EECS-3311 – Lab – Calculator

L2

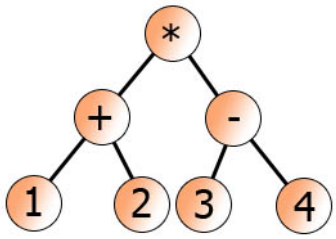
1/18/18 3:38:55 PM

1 Prerequisites

Lectures on contracting; Lab0 and Lab1; The Eiffel Language and Method; Use of the EStudio/IDE for construction of software, and testing and debugging it. Material covered in class and in the readings.

2 Introduction

Building compilers for programming languages is an important art and a science at the heart of software development and design. In this Lab, we build an interpreter for a very simple language – the language of arithmetic expressions.



((1+2)*(3-4))

The above figure[1] is an abstract syntax tree for an arithmetic expression whose *value* is -3. The *tokens*[2] are operators such as plus and minus and values such as integers. We require also the use of real numbers in expressions such as:

(99.8843 + 7.823) - (44.882/23.85668)

We will build a program *calculator* so that we can do the following from the command line:

```
red> calculator
Enter expression:
(99.8843 + 7.823) - (44.882/23.85668)
105.826
```

We would like to use fixed point decimals and do the calculations in 32 bit real numbers (REAL_32) as defined in the IEEE 754 Standard.

[1] Taken from https://www.codeproject.com/Articles/10316/Binary-Tree-Expression-Solver

[2] In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, though scanner is also a term for the first stage of a lexer. A lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth. (*Wikipedia*).

2.1 Dijkstra's two stack algorithm

Dijkstra's two stack algorithm (ingeniously) uses two stacks, one for the operators (such as plus and minus) and another for the values (in our case fixed point decimal numbers).[3]

Figure 1: Dikstra two stack algorithm

```
    ops:   STACK[STRING]
    vals:  STACK[REAL]
    value: REAL

    evaluate(a_expression: STRING)
        require
            -- ??? need a space between each token
        local
            list: LIST[STRING]; token: STRING; l_val: REAL; l_ops: STRING
        do
            create {ARRAYED_LIST[STRING]}list.make (10)
            list := a_expression.split (' ') -- tokenizer
            from list.start
            until list.after
            loop
                token := list.item
                if token ~ "(" then -- do nothing
                elseif token ~ "+" then
                    ops.push (token)
                elseif token ~ "-" then
                    ops.push (token)
                elseif token ~ "*" then
                    ops.push (token)
                elseif token ~ "/" then
                    ops.push (token)
                elseif token ~ ")" then
                    l_ops := ops.top
                    ops.pop
                    l_val := vals.top
                    vals.pop
                    if l_ops ~ "+" then
                        l_val := vals.top + l_val
                        vals.pop
                    elseif l_ops ~ "-" then
                        l_val := vals.top - l_val
                        vals.pop
                    elseif l_ops ~ "*" then
                        l_val := vals.top * l_val
                        vals.pop
                    elseif l_ops ~ "/" then
                        l_val := vals.top / l_val
                        vals.pop
                    end
                    vals.push (l_val)
                else
                    vals.push (token.to_real)
                end
                list.forth
            end
            value := l_val -- store the value of the expression
        end
```

[3] For the program in Java, see: https://algs4.cs.princeton.edu/13stacks/Evaluate.java.html

The above algorithm for `evaluate(some_string_expr)` requires that the string has spaces around each token (including brackets), e.g. ((1 + 2) * (3 – 4)). This is because the algorithm uses {STRING}*split* to obtain the tokens, splitting on a space.

But what happens if we enter (1+ 2)* (3 – 4)? Disaster! The 5th last line (`vals.push (token.to_real))` will crash with the attempt to convert the string "`2)`" to a real number.

The two-stack algorithm also requires that expressions be fully bracketed, whereas we require the use of precedence, and (optionally) to leave off the outer brackets.

2.2 We need a precondition for evaluate

What we need is a precondition for *evaluate* that will check when a string is a legitimate arithmetic expression. Ideally, we should be able to ignore whitespace. Also, an expression such as (4++3) is not valid and there should be some query to determine this before proceeding to evaluation.

Sometimes it is easy to write preconditions. Sometimes more work is needed. In this case, we cannot ignore the need for a precondition, if our calculator is to be useful. So, we need to do some work to obtain a useful precondition.

