# Software Project – Assignment 3

*Due on 20.12.2015 (23:55)*

## Introduction

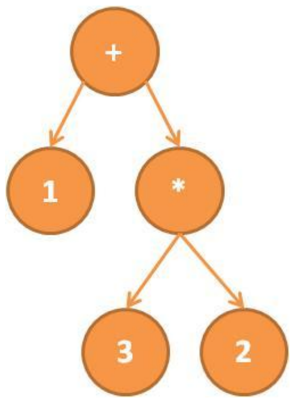
In this assignment, you will develop your coding skills in C and work with ANTLR to build a parser for simple calculator. Please note, in this assignment you are expected to build your code so it can be extended later when working on the project. Read carefully the guidelines given later on "Better code" section, this will significantly reduce the amount of work you will do when working on your project.

The assignment divided into 3 sections as follow:

1- C program – This program will receive a tree represented by a string in LISP-Style. Later the program will parse and build a tree from this string, and evaluate the expression represented by the tree.
2- Auto-Generated parser using ANTLR – In this section you will write a java program which parses an input and print the parse tree in LISP-Style. Before starting this section you need to carefully review the document on ANTLR which can be found on moodle.
3- Makefile and integrating two programs in Linux – In this section you will create your own makefile so it will build your project. Later we will use "pipes" in linux to integrate the java program with the C program.

## Representing a Arithmetic Expression As A tree

It is very useful to represent an arithmetic expression as a tree; we will call this kind of trees Expression Trees. Let us look at the following tree which is an expression tree:

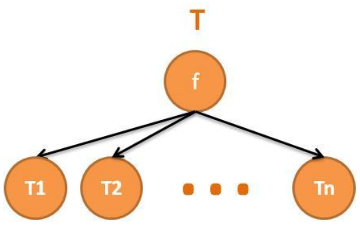


Expression trees have the following properties:

1- Leaves in such trees are operands
   In our example, the leaves are the operands 1,3,2.
2- Evaluating expression trees can be done recursively by evaluating the value of each subtree and apply the operation represented by the root node. The value of a leave (operand) is simply the value of the operand.
   In our example the value of the expression tree is 6
3- Arithmetic expression can be produced by recursively producing parenthesized subtrees and applying the operation represented by the root node.
   In our example the expression tree can produce the arithmetic expression
   (1)+((3)*(2))
   We see that parentheses on the leaves (operands) can be removed to get much clearer expression
   1+(3*2)

So in general to evaluate a tree we can follow the following rule:

$$Let\ T - Be\ a\ tree\ represented\ by\ the\ graph\ bellow, where\ T_i\ are\ subtrees\ of\ T.$$



$Given\ f(x_1, x_2, \ldots, x_n)\ a\ function\ with\ n\ variables.$
$Then\ \boldsymbol{Value}(\boldsymbol{T}) = \boldsymbol{f}\big(\boldsymbol{Value}(\boldsymbol{T_1}), \boldsymbol{Value}(\boldsymbol{T_2}), \ldots, \boldsymbol{Value}(\boldsymbol{T_n})\big)$
$You\ can\ think\ of\ an\ operator\ \blacksquare \in \{'+', '-', '*', '/', '\$'\}$
$as\ a\ function\ of\ two\ variables\ f_{\blacksquare}(x, y) = x \blacksquare y$

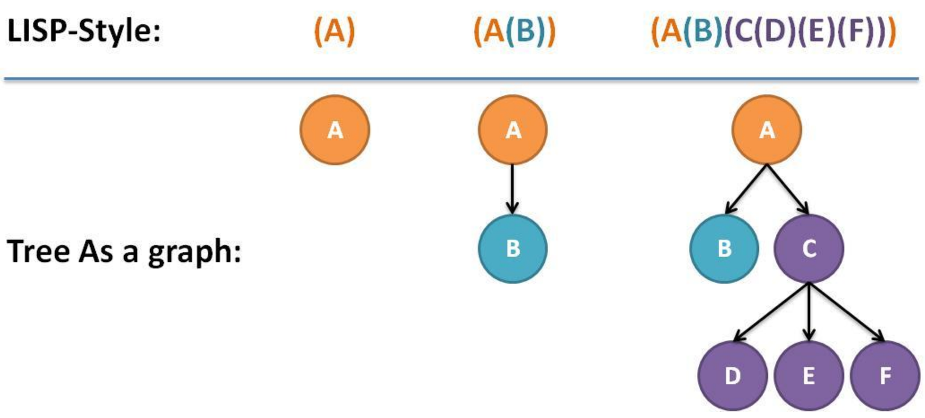
## Simple Calculator in C

In this section, you will need to implement a C calculator. The input is not a regular arithmetical expression, but a LISP-Style tree representing an expression tree.

