

First Name: My Anh – Last
Name: Huynh

Class: 4308/section 02.

Professor: Sharon Perry

Project 2nd Deliverable – Parser

Due date: Oct 31, 2021

Status: Project deliverable is
100% complete and working as
designed. I am exercising P2
push to Oct 31st for free.

1. Scanner: Lexical analyzer

Lexeme	Token
function	FUNCTION
s	IDENTIFIER
(LEFT_PARENTHESIS
)	RIGHT_PARENTHESIS
,	COMMA
return	IDENTIFIER
a	IDENTIFIER
+	ADD_OPERATOR
b	IDENTIFIER
end	END

The solution for the phase one is a demo code that take in put from a file and have a function to splits the tokens in the file. The file may contain alphabets, letters, numbers, mathematical symbols, et cetera.

Input file:

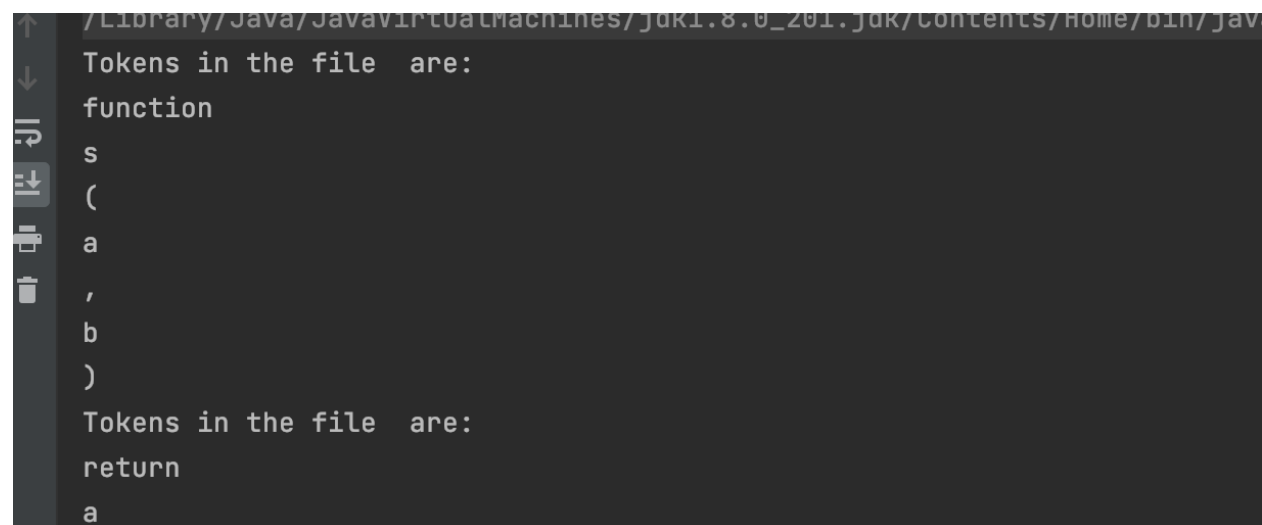
```
function s(a,b)
return a+b
end
```

Source code:

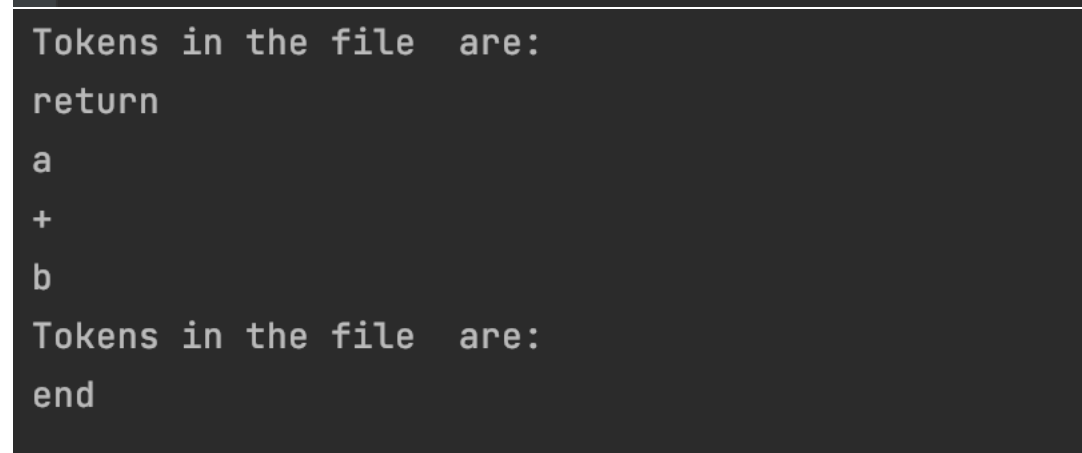
```
/*
 * Class:      CS 4308 Section n
 * Term:       Fall 2021
 * Name:       <My Anh Huynh>
 * Instructor: Sharon Perry
 * Project:    Deliverable 1 Scanner
 */
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.File;
import java.util.StringTokenizer;
public class ScannerPhase {
    public static void main(String[] args) throws FileNotFoundException {
        File file = new File("/Users/anhhuynh/Desktop/input.txt");// input
file of minimal form of Julia language
        Scanner sc = new Scanner(file); // Scanner object named as token is
initialized
    }
```

```
while (sc.hasNextLine()) {  
    String para = sc.nextLine();           // String is obtained in  
string format  
  
    String[] tokens = para.split(" ");     // split method in java  
is used to find the token in the file  
  
    System.out.println("Tokens in the file are: ");  
  
    for (String token : tokens) {         // for loop will run till  
the end of the file  
  
        System.out.println(token);       // tokens are printed  
  
    }  
}  
}
```

Ouput:



```
Tokens in the file are:  
function  
(  
a  
,  
b  
)  
Tokens in the file are:  
return  
a
```



```
Tokens in the file are:  
return  
a  
+  
b  
Tokens in the file are:  
end
```

2. Parser:

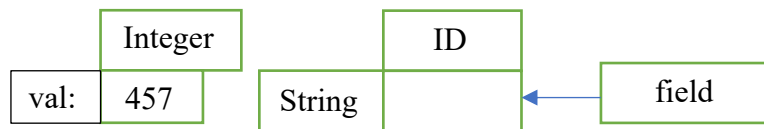
Before we can parse the input we need to break it up into a sequence of tokens. This is known as the lexical analysis part of the parser and we have a “LEXER” or “TOKENIZER” that perform this tokenization of the input. So it will read the input and break it into tokens.

For example:

- **CODE:** function scanToken();

This function will scan the input and sets nextToken to point to the newly scanned token.

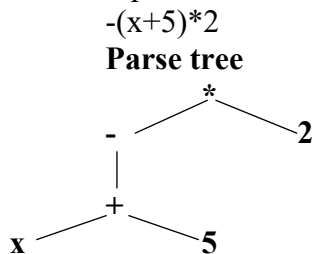
- **VARIABLE:** nextToken. I am assuming that we are using an object-oriented programming language. For example, we represent tokens with objects. Here is an object of class integer and object of class identifier.



The class integer has a single field value and the class identifier has a single field called string which points to the actual characters that are involved in the identifier.

The goal of our parser is to read in some expression and produce some output.