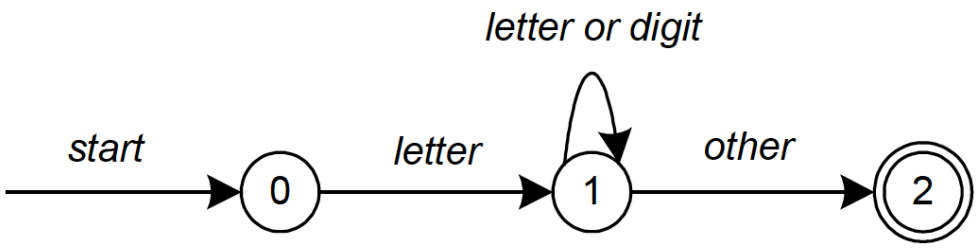
2.3 Parsing and Facade Design Pattern

What we need is a parser for arithmetic expressions written as strings. Compilers, spreadsheets programs and calculators all need this type of machinery.

Not all the students in this course will have learned about parsing (in a *Theory of Computation* course) using:

- Regular expressions (and state machines);
- Context Free Grammars (and pushdown automata).

With the knowledge of the above it is relatively easy to design a parser for arithmetic expressions.



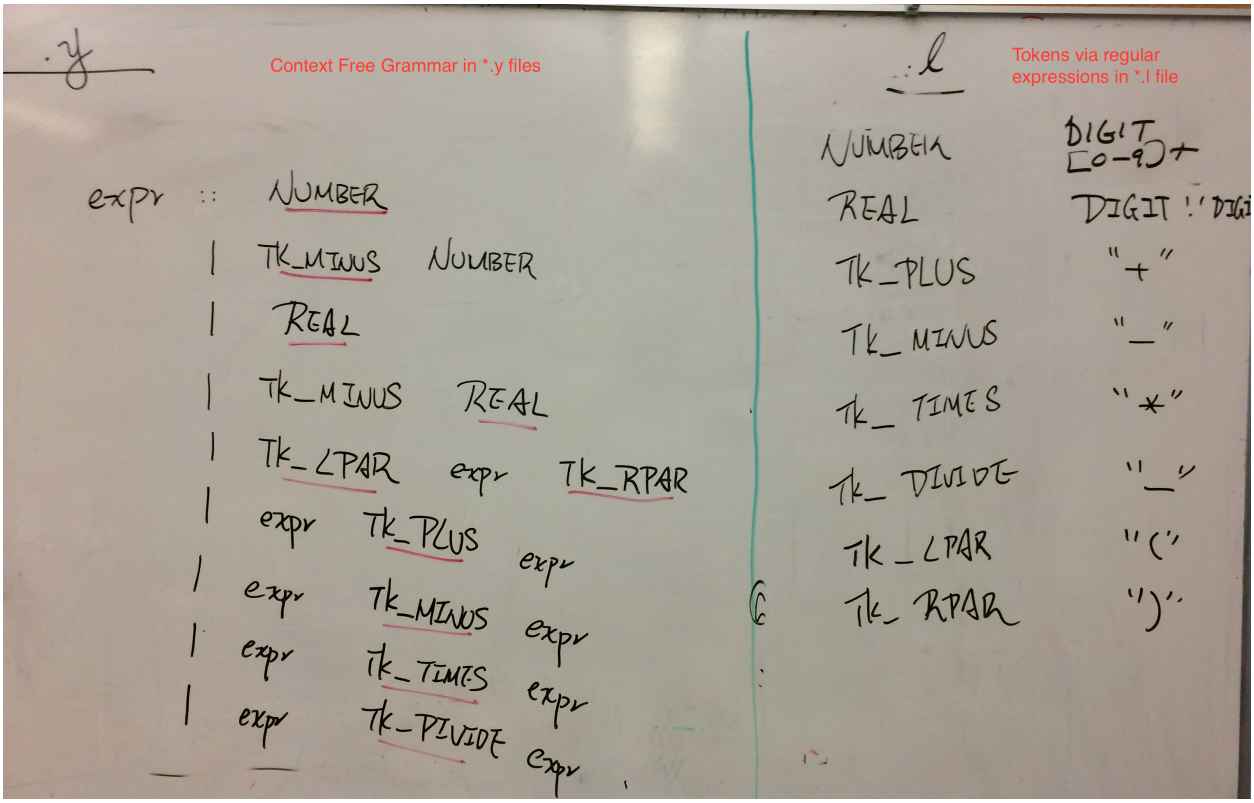
The state machine above (or the corresponding regular expression: `letter(letter|digit)*`) can be used to describe an identifier (the symbol "|" stands for alternation and "*" for repetition). You should familiarize yourselves with regular expressions and context free grammars. Most programming languages have libraries for using regular expressions.

