Compiler Construction Assignment 2 Report Nicolas Smith 11463982

Part 1. Preliminaries:

Given grammar:

S ::= Exp eof
Exp ::= Term | Exp + Term | Exp - Term
Term ::= Factor | Term * Factor |
Term / Factor
Factor ::= id | num | Fncall | Aref | (Exp) |
Fncall ::= id (Arguments)
Aref ::- id [Indices]
Arguments ::= Exp | Exp , Arguments
Indices ::= Exp | Exp , Indices

Grammar Sans Left Recursion:

S ::= Exp eof
Exp::= TermE'
E'::=+TermE' | -TermE' | ε
Term::= FactorT'
T'::= *FactorT' | /FactorT' | ε
Factor ::= id | num | Fncall | Aref | (Exp) |
Fncall ::= id (Arguments)
Aref ::- id [Indices]
Arguments ::= Exp | Exp , Arguments
Indices ::= Exp | Exp , Indices

Grammar Table:

Terminals:

0 = eof
1 = id
2 = num
3 = +
4 = -
5 = *
6 = /
7 = (
8 =)
9 = [
10 =]
11 = ,
12= ε

Non-terminals:

-1 = S
-2 = Expr
-3 = E'
-4= Term
-5= T'
-6=Factor
-7=FnCall
-8=Aref
-9=Arguments
-10=Indices

Part 2: Recursive Descent Parser

The recursive descent parse class has a Parse String() method which takes in a string of characters representing an expression from the grammar, for example the input "TOM eof" will return TOM eof: SUCCESS. The method tokenises the string into a series of integers (in this case 1 0 0 0) and takes every second one as input to the token stream.

A tokenizer class is used to do this, it turns a string into a dual array of tokens, but since only the first column (i.e. the symbol values of tokens) are used I have also implemented a method that returns only the first column as an array for ease of access in the Parser. But as the assignment requires testing input in token stream form <token> <value> the tokenizer also has a method createInputTokens() that takes in a String of tokens and turns them into an array of ints of type <token> to be input in the parser.

```
Inputs and Results:
1 0 SUCCESS Correct Choices=6 Wrong Choices=2
2 0 SUCCESS Correct Choices=6 Wrong Choices=2
1 3 1 5 1 0 SUCCESS Correct Choices=12 Wrong Choices=6
2 4 2 4 2 0 SUCCESS Correct Choices=14 Wrong Choices=6
1 9 2 11 2 10 0 FAILURE Correct Choices=11 Wrong Choices=5
1 9 1 9 7 2 3 2 8 5 2 11 1 10 11 1 10 0 FAILURE Correct Choices=28 Wrong
Choices=13
1 7 7 1 9 2 10 5 1 9 2 10 8 3 1 9 2 10 5 1 9 2 10 8 0 SUCCESS Correct Choices=54
Wrong Choices=14

Input and Results Modified Grammar
1 0 SUCCESS Correct Choices=7 Wrong Choices=6
2 0 SUCCESS Correct Choices=6 Wrong Choices=5
1 3 1 5 1 0 SUCCESS Correct Choices=15 Wrong Choices=12
2 4 2 4 2 0 SUCCESS Correct Choices=14 Wrong Choices=13
1 9 2 11 2 10 0 SUCCESS Correct Choices=21 Wrong Choices=16
1 9 1 9 7 2 3 2 8 5 2 11 1 10 11 1 10 0 SUCCESS Correct Choices=49 Wrong
Choices=39
1 7 7 1 9 2 10 5 1 9 2 10 8 3 1 9 2 10 5 1 9 2 10 8 0 SUCCESS Correct Choices=59
Wrong Choices=45
```

As we see above the RDP Parser works correctly for one Grammar but not the other, I feel that this is due to a small oversight in the Grammar which I was unable to spot. N

Part 3: Make Grammar LL

LL(1) Grammar after Left factoring:

S ::= Exp eof
Exp::= TermE'
E'::=+TermE' | -TermE' | ε
Term::= FactorT'
T'::= *Factor | /Factor | ε
Factor ::= id F' | num | (Exp) |
F'::= (Arguments) | [Indices] | ε
Arguments ::= ExpA'
A'::= ,Arguments | ε
Indices ::= ExpI'
I'::= ,Indices | ε

This Grammar includes the addition of some new nonterminals:
-12= F'

-13= A'
-14= I'

Notice how we have gotten rid of FnCall and Aref because they were not disjoint from factor, instead we encapsulated the rest of their productions in a new Production called F' Arguments and Indices both contained Exp on the left of the RHS' so this was also factored out with new productions A' and I'.

To use this grammar instead of the original in my program you call the Constructor of the grammar class modally, i.e. pass a value into the constructor, in this case 1.

