Software Project – Final Project

Due on 17.1.2016 (23:55)

Introduction

Many faculty members were thrilled by the calculator we did in assignment 3, therefore due to its massive success we decided to go global. However, like in any software application, our first release of SPCalculator had many missing features and few unexpected bugs. Students from different universities are trying to compete with us, thus for the final project we will need to extend SPCalculator in order to meet the expectations of our users.

SPCalculator 2.0

In this section we will first describe the bugs we encountered in Assignment 3 and we will try and fix it, later we will introduce new features to our amazing SPCalculator.

Bug and missing features

1- In SPCalculator 1.0 we saw that when a user enters an expression which contains partial valid expression, the behavior of our application isn't as expected. The result we got was the value of this partial expression.
For example: When we entered "1+2hello" we got "result = 3.00".

2- A research conducted by our team advises us to support the following features:
   a. Operands may be variables as well as integers
   b. Support special functions such as: max and min
   c. Allow users to initialize variables using a file
   d. Allow SPCalculator to output the results into a file

Functional Requirements

1- In order to fix the bug we saw in SPCalculator 1.0, the general form of a statement in the next release will have the following form:

$$\boldsymbol{statement};$$

were a statement is either a termination command a valid arithmetical expression or assignment expression. A line which contains a semicolon ';' and a sequence of characters afterwards will discard everything after the semicolon, that is: for the following input "5+2;555", the 555 will be discarded.

2- Termination command is the sequence "<>". Any number of white-spaces after or before the sequence will be ignored. For example, the following statements represents a termination command:
"<>;"
"<> ;"
" <> ;"
Note: " < >;" this is not a termination command.

3- Arithmetical expression are said to be valid if and only if:
   a. It contains operands of integer type or valid variables (definition later).
   b. It contains valid operations and functions (definition later)
   c. It **doesn't** contain implicit operations, that is the following expression "5(3+2);", which contains implicit multiplication operation is not valid.
   d. It follows the syntax defined by regular arithmetical expressions

e. It contains balanced parentheses, that is, each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

Examples of valid arithmetical expressions:
"5+2;"
"5$2+-5+7-5$2;"
"3*(2+5);"
"max(a1,2,3,4) + 17;"
"min(1,2,3,max(5,3,4,14))+17*max(1+2,4,5);"
"((((5))))+(5)*((7)*7);"

4- A valid variable is a variable which contains at least one letter. Examples for valid variables:
"a"
"b"
"thisIsAValidVariable"
"ThisIsAValidVariable"
NOTE: Variable names cannot be the sequence 'max' or 'min'.

5- Valid operations are "$,/,*,-,+", the "+,-" operations can be unary operation or binary operations. The following expressions are valid:
"a+2; "
"+2-a; "
"he+llo ;"
"--a;"

6- The functions: max and min are valid. The general form of a max/min functions is as follow:

$$\max(\exp_1, \exp_2, \dots, \exp_n)$$
$$\min(\exp_1, \exp_2, \dots, \exp_n)$$

were $\exp_i$ is a valid arithmetical expression, note that the max/min functions must receive at least one expression followed by any number of expressions.
Examples:
"max(1,2,3,4);"
"min(1,3+2*5,max(1,2,3,4,5,6),7);"
"min(1);"
"min(max(1));"
Assumptions: you may assume that the total number of arguments in max/min functions doesn't exceed 10. That is the following is not possible as in input: "max(1,2,3,4,5,6,7,8,9,10,11);"

7- Assignment expression is an expression of the form:

$$var = exp$$

were var is a valid variable, and exp is a valid arithmetical expression.
Examples:
"a=5;"
"Abc=a+2;"

8- Evaluation of an expression must be as follow:
a. max/min functions returns the maximum/minimum value of its arguments.
b. Operations are defined as in a regular arithmetical expression ('$' is defined as in previous assignments.

c. Precedence of operations is defined as in previous assignments. Note how would you chose the precedence of max/min function so that the following expression "1+2-max(1,2,3)+4;" value would be 4. And the following expression's value "1+2*min(1,2,5+4);" would be 3. Do you think it should matter where you put max/min in the alternative list of the exp rule in ANTLR?

d. Evaluation of a valid assignment operation is as follow:

$$given\ an\ assignment\ expression\ "var = exp;"$$

Compute exp, check the result of exp:

In case exp has valid result; If var is already defined, then the new value of var is the result of exp, otherwise we will define a new variable 'var' and initialize its value to be the result of exp.