Regular expressions are not powerful enough to describe expressions where the parentheses are balanced, i.e. number of right parentheses cannot exceed the number of left parentheses as we scan from left to right. A finite state automaton cannot remember the number of parentheses encountered. Thus, we need a more powerful language called context free grammars (implemented with a pushdown automaton).

Designing parsers has been made relatively easy using tools such as "lex" and "yacc". In Eiffel, these tools are called "*gelex*" and "*geyac*". We can use these tools to generate an abstract syntax tree for arithmetic expressions, making it feasible to compute with these expressions.[4]

2.4 Specifying the context free grammar for arithmetic expressions

To use gelex/geyaccc, we first specify the context free grammar for the arithmetic expressions we wish to parse. See below. Although not shown, unary expressions are also supported.



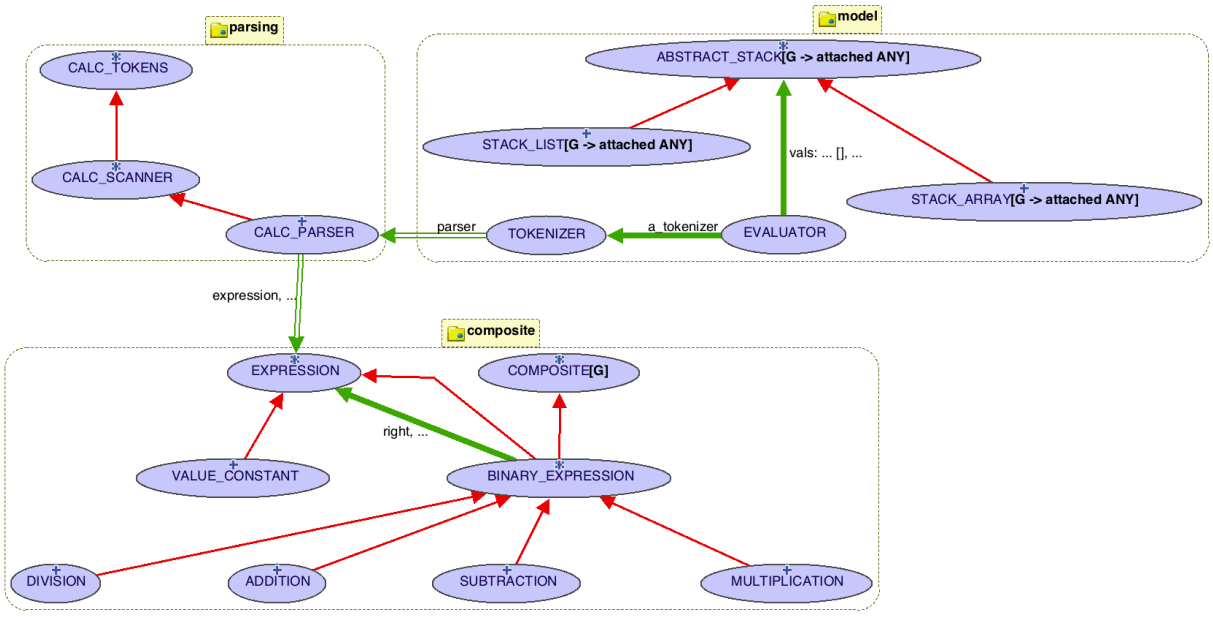
[4] To find out more see https://www.gobosoft.com/eiffel/gobo/gelex/ and https://www.gobosoft.com/eiffel/gobo/geyacc/.

[**Begin Aside**: If you would like to explore a smaller tutorial project than this one, with some documentation, see https://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/yacc-tutorial/docs/index.html (you need to login with "*anonymous*" and no password). You may obtain this smaller project and documentation by doing the following at the command line:
`> svn export http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/yacc-tutorial/`
We use the parsing in this Lab, but you do not need to understand how it is done, unless you optionally would like to explore more. **End Aside**]

We have done the parsing part for you and also provide a *façade* class called TOKENIZER that allows you to test if a string is a legal expression and to obtain the abstract syntax tree so that you can evaluate the expression using Dijkstra's two stack algorithm.