Recall a LISP-Style tree is a string which represents a tree. The general form of a LISP tree is as follow:

$(strCHILD_1CHILD_2CHILD_3 \ldots CHILD_n)$

Where str is the string value of the root node and $CHILD_i$ is a LISP-Style representation of a tree represented by $CHILD_i$. For example:



**Remarks:**

- A string can be a single node (example "(A)" )
- A string can hold any number of children (example "(C(D)(E)(F))")
- The string must NOT contain spaces.

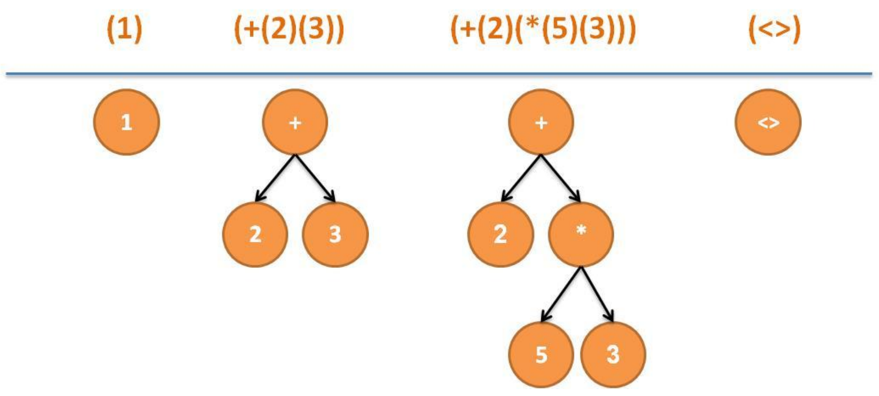
## Calculator

Write a program in C, which receives strings followed by a new line. Each string represents a LISP-style tree where a lisp-tree represents either an expression tree or a termination command. **In this section we will assume that the strings validly represent a LISP-Style tree and each tree either represents a valid arithmetical expression or a termination command.**

Example of an input:

```
(1)
(+(2)(3))
(+(2)(*(5)(3)))
(<>)
```

The input represents the following trees:

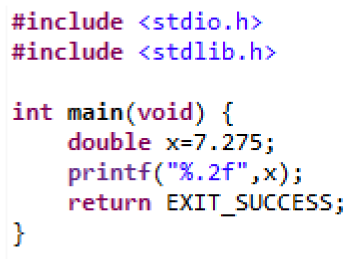


## Output

After each line the program will output the following:

1- If the string represents an arithmetical expression:

   a. If the expression has a valid result you need to calculate it and print:
   **"res = num".**
   Where num is the result of the calculation with 2 digits precision.
   Example
   >> (+(2)(*(5)(3)))
   >> res = 17.00
   >> (1)
   >> res = 1.00
   >>
   HINT: You can use the formatting in printf to print only two digits after the decimal point.



```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    double x=7.275;
    printf("%.2f",x);
    return EXIT_SUCCESS;
}
```

   **This will print 7.28**

   b. If the expression has an invalid result you need to print:
   **"Invalid Result"**
   Example:
   >> ($(5)(3))
   >> Invalid Result
   >>

2- If the string represents a termination command print:
   **"Exiting..."**
   Example:
   >> (<>)

```
>> Exiting...
>>
```

*NOTE: The program should exit! After printing "Exiting"*

3- In case any unexpected error occurs, you need to print **"Unexpected error occurred!"** and exit using the function **exit() (**google it).
   a. An error may occur for many reasons; malloc failure or passing null argument when you shouldn't etc…
   b. Although we assumed the input is valid, however your tree data-structure shouldn't assume this. Thus when passing a null argument as a string you should exit. The motivation behind this is this; When you want to use the tree data structure, and by accident you insert a child with a null string value, your program should exit. This will help you avoid unwanted segmentation faults when you run your program which in case happens, it will be much harder to trace the origin of the error.

**Notes:**

- Each message will be followed by a new line!
- Each string which represents an arithmetical expression can be calculated recursively. For example, the expression: 5+3 is represented by (+(5)(3)). And the expression: 2+5*3 is represented by (+(2)(*(5)(3))).

Assumptions

You can assume the following when writing your code:

- The maximal number of children for each node is at most 10
- Each Lisp-String is valid – This is guaranteed by the parser which we will see in later section. That is, you don't need to check the validity of your string, this will be guaranteed.
- Memory allocation is not guaranteed, in case of any memory allocation failure, your program should terminate with the following message
  **"Unexpected error occurred!"**
  **Notes:**
  ***** The message must be followed by a new line
  ***** Use exit() function to exit a program.

