L5

EECS-3311 – Lab – Analyzer

3/17/19 12:44:54 PM

1 Design Goals

```
require
     across 0 |..| 4 as i all lab_completed(i.item) end
     read accompanying document: Eiffel101[1]
ensure
     submitted on time
     no submission errors
rescue
     ask for help during scheduled labs
     attend office hours for TA William
```

This lab requires the implementation of the **visitor design pattern**. This Lab also makes use of the composite design pattern to parse expressions, check if their type is correct and evaluate them. The actual parser for expressions is provided.

2 Resources

- Lecture recording of the **composite** and **visitor design patterns** from Section M can be found here (Lectures 15 and 16):

  https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS3311_W19

- A tutorial series on the **composite** and **visitor design patterns** is available here:

  https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#composite_visitor

Aside: In the sequel, you are provided with a context free grammar for simple expressions, and the parsing is done for you via the Eiffel *gelex* and *geyacc* tools. You do not need to know how this parser works. However, if you are interested in learning more (not needed for this course), a simple tutorial and example is provided at (with anonymous login):
https://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/yacc-tutorial/docs/index.html

[1] See: https://www.eecs.yorku.ca/~eiffel/eiffel101/Eiffel101.pdf

3 The Problem

In this lab you are required to *analyze* (i.e., evaluate, calculate type, and type check) any given string whose **syntax** conforms to the following context-free grammar (CFG) of a small expression langauge:

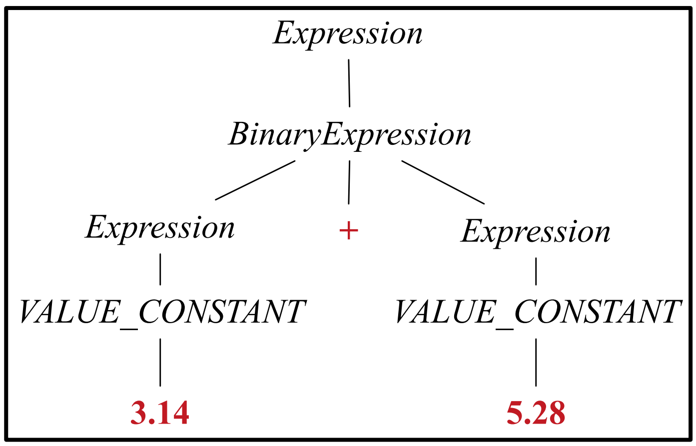
```
Expression ::=
    VALUE_CONSTANT
  | BinaryExpression
  | UnaryExpression
  | ( Expression )

BinaryExpression ::=
    Expression  +    Expression      /* addition */
  | Expression  –    Expression      /* subtraction */
  | Expression  *    Expression      /* multiplication */
  | Expression  /    Expression      /* division */
  | Expression  =    Expression      /* equal to */
  | Expression  /=   Expression      /* not equal to */
  | Expression  >    Expression      /* greater than */
  | Expression  >=   Expression      /* greater than or equal to */
  | Expression  <    Expression      /* less than */
  | Expression  <=   Expression      /* less than or equal to */
  | Expression  /\   Expression      /* conjunction */
  | Expression  \/   Expression      /* disjunction */
  | Expression  =>   Expression      /* implication */
  | Expression  <=>  Expression      /* if-and-only-if */

UnaryExpression ::=
    not Expression                   /* logical negation */
```

The above grammar consists of three rules, each of which with the form *R* **::=** *D*: the left-hand side of **::=** denotes the name of rule (e.g., *BinaryExpression*), whereas the right-hand side denotes the definition of rule. When a rule has multiple, alternative definitions, they are separated by a vertical bar (|).

Each italicized, capitalized word denotes a non-terminal (i.e., *Expression*, *BinaryExpression*, *UnaryExpression*). Each all-capital word (e.g., VALUE_CONSTANT), as well as a symbol or word in bold red, denotes a terminal (e.g., **not**, **(**, /\, *etc.*). The terminal VALUE_CONSTANT denotes any valid floating-point number (e.g., 3, 3.14, *etc.*). Texts enclosed within **/*** and ***/** denote comments that are not part of the grammar.