The *facade design pattern* provides a class that provides a simplified interface to a larger body of code, such as a class library. A facade can make a software library (or complex system) easier to use, understand, and test. The façade also reduces dependencies of code on the inner workings of a library or complex system. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.[5] Your work will depend only on class TOKENIZER. So the façade is an important class for *separation of concerns*.

The only classes you will need to work with are in the cluster *model*. See below.



[5] https://en.wikipedia.org/wiki/Facade_pattern

The BON diagram above was derived automatically using the EStudio IDE. BON diagrams (or UML diagrams) are useful for documenting and understanding the architecture of a design.

The clusters are as follows:

- *model*: stack classes which you must construct to be used by class EVALUATOR which you must complete.
- *parsing*: scanner and parser produced by gelex/geyacc from the context free grammar
- *composite*: where we store the abstract syntax tree for arithmetic expressions. This uses the *composite design pattern* and the *iterator design pattern* which we study in this course.

The above class diagram was generated automatically by the EiffelStudio IDE from the code. This synchronization is useful as the architecture may keep changing. The significance of the diagram is that it describes the *system architecture*, a major aspect of design.

3 Starter code

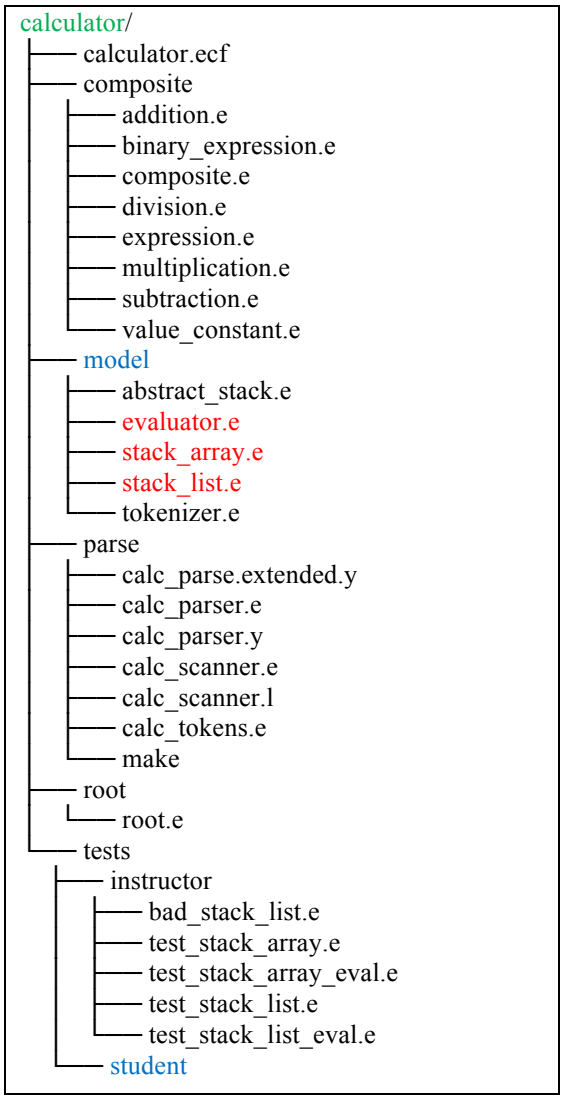
When you retrieve this Lab, you are provided with a folder *calculator* with the starter code shown in Figure 2.

You are required to add a class STUDENT_TESTS to the cluster student with at least 4 tests of your own (but much more is better).

The only classes you will work on are:

- STUDENT_TESTS
- EVALUATOR
- STACK_ARRAY and STACK_LIST which are implementations of an abstract stack.

Figure 2: Starter Code directory structure



When you first compile the project, and execute the unit tests we have provided you obtain the following:

Note: * indicates a violation test case

FAILED (18 failed & 5 passed out of 23)		
Case Type	**Passed**	**Total**
Violation	1	5
Boolean	4	18
All Cases	5	23
State	**Contract Violation**	**Test Name**
Test1	TEST_STACK_ARRAY	
PASSED	NONE	t0: create empty stack
FAILED	Postcondition violated.	t1: create stack, pop and push
FAILED	Postcondition violated.	t2: test array implementation
Test2	TEST_STACK_LIST	
PASSED	NONE	t0: create empty stack
FAILED	Postcondition violated.	t1: create stack, pop and push
FAILED	Postcondition violated.	t2: test list implementation
PASSED	NONE	*t3: test {STACK_LIST}.push post-condition of push not satisfied with bad implementation
Test3	TEST_STACK_ARRAY_EVAL	
FAILED	Check assertion violated.	t1: Evaluate (16.2) and also error condition
FAILED	Check assertion violated.	t2: Evaluate (-922337267) and error condition
FAILED	Check assertion violated.	t3: Evaluate (2+ 3) and ((3.1 + 2.9) + (2 + 8))
FAILED	Check assertion violated.	t4: Evaluate (2 - 3) and ((2 - 3) * (2.1 + 8))
FAILED	Check assertion violated.	t5: Evaluate (-2 - 3) and ((-2 - 3) * (-2.1 + 8))
PASSED	NONE	t6: (2.34*6.1 - 110.1/42.3))+3.5 unbalanced parenthesis better error reporting needed ...
FAILED	NONE	*t10: bad syntax (99.8843 ++ 7.823) - (44.882/23.85668) Precondition of `evaluate' violated due to `++'
FAILED	NONE	*t11: (.1 + .1 + .1) syntax not allowed, need 0.1
Test4	TEST_STACK_LIST_EVAL	
FAILED	Check assertion violated.	t1: Evaluate (16.2) and also error condition
FAILED	Check assertion violated.	t2: Evaluate (-922337267) and error condition
FAILED	Check assertion violated.	t3: Evaluate (2+ 3) and ((3.1 + 2.9) + (2 + 8))
FAILED	Check assertion violated.	t4: Evaluate (2 - 3) and ((2 - 3) * (2.1 + 8))
FAILED	Check assertion violated.	t5: Evaluate (-2 - 3) and ((-2 - 3) * (-2.1 + 8))
PASSED	NONE	t6: (2.34*6.1 - 110.1/42.3))+3.5 unbalanced parenthesis better error reporting needed ...
FAILED	NONE	*t10: bad syntax (99.8843 ++ 7.823) - (44.882/23.85668) Precondition of `evaluate' violated due to `++'
FAILED	NONE	*t11: (.1 + .1 + .1) syntax not allowed, need 0.1

You must get all these unit tests to work (one by one) and add your own tests

3.1 Stacks and a mathematical model to specify them

We will need a stack class for the Dijkstra two-stack algorithm. While the base library does contain various stack implementations, in this Lab, you will construct your own stacks from an ABSTRACT_STACK[G] *specification* in generic parameter G. The *contract view* is shown below.

Figure 3: Contract view of abstract stack

```
deferred class interface
        ABSTRACT_STACK [G -> attached ANY]

feature -- model
        model: SEQ [G] -- abstraction function

        is_equal (other: like Current): BOOLEAN
                        -- Is other attached to an object considered
                        -- equal to current object?
                ensure Result ~ model.first
feature -- Queries
        count: INTEGER_32 -- number of items in stack
                ensure Result = model.count

        top: G
                require not model.is_empty
                ensure Result = (model ~ other.model)

feature -- Commands
        push (x: G)
                        -- push x on to the stack
                ensure
                        pushed_othewise_unchanged:
                            model ~ ((old model.deep_twin) |<- x)       -- prepended
        pop
                require not model.is_empty
                ensure model ~ old model.deep_twin.tail

end -- class ABSTRACT_STACK
```

The contract view is generated by the EiffelStudio IDE (Integrated Development Environment). The IDE can generate a variety of documentation views including BON/UML diagrams. You are required to learn how to use these views in your own design documentation in later Labs and the project.

The *specification* of **deferred** class ABSTRACT_STACK is based on an immutable mathematical class SEQ[G] (from library *mathmodels*)[6], which provides queries for a finite sequence for elements of type G. Queries are side effect free and return sequences via deep_twin. A valid index is in 1..*count* and array notation can be used (e.g. `model[1]` for the first element of the sequence), as well as iteration (**across**). Sequences have a first item (the head), a tail and last item. The infix notation for *prepended_by(x:G)* is: `|<-`.