## Better Code

You are given the freedom on how to implement the C program. But we suggest you follow the guidelines bellow **(A code that doesn't follow these guidelines will cause in reduction in your grade)**:

- Divide your program into module, you should make each module functionality as general as possible. For example, if you need to build a tree data structure don't make it specific for the calculator. (Hint: You might need to use it in your project)
- One possible division:

  - Implementing a tree data structure. (The values of each node is a string)
  - Parsing a lisp-style string.
  - Rest of the program.
- To ease your debugging progress, make unit tests for each module.
- Make your code easy to understand. To do so you need to use meaningful names for variables. Write comments on parts of codes that are not trivial. This will help you when trying to modify the code later.
- When parsing a string which represents a LISP-Tree you might need to implement the following functions:
  - getRootStr() – A function which returns a copy of the string of the root node. Example getRootStr("(a(b)(c))" would return the new string "a".
  - getChildAtIndex() – A function which returns a copy of a string which represents the child at some index.
    Example getChildAtIndex("(a(b)(c))",0) would return the new string "(b)".
- Write short functions – Usually a function with more than 30 lines can be divided into sub-functions.
- Don't use numbers to represent some value – Use macro instead:
  - For example; the number 3.14 could represent many things, but if your write:
    #define PI 3.14
    it then represents the number π.
- Enums can be useful to define new type which has finite discrete values. For instance this can be used to represents errors.
- **Do not use global variables**; if you use global variable then something is WRONG. Please note that students who do use global variable their grade will be affected.

## ANTLR

***Remark:*** *Before going through this section, you need to make sure you know how to work with ANTLR. You can read the guide on moodle under the Tutorial materials section (It is recommended to review the tutorial slides as well).*

In the assignment zip file, there is a directory called SP. In this directory we will define all our java code (Including the auto generated code by ANTLR). Let us first review the Directory

SP includes the following files:

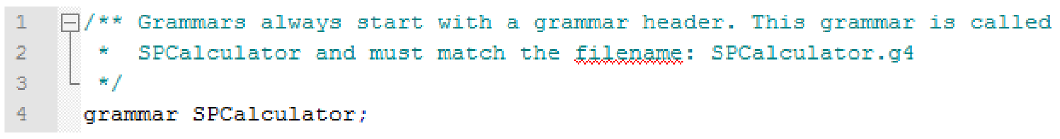
- **SPCalculator.g** – This is the grammar file which will define our Language Recognizer. You need to extend this file so the method **getLisp()** will work properly.
- **SPCalculatorMain.java** – This class contains the main method, note this class uses the auto-generated files by ANTLR. Changing the grammar name will result in an error when trying to compile this class.
- **SPTree.java** – Simple tree data structure that we will use in SPCalculator.g

- **makefile** – You will not need to change this makefile. This makefile will build our java program. We will invoke it from the makefile in the parent directory. [More on this file later]

We will shortly review the files SPCalculator.g and SPCalculatorMain.java, please read carefully the code given in SPTree.java (Make sure you understand the method **getLisp()**).

## SPCalculator.g

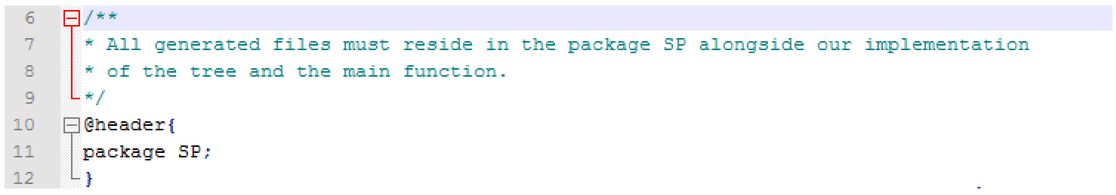
First, we define a grammar file by the syntax of ANTLR, we define our grammar to be SPCalculator, all generated files including the parser and the lexer will start with SPCalculator prefix. For instance, the generated parser java source code will be SPCalculatorParser.java.



```
/** Grammars always start with a grammar header. This grammar is called
 *  SPCalculator and must match the filename: SPCalculator.g4
 */
grammar SPCalculator;
```

Figure 1 SPCalculator.g: defining our grammar name.