Part 4: FIRST and FOLLOW sets.

Creating FIRST and FOLLOW sets is an important building block for this assignment as they are required to build the parsing table the second parser hinges on. The FIRST sets were relatively manageable to create but I must admit the FOLLOW sets had me quite befuddled for a while, though with the help of an online tool to check my outputs were correct and an unholy amount of time spent debugging, my routines eventually output the correct sets, below are the First and Follow sets for my modified LL Grammar

.

FIRST( ) sets generated by my routine:

```
First(-1):[1, 2, 7]
First(-2):[1, 2, 7]
First(-3):[3, 4, 12]
First(-4):[1, 2, 7]
First(-5):[5, 6, 12]
First(-6):[1, 2, 7]
First(-9):[1, 2, 7]
First(-10):[1, 2, 7]
First(-12):[7, 9, 12]
First(-13):[11, 12]
First(-14):[11, 12]
```

FOLLOW( ) sets generated by my routine:

```
-1: [20]
-2: [0, 8, 10, 11]
-3: [0, 8, 10, 11]
-4: [0, 3, 4, 8, 10, 11]
-5: [0, 3, 4, 8, 10, 11]
-6: [0, 3, 4, 5, 6, 8, 10, 11]
-9: [8]
-10: [10]
-12: [0, 3, 4, 5, 6, 8, 10, 11]
-13: [8]
-14: [10]
```

Part 6: Parsing Table.

The parsing table is a 2D array of type production indexed by nonterminals and terminals in the next position of the input stream. The location in Table[n][t] contains the production that can occur when we are dealing with the nonterminal n and t is the next inut stream token. If the parser reaches an empty table we know thee input is not a valid sentence.

Rough text representation of parsing table (no top row index because of different row sizes):

```
-1:  null   -2::= -4 -3  -2::= -4 -3  null null null null -2::= -4 -3  null null null null null
-2:  -3::= 12  null null -3::= 3 -4 -3  -3::= 4 -4 -3  null null null -3::= 12  null -3::= 12  -3::=
       12  -3::= 12
-3:  null -4::= -6 -5  -4::= -6 -5  null null null null -4::= -6 -5  null null null null null
```

```
-4:  -5::= 12  null null -5::= 12  -5::= 12  -5::= 5 -6 -5  -5::= 6 -6 -5  null -5::= 12  null -5::=
       12  -5::= 12  -5::= 12
-5  null -6::= 1 -12  -6::= 2  null null null null -6::= 7 -2 8  null null null null null
-6  null null null null null null null null null null null null null
-7  null null null null null null null null null null null null null
-8  null -9::= -2 -13  -9::= -2 -13  null null null null -9::= -2 -13  null null null null null
-9  null -10::= -2 -14  -10::= -2 -14  null null null null -10::= -2 -14  null null null null null
-10  null null null null null null null null null null null null null
-11  -12::= 12  null null -12::= 12  -12::= 12  -12::= 12  -12::= 12  -12::= 7 -9 8  -12::= 12
-12::= 9 -10 10  -12::= 12  -12::= 12  -12::= 12
-12  null null null null null null null null -13::= 12  null null -13::= 11 -9  -13::= 12
-13  null null null null null null null null null null -14::= 12  -14::= 11 -10  -14::= 12
```

Part 6 Predictve Parser:

Unlike the Recursive Descent parser the predictive parser need only return one value to indicate whether the parser was successful or not, thus I implemented the predictive parser to simply return a Boolean value.

A Predictive Parser should not make any incorrect choices unless the parsing table is incorrect or the sentence is invalid by the rules of the grammar, in which case it will terminate with failure upon detection. If we use the Recursive Descent Parser on the same Grammar as the predictive parser we will have the same amount of correct choices made by the RDP Parser as we will have total choices by our Predictive Parser.

```
Input and Results (number after boolean is # of choices).
1 0 true 7
2 0 true 6
1 3 1 5 1 0 true 15
2 4 2 4 2 0 true 14
1 9 2 11 2 10 0 true 21
1 9 1 9 7 2 3 2 8 5 2 11 1 10 11 1 10 0 true 49
1 7 7 1 9 2 10 5 1 9 2 10 8 3 1 9 2 10 5 1 9 2 10 8 0 true 59
```

To illustrate how it works I will display the state of the Stack for the first example:

```
[20, -1]
[20, 0, -2]
[20, 0, -3, -4]
[20, 0, -3, -5, -6]
[20, 0, -3, -5, -12, 1]
[20, 0, -3, -5, -12]
[20, 0, -3, -5, 12]
[20, 0, -3, -5]
[20, 0, -3, 12]
[20, 0, -3]
[20, 0, 12]
[20, 0]
true 7
```