The *push* postcondition is specified as

```
model ~ ((old model.deep_twin) |<- x)
```

It asserts that new model sequence is the same as the old model sequence, except that "x" is prepended to the old sequence. Thus all the original elements must remain unchanged. Any descendant class (such as STACK_ARRAY) is required to satisfy the specification. This is

[6] https://www.eecs.yorku.ca/course_archive/2016-17/W/3311/eiffel-docs/mathmodels/seq_chart.html. This documentation was generated by the IDE.

called *subcontracting* and thus child classes cannot be bad implementations (this will be enforced via *runtime assertion checking*).

Thus, a complete specification is much more than a Java interface. A Java interface describes the signatures of features such as *push*, but not its semantic specification.

4 Design Decisions

Given a specification of a class, we need to decide how to implement it.

There are a variety of possibilities for a stack. You will need to implement the abstract stack in two ways, first with an ARRAY[G] and second with a LIST[G].

For each implementation, you must write an abstraction function model that glues the implementation to the model. The *class invariants* will guide you in this endeavor.

It is important to study how to use these collection classes. In Figure 1, you can see one way to iterate over a LIST class. You can also use the **across** iterator. Arrays in Eiffel are adjustable in size.[7]

To implement an abstract stack, a class such as STACK_ARRAY[G] will need to define the abstraction function. The following is the flat view of STACK_ARRAY, which shows the contracts inherited from the abstract stack:

[7] The command *force* (v: G; i: INTEGER) may be used to add an additional element e.g. at `i = count + 1`. See the *chart view* at https://www.eecs.yorku.ca/course_archive/2016-17/W/3311/eiffel-docs/elks/array_chart.html

```
class
	STACK_ARRAY [G -> attached ANY]
create
	make

feature {NONE, ES_TEST} – creation

	implementation: ARRAY [G] -- implementation of stack as array

	   …

feature  -- model
	model: SEQ [G]
			-- abstraction function
		do
			-- To be Done
		end
feature -- Commands

	push (x: G)
			-- push x on to the stack
		do
			-- To be Done

		ensure -- from ABSTRACT_STACK
			pushed_othewise_unchanged:
			  model ~ ((old model.deep_twin) |<- x)
		end

invariant
	same_count: model.count = implementation.count
	equality: across
			1 |..| count as i
		all
			model [i.item] ~ implementation [count + 1 - i.item]
		end
		comment ("top of stack is model[1] and implementation[count]")
```

The class invariant provides the information needed to construct the abstraction function *model* that glues the implementation to the specification: the last element of the array is the top of the stack.

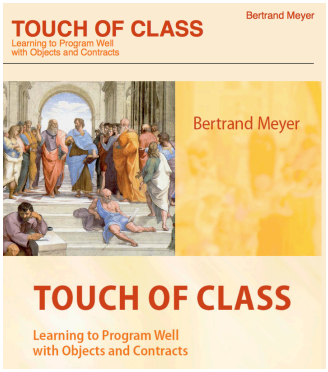
Features such as *push* inherit the contracts of the abstract stack and must be implemented in terms of arrays, lists etc.

5 Learning Outcomes

This Lab should produce understanding of the following

- Using inheritance for abstraction: the *complete* description of an abstract stack and a variety of implementations.
- The importance of specifications vs. implementations. *Subcontracting*: the stack implementations must satisfy the abstract (but complete) specification via pre/post conditions described in terms of a mathematical sequence. *Preconditions*: the difficulty yet importance of determining if a string is a legal expression before evaluating it. *Class invariants*: to ensure that implementations are glued to the mathematical *model*, i.e. the abstraction function. Runtime assertion checking ensures that the design is correct.
- Familiarizing yourself with software construction using collection classes such as ARRAY and LIST. LIST is a deferred class with implementations such as ARRAYED_LIST and LINKED_LIST. Although it is not required for this Lab, you can see how to represent abstract syntax trees via the parsing classes.
- Understanding the importance of design patterns. In this Lab, you see a variety of such patterns at work: iterator pattern, composite pattern, and façade pattern. In this Lab, we have provided code for these patterns and you merely use the code.
- Understanding the importance of testing and test driven development. Getting tests to work and writing your own tests:
    - In Test-driven development (TDD) the developer (a) adds an (initially failing) new test case that defines a desired improvement or new function, (b) constructs reliable code to make the test work, then (c) repeats the processes by adding yet another new test. A critical aspect of this is *regression testing* in which the developer automatically re-runs all the old tests to check that the previously developed and tested system still performs the same way after it was changed or extended changed by the new unit test.
- Understanding design decisions that must be made when constructing software