In case exp has invalid result; if var is already defined then its value will not be changed, otherwise nothing happens (Except for error message which will be defined later).

Examples of valid assignments:

"a=5/2;"

"b=7;"

"b= a + 2;" **(Given that 'a' is defined and has a valid value)**

Examples of invalid assignments:

"hello=10/3+5/a;" **(Given that 'a' has the value 0.0);**

"a=12$5;"

"a=b=19;"

e. Operands can be integers/variable; integer operands will have the regular value of this integer and variable operands will have a valid value if the variable is defined.

Examples: '5' will have the value 5, and 'a' will have the value 10 if it's defined and has the value 10. If 'a' is not defined then 'a' doesn't have a valid value.

f. A result of a valid expression is said to be valid i.f.f the value of this expression is also valid (If it contains variables, then all variable must be defined). Example of valid results:

"5+2;"

"10*2;"

"100/3$3;"

"max(1,2)$min(4,3);"

"max(1,2,3,4)$max(10/2,4)"

"a+2;" **(Given that 'a' is defined and has a valid value)**

Note: As we defined the $ operation, it is not valid for floats, thus the general rule for valid $ operation is:

$$\boldsymbol{a\$b\ is\ valid\ i.f.f.\ a \leq b\ and\ a, b \in \mathbb{Z}}$$

You may assume also that double results have decimal precision of 10^-6. That is 10.0000001 is not possible result, but 10.000001

The following results are invalid:

"1/0;"

"(5/3)$10;" **(Note unlike assignment 3, this is possible input)**

"(10*2)/max(-1,-2,-3,0);"

"max(1,2,3,4,5,20/3)$10;"

"max(100+2,23/(15-0),23,100);"
"a+2;" (Given that 'a' is not defined)

9- SPCalculator2.0 has two parts:
The first part is called **ANTLR Program**, it will parse an input using the java program we saw (You will not need to modify it), the parsed input (parse tree) will be piped to the C program (as we saw in assignment 3).
The second part is called **C Program**; the C program will evaluate expression tree given by the **ANTLR Program**.

10- SPCalculator 2.0 **ANTLR Program:** The only modification needed for this part is extending the grammar file SPCalculator.g. The ANTLR Program name is SPCalculatorMain, it resides in SP package. The program has optional command line arguments [-i filename] [-o filename] [-e filename]. Examples for possible runs:
>> java SP.SPCalculatorMain
This will run SPCalculatorMain main method wtih the standard input/output/error channels. (stdin,stdout,stderr)
>> java SP.SPCalculatorMain -i inputFile
This will run SPCalculatorMain main method with the standard output/error channels (stdout,stderr), however the input channel is read from the file inputFile
>> java SP.SPCalculatorMain -i inputFile -o outputFile -e errorFile
This will run SPCalculatorMain main method with the inputFile as the input channel, outputFile as the output channel and errorFile as the error channel.

11- SPCalcualtor 2.0 **C Program:** You will need to extend assignment 3 so you can support the new requirements for the next release (As given above). Before starting extending the **C Program,** make sure you have modified the ANTLR program so it supports everything we need. The requirements for the C Program are as follow:

a. Be able to evaluate any valid lisp tree (representing a valid expression tree) we get from ANTLR. NOTES:
* The way we defined the ANTLR program assures that any LISP tree is valid.
* The general structure of the LISP tree is not changed (Thus If you followed the guidelines in the assignment3 document, you can use the parser you did in ASSIGMENT 3)
* Like in assignment 3, we will not check the LISP tree produced by ANTLR. However we do expect the tree to be an expression tree.
For example: if the input for the ANTLR program is "2+3*2;" Then we expect the lisp tree to be: "(+(2)(*(3)(2)))" and **not** "(*(+(2)(3))(2))"

b. Support initializing variables from a file: This feature allows the user to give a filename as an input (in the argument list) and initializes variables from the given file. The file must exist and be readable, if not we will print an error and terminate **(More on error messages later). Note: If we initialize a variable more than once, the last initializing statement is valid.**
The general form of the file is

$$var = VALUE$$
$$var = VALUE$$
...