Now, we need to tell ANTLR that all generated code will be in the package SP. The @header{} allows us to tell ANTLR what will be in the header (The first few lines in the source code) of our generated code (import statements as well). This is done as follow:

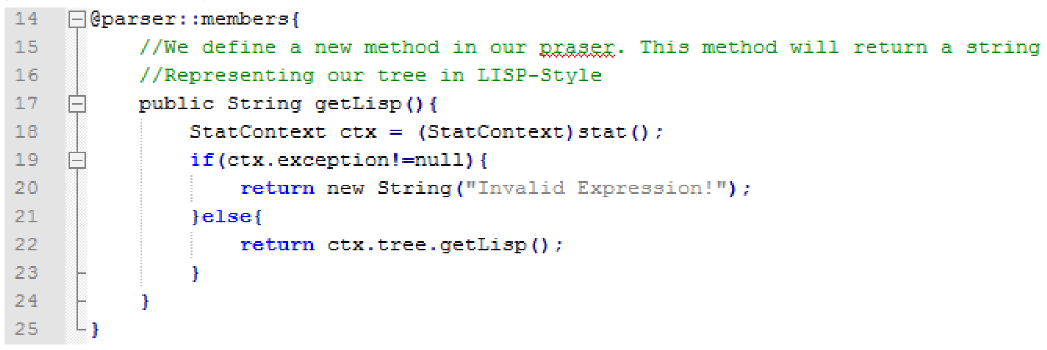


```
/**
* All generated files must reside in the package SP alongside our implementation
* of the tree and the main function.
*/
@header{
package SP;
}
```

Figure 2 SPCalculator.g: setting the package for all generated files

***Note:*** *If you need to use other packages such as java.io then you will need to put the import statement "import java.io.*;". But we will not need this so you don't need to change this part.*

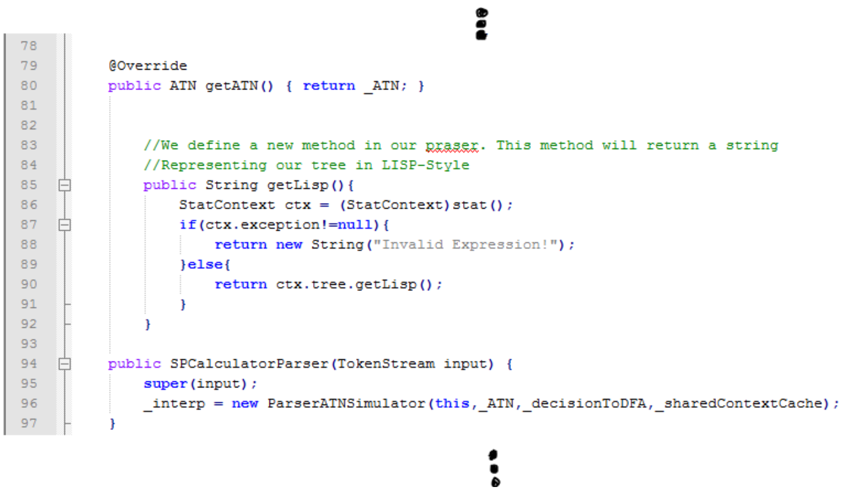
Next we will need to define a new method for the parser which returns a string representing the parsed tree. ANTLR allows us to write java code that will be embedded when the code is generated. The syntax to do that is this:



```
@parser::members{
    //We define a new method in our praser. This method will return a string
    //Representing our tree in LISP-Style
    public String getLisp(){
        StatContext ctx = (StatContext)stat();
        if(ctx.exception!=null){
            return new String("Invalid Expression!");
        }else{
            return ctx.tree.getLisp();
        }
    }
}
```

Figure3  SPCalculator.g : This will result in defining a new method in SPCalculatorParser.java

Let us review what we have above. First the identifier @parser::members{ <<Java Code>> } tells ANTLR, when generating SPCalculatorParser.java, put the java code inside the braces as a member. In our example, ANTLR will copy everything between Line 15 to 24 (Including comments) and will put it as a member in the praser class (i.e SPCalculatorParser.java). Thus, now our SPCalculatorParser.java class will have a public method which returns a String representing a LISP-Style tree. A snapshot on the generated parser would look something like this:

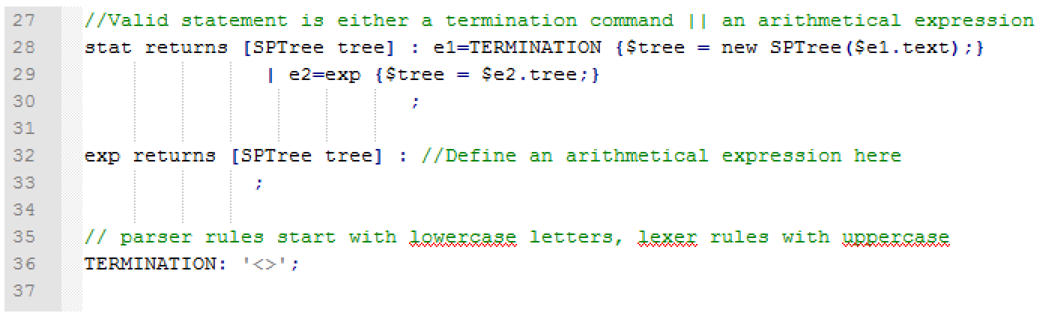


```
    @Override
    public ATN getATN() { return _ATN; }


        //We define a new method in our praser. This method will return a string
        //Representing our tree in LISP-Style
        public String getLisp(){
            StatContext ctx = (StatContext)stat();
            if(ctx.exception!=null){
                return new String("Invalid Expression!");
            }else{
                return ctx.tree.getLisp();
            }
        }

    public SPCalculatorParser(TokenStream input) {
        super(input);
        _interp = new ParserATNSimulator(this,_ATN,_decisionToDFA,_sharedContextCache);
    }
```