The above grammar can produce an infinite number of *syntactically-correct* strings. Terminals are base cases of the production, whereas non-terminal are recursive cases. For the purpose of this lab, we say that a given string is *syntactically correct* if that string can be produced by the above grammar. As examples:

- The string **3.14** can be produced (because *Expression* can be a VALUE_CONSTANT).
- The string **3.14 + 5.28** can be produced (because Expression can be *Expression* + *Expression*, and the left and right *Expression* can each produce a VALUE_CONSTANT). Here is the so called **abstract syntax tree** (AST) for this syntactically correct string:



- As exercises:
    - Why is **(23.4 * (34.04 + 28.9))** syntactically correct?
    - Why is **(23.4 * (not (34.04 + 28.9)))** also syntactically correct?
    - Why is **(23.4 * (not (34.04 + 28.9) (34.28 / 19.29)))** not syntactically correct?

In this lab, you are given a parser that is able to take as input a given string and determine if it is syntactically correct with respect to the above grammar. This parser is useful: we should not bother to analyze a random string which does not even conform to the expected syntax. If there is no syntax, then the input string is transformed into an **AST**, for which we implement using the **composite design pattern**, where the top deferred class is `EXPRESSION`. That is, a runtime object of type `EXPRESSION` has a tree structure similar to the above AST. See Figure 2.

Can we always evaluate a string that is *syntactically correct*? Maybe not. For example:

(23.4 * (not (34.04 + 28.9)))

conforms to the above grammar but it is not *type-correct*: **not** is the unary operator for logical negation, but its operand **(34.04 + 28.9)** is not a Boolean expression.

In this lab exercise, you are given working implementation of classes that implement the above CFG using the **composite design pattern**. Your tasks will be to implement the visitor design pattern to support the following operations that may be performed upon any string which conforms to the above grammar:

- Type Calculation
- Type Checking (which should make use of the type calculation)
- Evaluation

A working implementation for the `PRETTY_PRINTER` visitor is given to you as the starting reference.

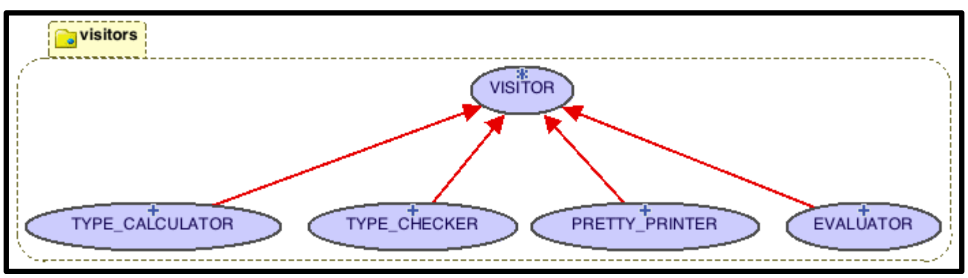
4 Design of Classes

Your starter project contains the following clusters:

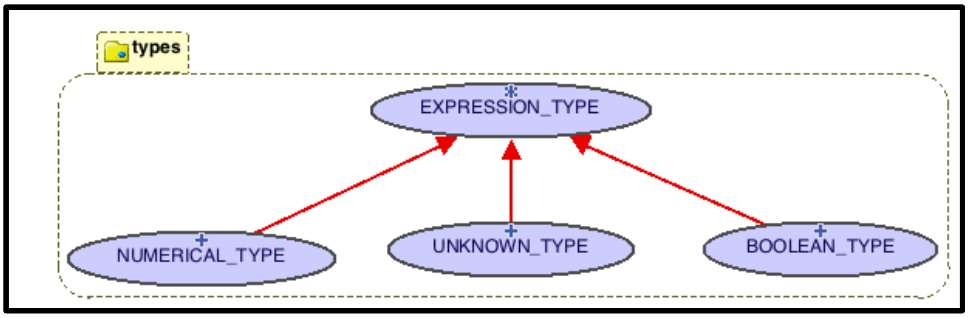
- `parsing`: classes in this cluster parse a given string and determine if it conforms to the above grammar of the small expression language. If the given string is syntactically correct, the parser constructs a `COMPOSITE` object (which is an AST of the given string). The parsing classes are provided, so you do not need to do the actual parsing.

- `composite`: classes in this cluster implement the composite design pattern for constructing objects that correspond to ASTs of string that are syntactically correct. **See Figure 2**. You will have to complete classes in this cluster, as these classes are used to represent simple and composite expressions.