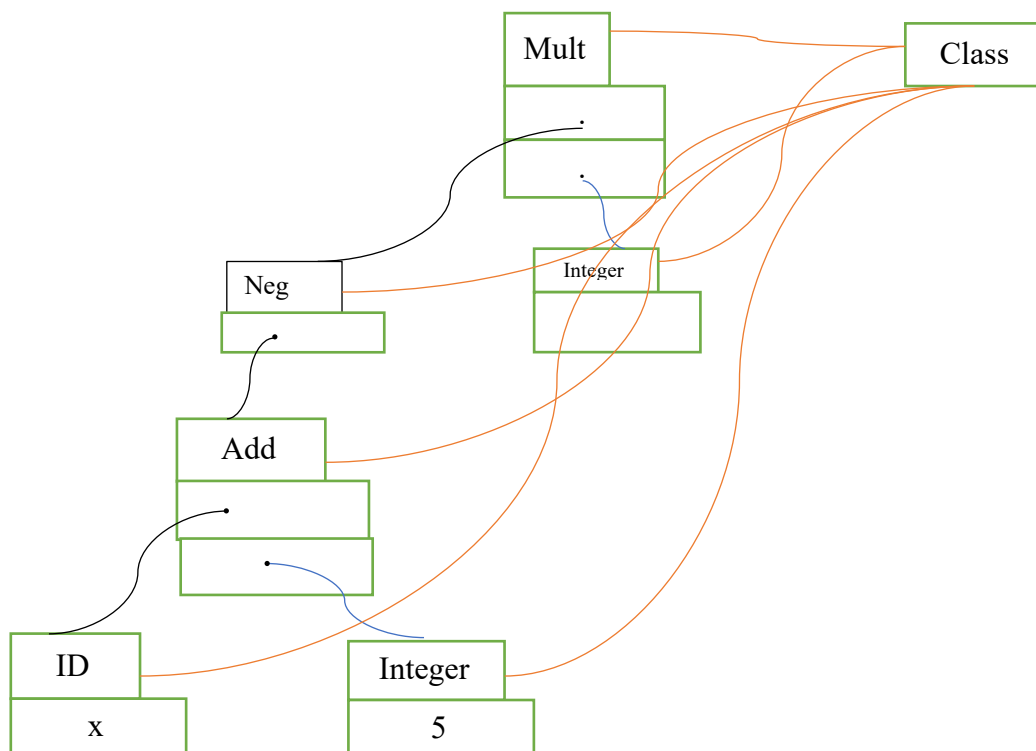
For example:



we could add **x** and **5** first and then **negate** that and we multiply that with **2**.

So this tree reflects the meaning of the expression.

Furthermore, we represent the tree with a collection of objects such as Mult, Integer, Neg, Add, ID, Integer.



○ **CLASSES**

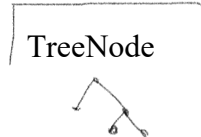
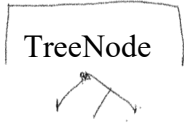
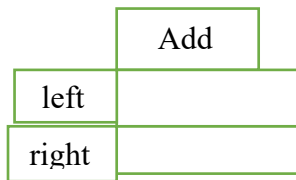
TreeNode ← Superclass

Add
Subtract
Mult
Div
Negate
ID(identifier)
Integer

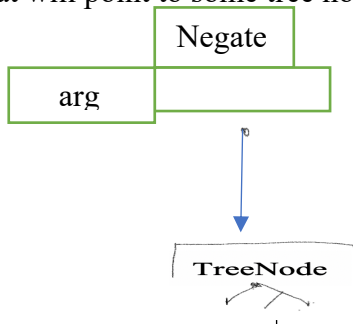
Subclasses

These are all different kinds of tree nodes.

The INFIX OPERATOR like **add** and **multiply**, we have two fields in the class called left and right. These will contain pointers to other tree nodes depending what kind of sub-expressions are being added together. That will determine exactly is down here



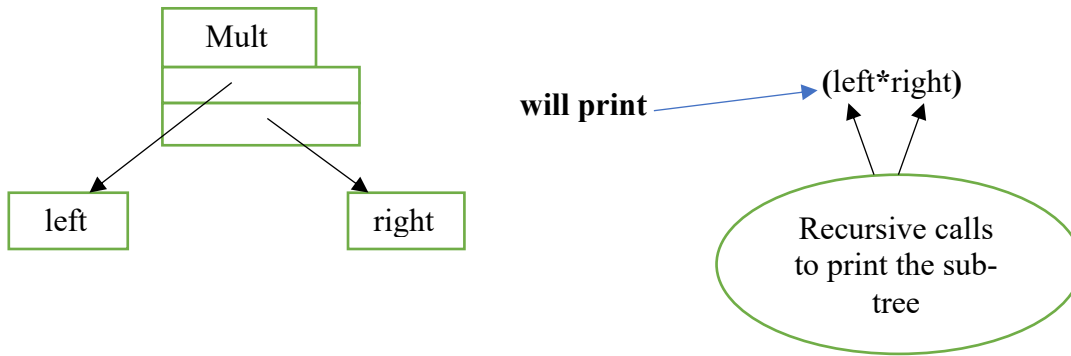
The PREFIX UNARY OPERATORS: so the Negate has a single field called arg(argument) and that will point to some tree node.