Figure 4 Snapshot of SPCalculatorParser.java where we can see the definition of our method

We see that ANTLR just copied everything between the braces and put it in the class SPCalculatorParser.java. We will examine the code of the method **getLisp()** later.

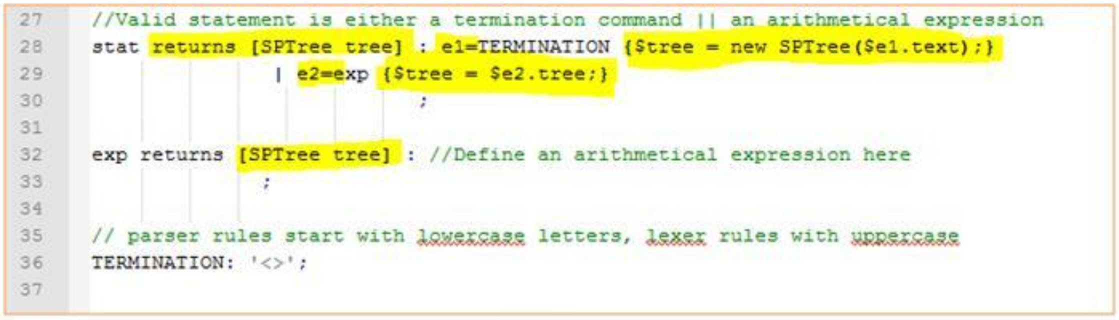
We continue examining the rest of the file and see where we actually define our rules (Using EBNF notation):



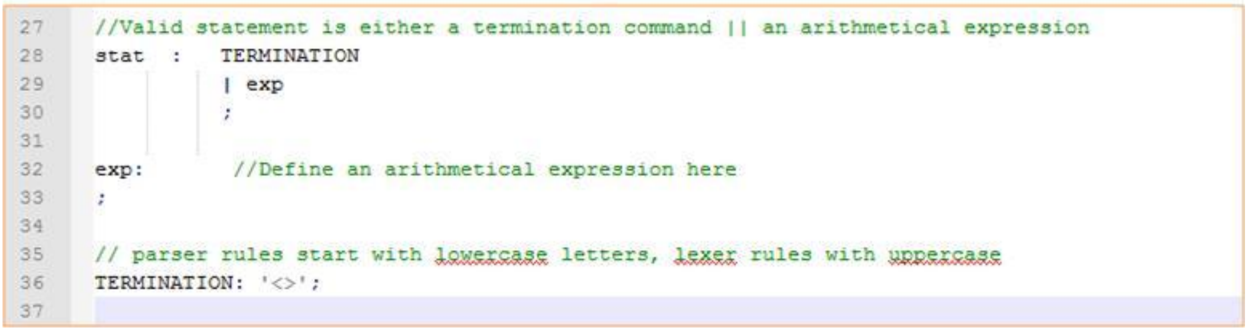
```
//Valid statement is either a termination command || an arithmetical expression
stat returns [SPTree tree] : e1=TERMINATION {$tree = new SPTree($e1.text);}
                 | e2=exp {$tree = $e2.tree;}
                         ;

exp returns [SPTree tree] : //Define an arithmetical expression here
               ;

// parser rules start with lowercase letters, lexer rules with uppercase
TERMINATION: '<>';
```

Figure 5 SPCalculator.g : Definition of rules using EBNF

Before getting into details, let us look at the grammer without the highlighted parts



```
27  //Valid statement is either a termination command || an arithmetical expression
28  stat returns [SPTree tree] : e1=TERMINATION {$tree = new SPTree($e1.text);}
29                 | e2=exp {$tree = $e2.tree;}
30                 ;
31
32  exp returns [SPTree tree] : //Define an arithmetical expression here
33             ;
34
35  // parser rules start with lowercase letters, lexer rules with uppercase
36  TERMINATION: '<>';
37
```



```
27  //Valid statement is either a termination command || an arithmetical expression
28  stat  :   TERMINATION
29            | exp
30            ;
31
32  exp:        //Define an arithmetical expression here
33  ;
34
35  // parser rules start with lowercase letters, lexer rules with uppercase
36  TERMINATION: '<>';
37
```