Resource: The textbook *Touch of Class* is available online at Steacie Library. It explains object-oriented software construction, algorithms and data structures in Eiffel terms, but also deals with contracting in detail. Consult this text as needed.



http://touch.ethz.ch

6 What you must do

1. Download the starter code
2. Add correct implementations and additional contracts to the following classes
    a. STACK_ARRAY
    b. STACK_LIST
    c. EVALUATE
    d. Add a class STUDENT_TESTS in the *student* cluster for your own tests
3. Work incrementally one feature at a time. Run all unit regression tests before moving to the next feature. This will help to ensure that you have not added new bugs, and that the prior code you developed still executes correctly.
4. Add at least 4 tests of your own to STUDENT_TESTS, i.e. don't just rely on our tests.
5. Don't make any changes to any of the other classes
6. Ensure that you get a green bar for all tests. Before running the tests, always freeze first.

You must make an electronic submission as described below.

1. On Prism (Linux), *eclean* your system, freeze it, and re-run all the tests to ensure that you get the green bar.
2. *eclean* your directory *calculator* again to remove all EIFGENs.
3. Submit your Lab as follows: `submit 3311 Lab2 calculator`

Remember

- Your code must compile and execute on the departmental Linux system (Prism) under CentOS7. That is where it must work and that is where it will be compiled and tested for correctness.
- Equip each test t with a *comment* ("t: …") clause to ensure that the ESpec testing framework and grading scripts process your tests properly. (Note that the colon ":" in test comments is mandatory.). An improper submission will not be given a passing grade.
- The directory structure of folder *calculator* **must** contain the structure outlined in Figure 2.

We have not provided you with all the tests we will be using to grade your software. We thus encourage you to write your own tests to ensure the correctness of your submission. See the above description of Test Driven Development.

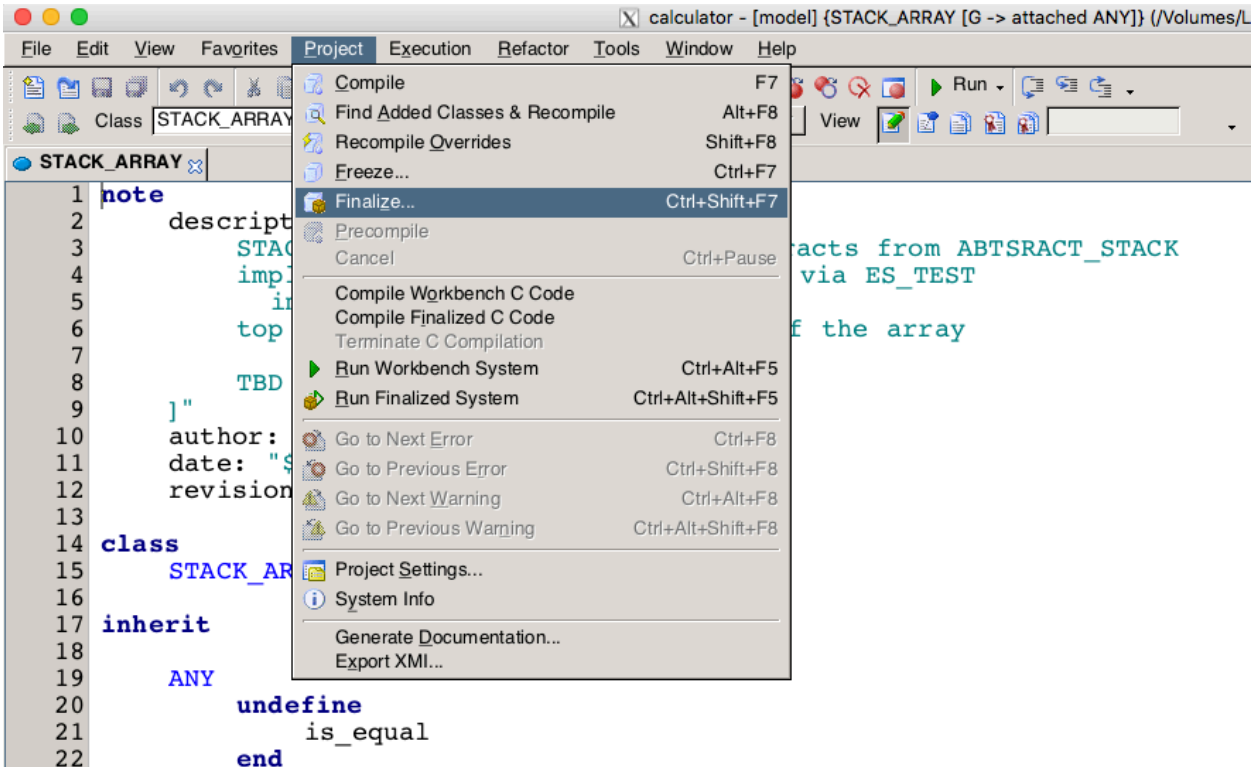
7 Appendix Deployment

The root class allows you to choose between a testing mode (via ESpec) and a finalized executable program.