ASSUMPTIONS:
*** You may assume 'var' is a valid variable name.**
*** You may assume VALUE is a valid integer value**

*** you may assume every file follow the general form**
*** At least one whitespace is separating 'var' '=' and 'VALUE'**
Possible Input files:

```
A = 17
b  =   -100
b   = 7
ABC = -1
```

Impossible Input files:

```
A=17
b  =   -100/12
b1  = 7
ABC == -1
```

c. Support printing to a file; this feature allows the user to print results to a file including the input expressions. If the file exist but is read only we will print an error message (More on messaging later). If the file exists and is not read only, then all content of the file will be deleted, the new content will be the output of the current run of SPCalculator. If the file doesn't exist, we will create a new file with the given name.
Examples if the user specified the filename a.out, then for the following
Input (This input is fed form ANTLR part):
(+(*(2)(3))(2))
(/(2)(0))
The file **a.out** will contain:

```
( (2*3)+2)
res = 8.00
(2/0)
Invalid Result
```

NOTE: in case we initialize variables from a file, then you don't need to print the assignments to the output file (Or values). Just expressions from stdin will be printed into the output file.

d. The C program executable name must be **SPCalculator** (Unlike assignment 3 which was ex3). The C Program has command line arguments which are optional:
* -v filename : This flag tells SPCalculator to initialize the variables from the file 'filename'
* -o filename: This flag tells SPCalculator to print all result to the file 'filename' as stated in the previous subsection.
NOTE: The order in which the options appear doesn't matter. If both options are given, filenames must be different!
Examples:
>> ./SPCalculator
This will run SPCalculator with the standard channels
>> ./SPCalculator -v variableInputFile.txt
This will run SPCalculator with variable initializing as described in previous subsections
>> ./SPCalculator -v variableInputFile.txt -o outputFile.txt

This will run SPCalculator with variable initializing as described in previous subsections, plus the standard output channel will be the file 'outputFile.txt'

12- Messaging **ANTLR Part**: This was taken care of, you need not to do anything

13- Messaging in **C Part:**

a. If allocation failure occurs:
print:
"Unexpected error occurred!"
do:
Exit right afterwards. (Need not to care about memory leaks)

b. If the user entered invalid command line arguments that doesn't follow the general form of a command line argument:
print:
"Invalid command line arguments, use [-v filename1] [-o filename2]"
do:
Exit right afterwards. (You need to make sure there are no memory leaks when you exit)
Example:

```
>> ./SPCalculator -i variableInputFile.txt
>> Invalid command line arguments, use [-v filename1] [-o filename2]
>> ./SPCalculator variableInputFile.txt -v
>> Invalid command line arguments, use [-v filename1] [-o filename2]
>> ./SPCalculator -v -o filename filename
>> Invalid command line arguments, use [-v filename1] [-o filename2]
```

c. If the user entered valid command line arguments that follow the general form:

i. If both file names are equal:
print:
"Files must be different"
do:
Exit right afterwards. (You need to make sure there are no memory leaks when you exit)

```
>> ./SPCalculator -v filename.txt -o filename.txt
>> Files must be different
```

ii. If the variable initializing file doesn't exist or cannot be read:
print:
"Variable init file doesn't exist or is not readable"
do:
Exit right afterwards. (You need to make sure there are no memory leaks when you exit)
[Assume v1.in doesn't exist and o1.out is read-only]

```
>> ./SPCalculator -v v1.in -o out1.out
>> Variable init file doesn't exist or is not readable
```

iii. If the output file is read-only or cannot be created:
print:
"Output file is read-only or cannot be created"
do:
Exit right afterwards. (You need to make sure there are no memory leaks when you exit)

[Assume v1.in exists and o1.out is read-only]
>> ./SPCalculator -v v1.in -o out1.out
>> Output file is read-only or cannot be created

d. After receiving the command line arguments (or not since it's optional) we will receive from the standard input channels LISP trees. The output of the program depends on the output channel and the lisp string:
    i. If The output channel is the standard output channel (no output file was given in the command line arguments):
        1. If the expression is an arithmetical expression and 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
           >>
        2. If the expression is an arithmetical expression and has an invalid result you need to print:
           "Invalid Result"
           Example:
           >> ($(5)(3))
           >> Invalid Result
           >>
        3. If the expression is an assignment expression and has a valid expression then you need print:
           "var = num"
           Where 'var' is the variable name and num is the expression value with 2 decimal points percison
           Example:
           >> (=(a)(+(5)(7))) //This represents the assignment a=5+7;
           >> a = 12.00
        4. If the expression is an assignment expression and has an invalid arithmetical expression (defined as invalid assignment in previous sections) you need to print:
           "Invalid Assignment"
           Example:
           >> (=(x)($(5)(3))) //This represent the assignment x = 5$3;
           >> Invalid Assignment
           >>