We see that our grammar has a rule named **stat** which defines all the words derived from **TERMINATION** (in this case the word “<>” which represents a termination command) and the words derived from **exp.** You will need to extend **exp** so you could derive all the arithmetical expressions from **exp**. **It is recommended to start with a simple grammar as we did in class for ArrayInit and then extend it so we could get our tree in LispStyle string.**

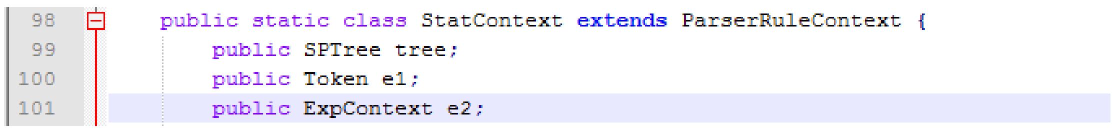
Getting back to **Figure 5** we see that we defined a token ***TERMINATION*** **(line 36)** which represents a termination command, a rule called ***exp*** **(line 32)**, and start rule ***stat*** **(line 28)**. When extending exp you may need to define new rules/tokens.

Recall from the tutorial, in the SPCalculatorParser.java code, each rule will be represented by a method with its name, that is our rule **stat** will have a method called **stat()** in

SPCalculatorParser.java. The signature of the function will look like this:

```
public final StatContext stat() throws RecognitionException
```

The class StatContext contains many things, in particular, when we write that a rule returns a value (**line 28**) the value in the brackets will be a member inside StatContext. Let's have a look:



```
98      public static class StatContext extends ParserRuleContext {
99          public SPTree tree;
100         public Token e1;
101         public ExpContext e2;
```

Figure 6 SPCalculatorParser.java : The return values will be defined as members in the class StatContext

We see that the first member in the class StatContext is actually "*SPTree tree*", our return value. Notice also that we have two other members **e1** and **e2** (**lines 100-101**).
*Note: Each method/rule returns a context class, the class name will be the rule name followed by Context. For example, the rule exp, will return a context class of type ExpContext.*

Referring back to **Figure 5** the following line:

**e1=TERMINATION {$tree = new SPTree($e1.text);}**

Tells ANTLR to set the value of tree (in **StatContext** class) to be a new SPTree which contains the text given by the token TERMINATION which is "<>".

The syntax we use here is different from java code, but we see a resemblance. When writing e1=TERMINATION, this tells ANTLR to refer to the token which matches TERMINATION by the variable e1. When we write $e1.text, we access the token text in this case it's "<>". The code inside the braces {$tree = new SPTree($e1.text);} will set the value of tree (The return value of our rule **stat**) to be a new SPTree object which has a single node with the text "<>".
*Note: You can write any java code within the braces, as well as using ANTLR syntax to refer to other rules as we did above).*
The next line:

**e2=exp {$tree = $e2.tree;}**

Tells ANTLR to set the value of tree (Again, the return value of the rule **stat**) to be the tree value returned by **exp().** As before, when we write e2=exp, we refer to the return value of exp with e2. When we write $e2.tree we get the tree returned from **exp()**. Thus the code $tree=$e2.tree will set the tree returned **stat** to be exactly the tree returned by **exp.**

In summary, when a text is matched to TERMINATION, the tree will hold a single node tree which has the value "<>", and when matched to exp the tree will hold a Parse tree representing a valid arithmetical expression.

**You need to build the tree recursively; each time you match a rule, this in turn must build a tree (Say T1) and then you can use inserChild() method to insert T1 as a child of the current tree you are trying build.**

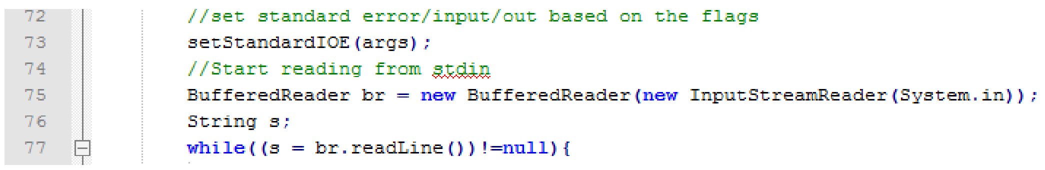
**Hints:**

- It is very recommended to start with a simple grammar file (as in the example ArrayInit.g) and use ANTLR test rig to see if the grammar defines the language we need (Arithmetic expressions and Termination commands).
- Remember, in EBNF, the first rule is matched will be invoked as well, this could be used in order to solve the order of the operations when we need to calculate parse tree.
- The web contains many grammar files for ANTLR. You can use it as a reference.
- Read more about parsing and parse trees on the internet so things would be clearer.