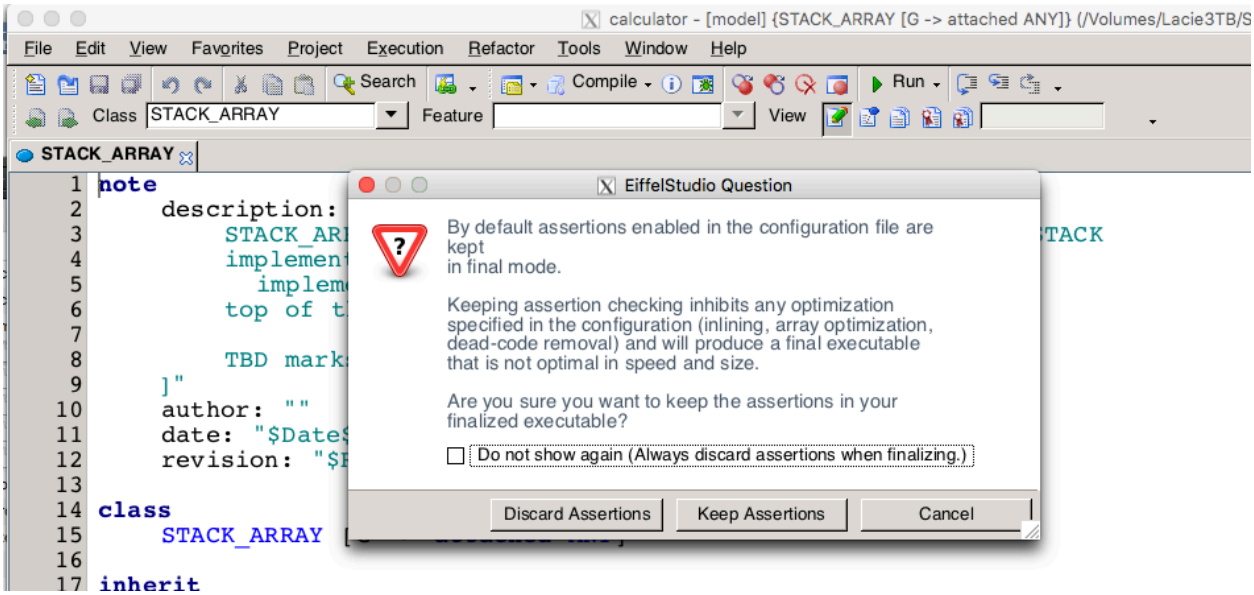
```
class
	ROOT

inherit
	ES_SUITE

create
	make

feature {NONE} -- Initialization

	Is_print: BOOLEAN = False
			-- when submitting leave this as false

	make
		local
			line: STRING_8
			eval: EVALUATOR
		do
			if not Is_print then
				add_test (create {TEST_STACK_ARRAY}.make ("array"))
				add_test (create {TEST_STACK_LIST}.make ("list"))
				add_test (create {TEST_STACK_ARRAY_EVAL}.make ("array"))
				add_test (create {TEST_STACK_LIST_EVAL}.make ("list"))
				run_espec
			else
				create eval.make ("array")
				print ("Enter expression:%N")
				Io.read_line
				line := Io.last_string.twin
				if eval.is_valid (line) then
					eval.evaluate (line)
					print (eval.value.out)
					Io.new_line
				else
					print ("Expression not valid%N")
				end
			end
		end

end -- class ROOT
```

To obtain an independent executable that you can provide to your friend (deployment) do the following:

- Set: `Is_print: BOOLEAN = True`
- Finalize as shown below



Click on *Discard Assertions* for efficient execution:



The executable **calculator** will be in the EIFGENs directory *F_code*. This executable is independent of the workbench and will work on its own on any Centos7 system or later.