5. If the string represents a termination command print (No memory leaks):
   "Exiting..."
   Example:
   >> (<>)
   >> Exiting...
   >>

ii. If The output channel is a file given in the command line channel (using the [-o filename] option):
   1. The messages are the same as in (d.i) but, you need to print the expressions right before every message:
      For example, if the output file is out.txt then for the following input:
      >> (+(5)(2))
      >> (=(a)(3))
      >> (+(a)(*(3)(2)))
      >> (=(b)(+(a)(1)))
      >> (<>)
      >>
      The file out.txt will contain the following text:

      ```
      (5+2)
      res = 7.00
      (a=3)
      a = 3.00
      (a+(3*2))
      res = 9.00
      Exiting...
      ```

iii. **NOTES:**
   1. **Each message should be followed by a new line!**
   2. **If there are two possible error messages, you should print the first message that appears in the order presented by previous sections.**
   3. **errors caused by invalid command line arguments shall be printed to stdout (Even if you succeeded in opening the file in the [-o filename] option).**

Non-Functional Requirements

Moab has decided to join your team, and he suggested doing the following:

1- Write the java program
2- Write a hash-table so we could map chars into doubles.

Due to the massive work he had to do, he quit before finishing the job. The progress he made is as follow:

- Finished writing the java program ☺
- Finished writing a linked list to solve collisions in a hash-map data structure☺.
  The linked list uses a Data-type called SPListElement, the name of the linked list data type is SPList. See the headers file before using it.
- Didn't finish the hash-map data structure ☹ Although, he suggest you to do the following:
  a. Use a 100 entries hash table
  b. Use the following hash function for strings:

```c
#define FIRST_PRIME 571
#define COEFFICIENT_PRIME 31
#define NUMBER_OF_ENTRIES 100
int hash(char* str){
    if(str==NULL){
        return -1;
    }else{
        int hash = FIRST_PRIME;
        for(int i =0;i<strlen(str);i++){
            hash = COEFFICIENT_PRIME*hash + (int)str[i];
            hash%=NUMBER_OF_ENTRIES;
        }
        return hash;
    }
}
```

  c. You should build your hash table such that:
  hashContains: O(maxListSize)
  hashInsert: O(maxListSize)
  hashDelete: O(maxListSize)
  hashGetValue: O(maxListSize)
  hashGetSize: O(1)
  hashIsEmpty: O(1)
  You may assume maxListSize=O(1) as well. Just make sure you do resolve collision by using the List you were provided.
- When you want to print a lisp-string as an expression, it is easier first to convert it to a tree using the parser and the tree data structure you built in Assignment 3. Afterwards you can implement a recursive function, such that given a node of a Tree it returns a string of the expression as follow:

```
String getExpressionOfTree(Tree NODE):
    if NODE.text is a VAR or INT:
        return Node.txt
    if NODE.text is '*','/','$':
        return '('+ getExpression(Node.child[0]) + NODE.text + getExpression(Node.child[1]) + ')'
...
```

  Hints & Assumptions:
  * You may assume that any tree can be fit into a string of size 1024
  * You may use **strcat** for this purpose.

Bonus Section

We will give a bonus section later. It's not obligatory; students who decide to do the bonus section will receive extra points on the Final Project grade.

Makefile

In this section you will need to extend the make file in the zip file finalproject.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 "SPCalculator" such that if any of your C source files was changed, the rule SPCalculator 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 **"SPCalculator"**
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, SPList.c, SPList.h, SPListElement.c, SPListElement.h**
    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 **"SPCalculator"** which will compile your C source code and generate an executable called **SPCalculator**. 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.

Integrating ANTLR With C

We will use pipes as we did in Assignment 3 in order to integrate **ANTLR Part** with the **C Program**

Submission

Please submit a zip file named **id1_id2_spcalculator.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 **(Including SPList.c SPList.h SPListElement.c and SPListElement.h)**
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