PRINT A TREE

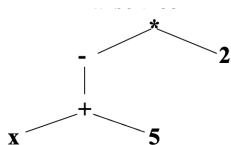
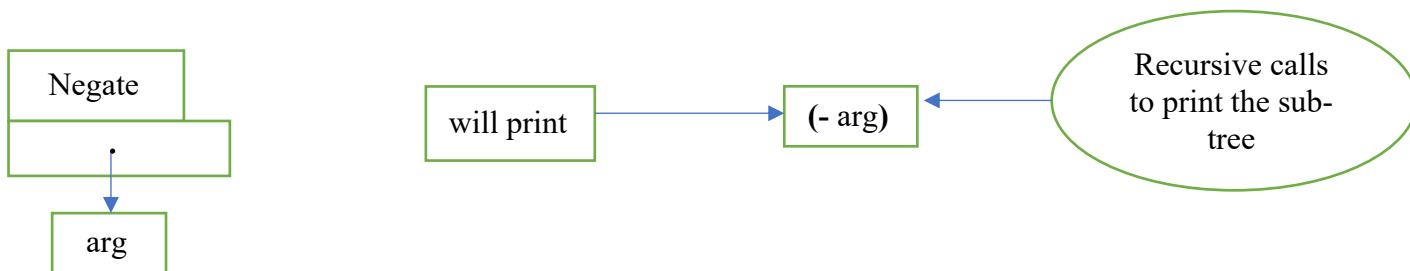
So to verify that the parser is working correctly by printing out the internal representation that the parser is producing.

Each class implements a **print** method.



For example, we have a print method for each one of our classes, and these are going to be recursive methods that call each other. The class Multiply or any other infix operator would work like getting a pointer to the left sub-tree and a pointer to the right subtree. What print is going to do is going to do when called on this particular node is print an opening parenthesis then it is going to call itself recursively to print out whatever is pointer to by the left pointer then an address and then call itself again recursively to print out whatever is pointer to by the right of our field and finally a closing parenthesis.

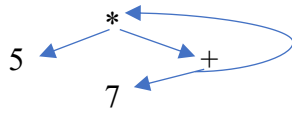
A negate class has a single field points to its argument and it will print an opening parenthesis followed by the negate symbol followed by the subtree and a closing parenthesis.



so this particular expression will print out $((-(x+5))*2)$

We could see that it has got several parentheses but we should make sure the output is not unambiguous.

There may an issue that could get a cycle in the tree like when trying to print this out, we could run into an infinitive loop.



Cycle in Tree
_Bug

5*(7+(5*(7+(...

- Evaluating the expression:
Each class implement an **eval** method and returns the value of expression.
For example, in the Add.eval method
return left.val() + right.eval()
in the Negate.eval method return -(arg.eval())
in the Integer.val method return val
in the ID.eval method, the method will figure out the current value of the variable and return it.
- Parsing Algorithm:
Functions:
parseE(): will parse an expression and return a pointer to the tree that expression.
parseT(): will parse a term
parseF(): will parse a factor
one for each non-terminal grammar symbol, each of these functions will scan a bunch of tokens and return a pointer to the tree it builds to represent whatever it parsed.
At any moment we will have a variable called next token which will contained the next unscanned thing from the input. Call scanToken() to advance and return NULL if there are any problems.
- Types of languages and grammar: LL(1), LL(k), LR, LR(k), SLR, LALR, ambiguous grammars, ambiguous Languages
- Parser tool/ parser generators: the input is a grammar, and the output is a parser. Finally, the tool supposed to make writing a compiler easier.

3. Screenshot:

```
ParseMain
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java ...
*****
Parsing the statement: function a()
```