -1
17  declaration  16  33
18  type  17  26
19  vector_decl  18  -1
20  type_1  19  22
21  43  20  -1
22  45  -1  23
23  19  -1  24
```

```
24  1  -1  25
25  20  -1  -1
26  id_list  -1  -1
27  0  26  -1
28  24  -1  -1
29  stmt_list  -1  -1
30  stmt  29  40
31  simple_stmt  30  36
32  io_stmt  31  -1
33  41  32  34
34  id_list  -1  -1
35  0  34  -1
36  24  -1  -1
37  stmt_list  -1  -1
38  stmt  37  52
39  simple_stmt  38  46
40  assignmnt  39  -1
41  0  40  42
42  6  -1  43
43  expression  -1  -1
44  term  43  -1
45  1  44  -1
46  24  -1  -1
47  stmt_list  -1  -1
48  stmt  47  84
49  struct_stmt  48  -1
50  for_stmt  49  -1
51  34  50  52
52  0  -1  53
53  35  -1  54
54  1  -1  55
55  36  -1  56
56  0  -1  57
57  23  -1  58
58  stmt_list  -1  74
59  stmt  58  69
60  simple_stmt  59  65
61  io_stmt  60  -1
62  41  61  63
63  id_list  -1  -1
64  0  63  -1
65  24  -1  -1
66  stmt_list  -1  -1
67  stmt  66  85
68  simple_stmt  67  77
69  assignmnt  68  -1
70  elem_vector  69  75
71  0  70  72
72  19  -1  73
73  0  -1  74
74  20  -1  -1
75  6  -1  76
76  expression  -1  -1
77  term  76  -1
78  0  77  -1
79  24  -1  -1
80  stmt_list  -1  -1
81  stmt  80  -1
82  simple_stmt  81  92
83  assignmnt  82  -1
84  0  83  85
85  6  -1  86
86  expression  -1  -1
87  term  86  89
88  0  87  -1
89  operation  -1  91
90  2  89  -1
```

```
91  expression  -1  -1
92  term  91  -1
93  0  92  -1
94  24  -1  -1
95  38  -1  -1
96  stmt_list  -1  -1
97  stmt  96  -1
98  simple_stmt  97  103
99  io_stmt  98  -1
100  42  99  101
101  id_list  -1  -1
102  0  101  -1
103  24  -1  -1
104  30  -1  -1
```