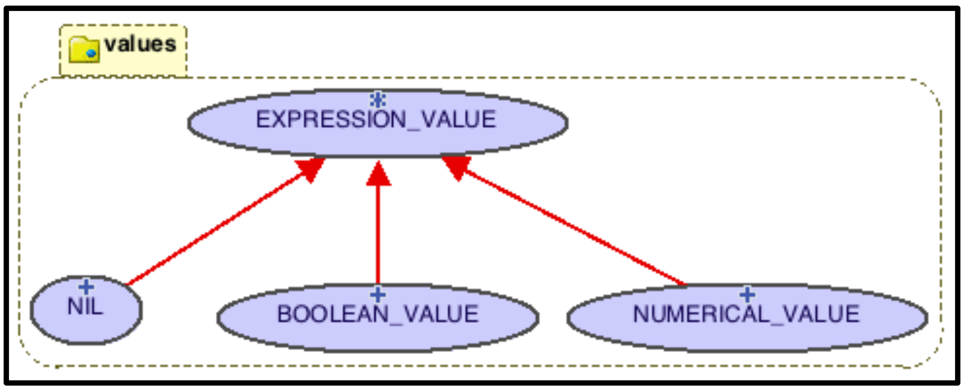
- `visitors`: classes in this cluster implement the visitor design pattern for performing various operations (pretty printing, evaluation, type calculation, and type checking) upon the composite objects.



- `types`: classes in this cluster should be used by the `TYPE_CALCULATOR` visitor class.



- `values`: classes in this cluster should be used by the `EVALUATOR` visitor class.



```
analyzer
├── analyzer.ecf
├── composite
│   ├── addition.e
│   ├── binary_expression.e
│   ├── composite.e
│   ├── conjunction.e
│   ├── disjunction.e
│   ├── division.e
│   ├── equals.e
│   ├── expression.e
│   ├── greater_than.e
│   ├── greater_than_or_equals.e
│   ├── iff.e
│   ├── implication.e
│   ├── less_than.e
│   ├── less_than_or_equals.e
│   ├── multiplication.e
│   ├── negation.e
│   ├── not_equals.e
│   ├── subtraction.e
│   ├── unary_expression.e
│   └── value_constant.e
├── parse
├── docs/
├── root
│   └── root.e
├── tests
│   ├── instructor
│   │   └── basic_tests.e
│   └── student
│       └── student_tests.e
├── types
│   ├── boolean_type.e
│   ├── expression_type.e
│   ├── numerical_type.e
│   └── unknown_type.e
├── values
│   ├── boolean_value.e
│   ├── expression_value.e
│   ├── nil.e
│   └── numerical_value.e
└── visitors
    ├── evaluator.e
    ├── pretty_printer.e
    ├── type_calculator.e
    ├── type_checker.e
    └── visitor.e
```

Figure 1 starter directory analyzer (you complete classes in red)

