Parser II (1)

Hostiuc Robert & Aldea Denise

Main:

The main will take every functions from the project and will merge all of them for the output. This module is responsible for all the combinations like g1.txt with seq.txt (the sequence can also be written by hand) or g2.txt with PIF.tft (P1.txt or P1err.txt).

Grammar:

In grammar we have the read function that will take all the elements from the local file. Beside that we have print functions for every thing (nonterminal, terminal, start, productions). And for everything we also have a get to be able to work with them.

Parser:

This module will have all the main requirements (first, follow, parsertable) with the implementations.
FIRST is a function on each non-terminal (E, E', T, T', and F) that tells us which terminals (tokens from the lexer) can appear as the first part of one of these non-terminals. Epsilon (neither a terminal nor a non-terminal) may also be included in this **FIRST** function. (**FIRST** is also defined for terminals, but its value is just equal to the terminal itself, so we won't talk about it more.) What this means is that the parser is going to invoke some of these productions in the grammar. We need to know which one to pick when we see a particular token in the input stream.
FOLLOW. Just as **FIRST** shows us the terminals that can be at the beginning of a derived non-terminal, **FOLLOW** shows us the terminals that can come *after* a derived non-terminal. Note, this does *not* mean the last terminal derived from a non-terminal. It's the set of terminals that can come after it. We define **FOLLOW** for all the non-terminals in the grammar.Instead of looking at the first terminal for each phrase on the right side of the arrow, we find every place our non-terminal is located on the right side of *any* of the arrows. Then we look for some terminals. As we go through our example, you'll see almost all of the different ways we figure out the **FOLLOW** of a non-terminal.

Talking about the table, we will output all the combinations (index, column) with the respectiv value (represented as a tuple). We will use this table to gen the Parser sequence. For this we will take all the terminals from the initial sequence (saved in alpha) and the start nonterminal (saved in beta) after we will create a tuple (beta.peek(), alpha.peek()) peek - representing the top of the stack. In beta we will pop the top of the stack and add the product of it, for exemple if we have: S -> A B (S being the start) with the sequence: int, we will start with the tuple (S, int), and alpha will remain int. but beta will become B A instead of S, and so on.

The result of the parser will be save in out1.txt (if g1.txt is the input) and out2.txt (if g2.txt is the input). This result is made in the PaserOutput module.

As exemple:

g1:

```
N = { S A B C }

E = { ( ) + * int }
```

```
S = S

P = {

    S -> A B

    A -> ( S ) | int C

    B -> + S | epsilon

    C -> * A | epsilon

}

seq.txt:


( int ) + int


parserTable:


(*,)) -> (err,-1)

(*,*) -> (pop,-1)

(C,$) -> (epsilon,6)

(*,+) -> (err,-1)

(C,() -> (err,-1)

(C,)) -> (epsilon,6)

(C,*) -> (* A,7)

(C,+) -> (epsilon,6)

((,int) -> (err,-1)

($,int) -> (err,-1)

(A,int) -> (int C,1)

(int,int) -> (pop,-1)

(+,$) -> (err,-1)

(+,() -> (err,-1)
(+,)) -> (err,-1)
(+,*) -> (err,-1)
(+,+) -> (pop,-1)
(),int) -> (err,-1)
(B,int) -> (err,-1)
(S,int) -> (A B,5)
((,$) -> (err,-1)
((,() -> (pop,-1)
($,$) -> (acc,-1)
((,)) -> (err,-1)
((,*) -> (err,-1)
(A,$) -> (err,-1)
((,+) -> (err,-1)
($,() -> (err,-1)
($,)) -> (err,-1)
(*,int) -> (err,-1)
($,*) -> (err,-1)
(A,() -> (( S ),2)
($,+) -> (err,-1)
(A,)) -> (err,-1)
(A,*) -> (err,-1)
(A,+) -> (err,-1)
(C,int) -> (err,-1)
(),$) -> (err,-1)
(),() -> (err,-1)
(),)) -> (pop,-1)
(),*) -> (err,-1)
```

```
(B,$) -> (epsilon,3)

(),+) -> (err,-1)

(+,int) -> (err,-1)

(B,() -> (err,-1)

(int,$) -> (err,-1)

(B,)) -> (epsilon,3)

(B,*) -> (err,-1)

(B,+) -> (+ S,4)

(int,)) -> (err,-1)

(int,() -> (err,-1)

(int,+) -> (err,-1)

(int,*) -> (err,-1)

(S,$) -> (err,-1)

(S,() -> (A B,5)

(S,)) -> (err,-1)

(S,*) -> (err,-1)

(S,+) -> (err,-1)

(*,$) -> (err,-1)

(*,() -> (err,-1)


Parse Sequence:


[5, 2, 5, 1, 6, 3, 4, 5, 1, 6, 3]


out1.txt:


Index | Value | Parent | Sibling
```

```
1 | S | 0 | 0

2 | A | 1 | 0

3 | B | 1 | 2

4 | ( | 2 | 0

5 | S | 2 | 4

6 | ) | 2 | 5

7 | A | 5 | 0

8 | B | 5 | 7

9 | int | 7 | 0

10 | C | 7 | 9

11 | epsilon | 10 | 0

12 | epsilon | 8 | 0

13 | + | 3 | 0

14 | S | 3 | 13

15 | A | 14 | 0

16 | B | 14 | 15

17 | int | 15 | 0

18 | C | 15 | 17

19 | epsilon | 18 | 0

20 | epsilon | 16 | 0
```