## SPCalculatorMain.java

We will examine only the main function in this file. The main function receives from the standard input an expression and eventually will output a lisp tree in case the parsing process successfully finished.

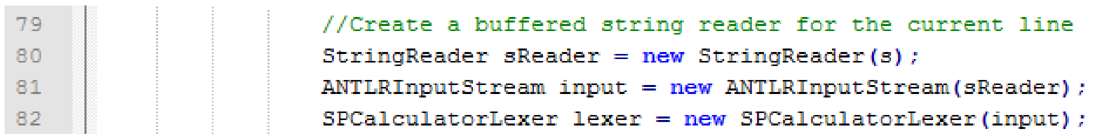
Let's have a look:



```
72        //set standard error/input/out based on the flags
73        setStandardIOE(args);
74        //Start reading from stdin
75        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
76        String s;
77        while((s = br.readLine())!=null){
```

**In line 73** we set System.in/System.out/System.err to be either the standard channel or a file. For instance; if the user put the flag **-o filename.txt** the output of our program will be redirected to a file called filename.txt, and if the user put the flag **–e filename.txt** all errors will be redirected to a file called filename.txt. Afterwards we start reading line after line from the input channel and store each line in a string called **s.**

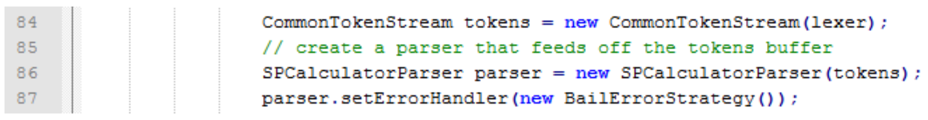
The following code:



```
79        //Create a buffered string reader for the current line
80        StringReader sReader = new StringReader(s);
81        ANTLRInputStream input = new ANTLRInputStream(sReader);
82        SPCalculatorLexer lexer = new SPCalculatorLexer(input);
```

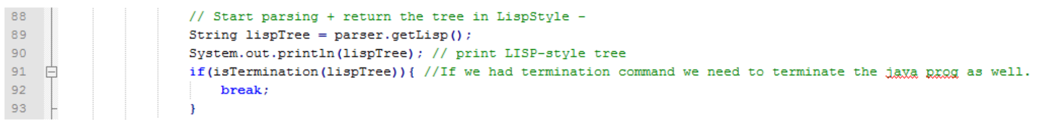
creates a lexer object (**SPCalculatorLexer**) which will later be used to tokenize the line given by the string **s**.

Next, we create a parser which feeds off tokens given by the lexer:



```
84        CommonTokenStream tokens = new CommonTokenStream(lexer);
85        // create a parser that feeds off the tokens buffer
86        SPCalculatorParser parser = new SPCalculatorParser(tokens);
87        parser.setErrorHandler(new BailErrorStrategy());
```

Finally we start parsing by invoking the method **getLisp():**

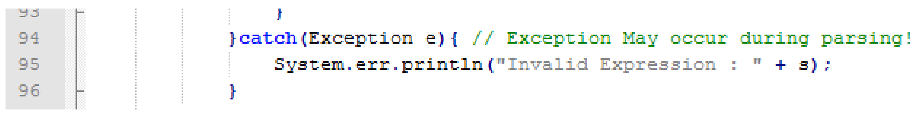


```
88        // Start parsing + return the tree in LispStyle -
89        String lispTree = parser.getLisp();
90        System.out.println(lispTree); // print LISP-style tree
91        if(isTermination(lispTree)){ //If we had termination command we need to terminate the java prog as well.
92            break;
93        }
```

Recall, that the method **getLisp()** invokes the method **stat()** which triggers the parsing process (The process starts from the rule stat). Referring back to **Figure**3 we see that the line:
StatContext ctx = (StatContext)parser.stat();
Will start the parsing process, and will return a Context class (i.e StatContext) which contains a member called tree that represents our parsed tree.

Finally in **Line 90** we print the lisp to System.out (This could be a file/standard output channel)

Notice that when invoking **stat()** this may cause the program to throw an exception in case of a parsing error, thus we catch this exception and print the error to the **System.err**



```
        }
    }catch(Exception e){ // Exception May occur during parsing!
        System.err.println("Invalid Expression : " + s);
    }
```

### Requirements for this parts

- You will only need to extend the file SPCalculator.g
- Don't change the following parts in SPCalculator.g:
    - grammar name
    - @header{}
    - @members{}
    - The rule stat
- Don't change SPCalculatorMain.java
- You may add TOKEN rules or parsing rules.