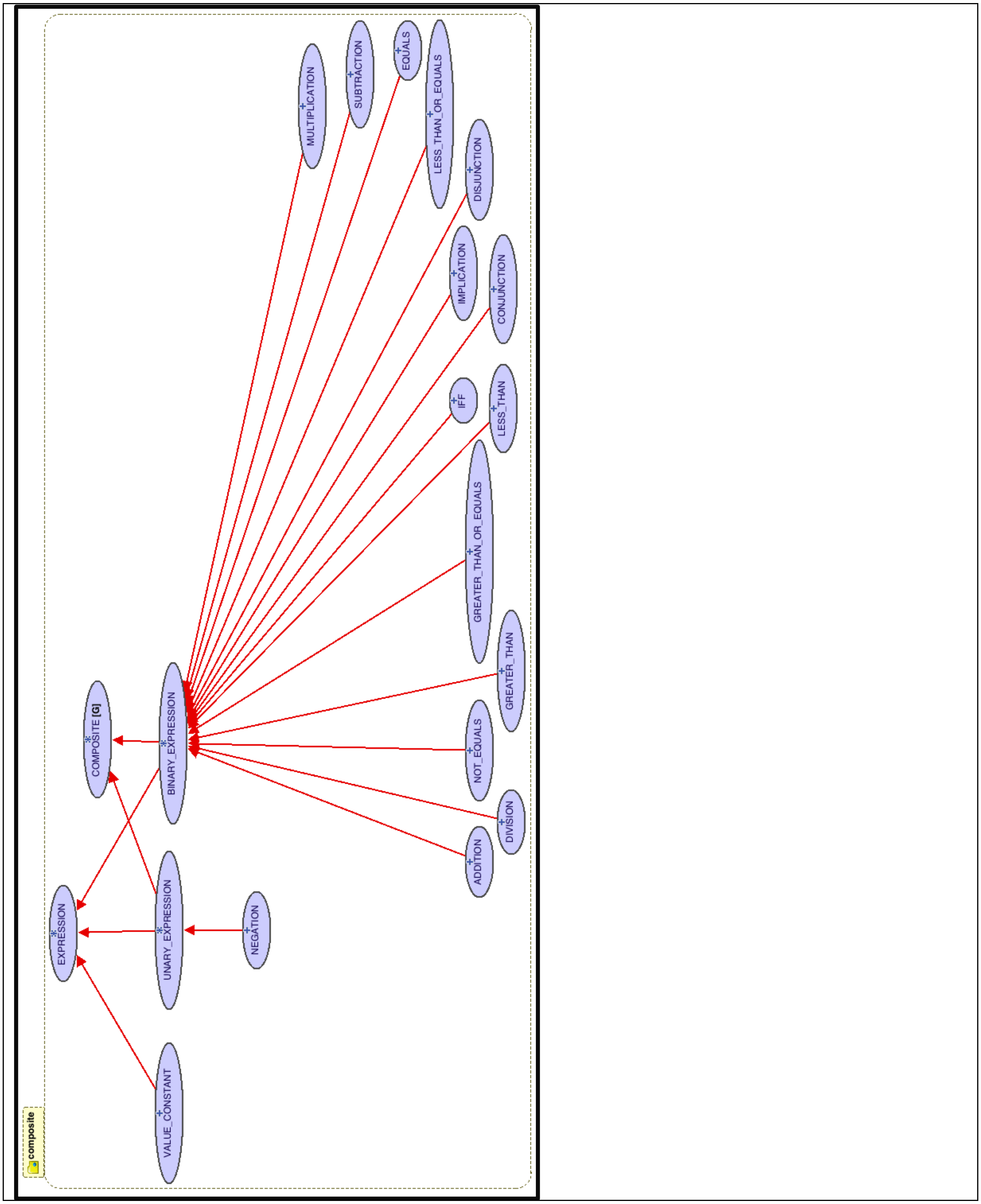


Figure 2 Expressions and Composite Expressions

5 Getting started

These instructions are for when you work on one of the EECS Linux Workstations or Servers (e.g *red*). You should not compile on *red* as it is a shared server; compile on an EECS workstation. See the course wiki for how to use the SEL-VM or your Laptop.

5.1 Retrieve Lab5

```
> ~sel/retrieve/3311/lab5
```

This will provide you with a starter directory `analyzer` with the project structure as shown in Figure 1. Classes in **bold red** are those that you are required to modify.

5.2 Compile the Lab

You can now compile the Lab.

```
> estudio analyzer/analyzer.ecf &
```

where `analyzer.ecf` is the Eiffel configuration file for this Lab, with the `ROOT` class:

```
note
    description: "gelex-calculator application root class"
    date: "$Date$"
    revision: "$Revision$"

class
    ROOT

inherit

    ES_SUITE

create
    make

feature {NONE} -- Initialization

    make
        do
            add_test (create {BASIC_TESTS}.make)
            show_browser
            run_espec
        end

end
```

This initial starter project given to you compiles. Running Workbench System will give you the following test report:

FAILED (13 failed & 2 passed out of 15)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	15
All Cases	2	15
State	**Contract Violation**	**Test Name**
Test1	**BASIC_TESTS**	
FAILED	NONE	test_01: print 3
FAILED	NONE	test_02: print ((3 + 4) * 45 >= 64) ∧ (4 /= 34) ∨ (not (5 = 65))
FAILED	Check assertion violated.	t_03: 3.23 evaluates to 3.23
FAILED	Check assertion violated.	t_04a: (3.23 + 4) * 3 evaluates to 21.69
FAILED	Check assertion violated.	t_04b: (3.23 + 4) * 3 = 21.69 evaluates to true
FAILED	NONE	test_05: 3.23 has numerical type
FAILED	NONE	test_06: 3.23 + 4 has numerical type
PASSED	NONE	test_07: (3 < 4) + (23 >= 34) has unknown type
FAILED	NONE	test_08: (3 < 4) => (23 >= 34) has boolean type
FAILED	NONE	test_09: (3 < 4) <=> (23 >= 34) has boolean type
FAILED	NONE	test_10: 3.23 is type correct
FAILED	NONE	test_11: 3.23 + 4 is type correct
PASSED	NONE	test_12: (3 < 4) + (23 >= 34) is not type correct
FAILED	NONE	test_13: (3 < 4) => (23 >= 34) is type correct
FAILED	NONE	test_14: (3 < 4) <=> (23 >= 34) is type correct

Note Two tests (`test_07` and `test_12`) are passed initially because the expected results (`unknown type` and `false`) happen to match the initial, default values of the corresponding `VISITOR` objects (`TYPE_CALCULATOR` and `TYPE_CHECKER`).

5.3 Understanding the API

Study carefully the tests given to you in the `BASIC_TESTS` class. For example:

```
test_02: BOOLEAN
    local
        v: PRETTY_PRINTER
    do
        comment ("test_02: print ((3 + 4) * 45 >= 64) /\ (4 /= 34) \/ (not (5 = 65))")
        parser.parse_string ("((3 + 4) * 45 >= 64) /\ (4 /= 34) \/ (not (5 = 65))")
        Result := parser.error_count = 0
        check Result end

        create v.make
        -- parser.expression returns a composite object of static type EXPRESSION
        -- In this case, its dynamic type is DISJUNCTION
        parser.expression.accept (v)
        Result := v.value ~ "(((((3+4)*45)>=64)/\(4/=34))\/(not(5=65)))"
    end
```

The call `parser.parse_string (…)` takes a string and determines if it is syntactically correct (it is not if `parser.error_count` is larger than zero). If there is no syntax error in the input string, then `parser.expression` returns an `EXPRESSION` object. To use the **visitor design pattern**, we call the `accept` feature on the `EXPRESSION` object:

`parser.expression.accept (v)`

Here you need to review how the **double dispatch** work in the visitor pattern, given that the dynamic type of `parser.expression` is `DISJUNCTION` and the dynamic type of `v` is `PRETTY_PRINTER`.

6 What You Must Do

Here is what you are required to complete (see the **To Do** tags):

- `composite` cluster
    - The deferred feature `accept` is declared in the `EXPRESSION` class. You are required to implement that feature in all its effective descendant classes.
- `visitors` cluster
    - Complete the classes here according to the **visitor design pattern** taught in class.
    - Note that:
        - A working implementation is given to you for the `PRETTY_PRINTER`.
        - For each of the three remaining visitors, an attribute `value` is declared with the type that corresponds to the visitor's expected operation:
            - For the `EVALUATOR` visitor, attribute `value` is with type `EXPRESSION_VALUE` (for the result of evaluation).

              For evaluating equality (=) between floating-point numbers, we have 0.001 tolerance. A helper query `is_equal_within` is included in this class for you to use.

            - For the `TYPE_CALCULATOR` visitor, attribute `value` is with type `EXPRESSION_TYPE` (for the result of type calculation).
            - For the `TYPE_CHECKER` visitor, attribute `value` is declared with type `BOOLEAN` (for whether or not the string is type-correct).

7 To Submit

Before submitting, please consult the course wiki for final details as well as when the submission machinery is available.

To obtain a passing grade, your submission must compile and pass the tests we provided.

We will be grading the correctness of your submission with additional tests.

The main requirements for submission are as follows.

1. Add correct implementations as specified.
2. Add your own contracts (precondition, postcondition, and invariant) when appropriate.
3. Work incrementally one feature at a time. Run all 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. do not just rely on our tests.
5. Don't make any changes to classes other than the ones specified.
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 *sorted-variants* again to remove all EIFGENs.

Submit your Lab from the command line as follows:

```
submit 3311 Lab5 analyzer
```

You will be provided with some feedback. Examine your feedback carefully. Submit often and as many times as you like.

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 your folder sorted-variants must be a superset of Figure 1.