## Makefile and Pipes

### Makefile

In this section you will need to extend the make file in the zip file assigment3.zip (Note you will need not to change the makefile in SP directory). The requirements are as follow:

1- You need to change the rule "ex3" such that if any of your C source files was changed, the rule ex3 will be invoked to create your C program executable file
2- You need to change the rule "clean" such that invoking the rule "make clean" will result in deleting all the object files of your C code.
3- Your C program executable file name must be **"ex3"**
4- Your compilation command should be the same as given in previous assignments:
```
>> gcc -std=c99 -Wall -Werror -pedantic-errors
```
5- All your C source files must be located where the makefile located. Plus the directory SP must be in the current path as well. For example, if your C source code contains

(main.c, Tree.c, Tree.h), and your current directory is SPCalculator then the files in this directory must be the following:

   a. **main.c** , **Tree.c**, **Tree.h**, **makefile**
   b. The directory **SP** which contains:
      i. **makefile**
      ii. All the Java code and the generated files by ANTLR (**SPCalculatorMain.java** , **SPCalculatorParser.java**, **SPTree.java** etc..)
      iii. Your grammar file **SPCalculator.g**

After you changed the makefile invoking the following command

```
>> make all
```

Will first invoke the "all" rule in the makefile located at SP. Next it will invoke the rule **"ex3"** which will compile your C source code and generate an executable called ex3. The rule all in in the makefile SP/makefile will run ANTLR tool to generate the parser and lexer and then it will compile all the java source code to generate our java program.

Invoking:

```
>> make clean
```

Will first clean the directory SP by invoking rule clean in the makefile located at SP directory, and then it will delete all the object files in the current directory.

## Pipes

We saw how to redirect input/out channels of a program to a file. Now we will see how we can set an output of one program to be an input of another program. In linux, one can use pipes to do so.

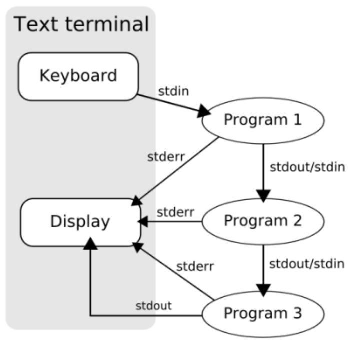
In linux a pipeline (pipe) is a set of processes chained by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one. The general form of a pipeline is as follow:

```
>> program1 | program2 | program3
```

This will execute program1,program2 and program3 concurrently where:

1- The input of **program1** is the standard input (keyboard)
2- The input of **program2** is the output of **program1** (The output of program1 is pipelined to program2)
3- The input of **program3** is the output of **program2**
4- The output of **program3** will be displayed on the screen.

This can be shown in the bellow figure:



Now that we know how pipes work. We will use the parser we built to generate lisp-trees and send lisp-tree strings to our C  program. Assuming we are currently in our working directory where all the files are executing:

```
>> java SP.SPCalculatorMain -i input1.in -e expected1.err | ./ex3 > expected1.out
```

Will result in running the java program SP.SPCalculatorMain and ex3 (The C program), where SP.SPCalculatorMain reads its input from the file input1.in and its output is piped to the program ./ex3 which will calculate the lisp tree and print the result to the file expected1.out. The flag –e expected1.err tells SP.SPCalculatorMain that in case of any errors, print it to the file expected1.err.

## Submission

Please submit a zip file named **id1_id2_assignment3.zip** where id1 and id2 are the ids of the partners. The zipped file must contain the following files:

1- Your source code of the C program
2- **makefile** – A makefile which follow the requirements in the makefile section.
3- **SP** – The directory which contains the java code, it must include the following:
    a. **SPCalculator.g –** Your grammar file you extended. This file must work and when giving to ANTLR, it is expected to generate working Java CODE
    b. **makefile –** The makefile given in the assignment file. DO NOT CHANGE THIS FILE
    **c.** **SPCalculatorMain.java –** The java source file for the main function. DO NOT CHANGE THIS FILE
    **d.** **SPTree.java –** The java source file for the Tree data structure. DO NOT CHANGE THIS FILE
**4-** **Partners.txt –** This file must contain the full name, id and moodle username for both partners. Please follow the pattern in the assignment files. **(Do not change the pattern)**

## Remarks

- For any question regarding the assignment, please don't hesitate to contact Moab Arar by mail: moabarar@mail.tau.ac.il or on moodle.

- Late submission is not acceptable unless you have the lecturer approval.
- Cheating is not acceptable.

*Good Luck*