EECS3311 Software Design Fall 2017
Project
An Analyzer for an Expression Language

Due: **12 NOON, Friday, December 1**

Abstract

This documents contains the requirements for you to complete the project. You are required to use the Eiffel Testing Framework (ETF) for this project. Before the due date, you are required to: **1)** submit a hard copy of the report to the EECS3311 dropbox; and **2)** make an electronic submission of your ETF project.

Contents

1 Policies

- **Your (submitted or unsubmitted) solution to this project (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., github) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.**
- You are required to **work as a team of two or three** for this project.
- When you submit your project, you claim that it is **solely** your team's work. Therefore, it is considered as **an violation of academic integrity** if you copy or share **any** parts of your Eiffel code during **any** stages of your development.
- When assessing your submission, the instructor and TA will examine your code, and suspicious submissions will be reported to the department if necessary. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- You are entirely responsible for making your submission in time. Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur.
- The deadline is **strict** with no excuses: you receive **0** for not making your submission in time.
- You are free to work on this lab anywhere, but you are advised to test your code via your EECS account before the submission.

2 Required Readings

- Tutorial on ETF: a Bank Application
- Starting Page on ETF
- Tutorial Videos on ETF (Installation, Launching Default Project, and Error Reporting and Debugging)

  Caveats:

  - These tutorial videos were made prior to the latest release of ETF. The majority of the technical details discussed in the videos still apply. In case of doubts (about possibly obsolete features), report to the instructor and clarifications will be added to the amendments section.
  - In the video "ETF: Eiffel Testing Framework (01) Installation", it mentions about the executables of ETF generator for Windows and MacOS X. We no longer distribute these executables. You can get access to the ETF generator via either the Prism Lab machines, or the virtual box image.
  - The three sample files mentioned in the tutorial videos (`definitions.txt`, `input.txt` `input_bad.txt`) are no longer available. Use the files mentioned in the bank example in the tutorial document.

- This tutorial video shows you how to build and connect a sample business model to the generated ETF project.

3 Working as a Group of Two or Three

You **must** work with one or two partners. That is, only teams with two or three members are allowed.

4 Working from Home

- To generate the ETF project, you must use the `etf` command that is available on either the Prism Lab machines or the virtual box image.
- For a generated ETF project to compile on your machine, you need to first download a library called `MATHMODELS` (available as a zip from the **course moodle** page), and then set a environment variable `MATHMODELS` which points to the location of its download.

  This tutorial video walks you through the process.

5 Problems

In this project, you are asked to complete, under the Eiffel Testing Framework (ETF), the design and implementation of: **1)** a small expression language; and **2)** its three associated functionalities (i.e., pretty printing, type checking, and evaluation).

5.1 An Expression Language

The context-free grammar of the small expression language that you will design and implement is defined as follows:

Expression	::=	*IntegerConstant*
	|	*BooleanConstant*
	|	**LPAREN** *BinaryOp* **RPAREN**
	|	**LPAREN** *UnaryOp* **RPAREN**
	|	*SetEnumeration*
IntegerConstant	::=	**DIGIT**$^+$
BooleanConstant	::=	**TRUE**
	|	**FALSE**
BinaryOp	::=	*Expression* **PLUS** *Expression*
	|	*Expression* **MINUS** *Expression*
	|	*Expression* **TIMES** *Expression*
	|	*Expression* **DIVIDES** *Expression*
	|	*Expression* **AND** *Expression*
	|	*Expression* **OR** *Expression*
	|	*Expression* **IMPLIES** *Expression*
	|	*Expression* **EQUAL** *Expression*
	|	*Expression* **GT** *Expression*
	|	*Expression* **LT** *Expression*
	|	*Expression* **UNION** *Expression*
	|	*Expression* **INTERSECT** *Expression*
	|	*Expression* **DIFFERENCE** *Expression*
UnaryOp	::=	**NEGATIVE** *Expression*
	|	**NEGATION** *Expression*
	|	**SUM** *Expression*
	|	**FORALL** *Expression*
	|	**EXISTS** *Expression*
SetEnumeration	::=	**LBRACE** *Expression* (**COMMA** *Expression*)* **RBRACE**

Italic words that start with a capital letter such as *Expression* are *non-terminals.* We use $*$ and $+$ to denote, respectively, zero-or-more and one-or-more repetitions of a non-terminal or a terminal. The use of vertical bars (`|`) allows us to specify alternatives of a non-terminal. All-capital words are *terminals.* Table 1 defines the corresponding ASCII character(s) and meaning for each terminal.

Here are some syntactically correct expressions that can be produced by the above grammar:

- `1`
- `(- 10)`
- `(2 + 4)`
- `{1, 2}`

Terminal	ASCII Character(s)	Meaning
LPAREN	`(`	starting delimiter for an expression
LPAREN	`)`	ending delimiter for an expression
LBRACE	`{`	starting delimiter for a set enumeration
RBRACE	`}`	ending delimiter for a set enumeration
COMMA	`,`	delimiter between two members in a set enumeration
DIGIT	`0` .. `9`	numerical digit
TRUE	`True`	
FALSE	`False`	
PLUS	`+`	addition
MINUS	`-`	subtraction
TIMES	`*`	multiplication
DIVIES	`/`	division, which returns the integer quotient
AND	`&&`	logical conjunction
OR	`||`	logical disjunction
IMPLIES	`=>`	logical implication
EQUAL	`=`	equality
GT	`>`	greater than
LT	`<`	less than
UNION	`\/`	set union
INTERSECT	`/\`	set intersection
DIFFERENCE	`\`	set difference
NEGATIVE	`-`	negative sign
NEGATION	`!`	logical negation
SUM	`+`	summation (over a set of integer expressions)
FORALL	`&&`	generalized conjunction (over a set of Boolean expressions)
EXISTS	`||`	generalized disjunction (over a set of Boolean expressions)

Table 1: Terminals: ASCII Representations and Meanings

- `(+ {1, 2})`
- `({True, False} \/ {2+3, 4})`

Notice that empty set enumeration (i.e., `{}`) is **not** allowed by the grammar. Of course, the above grammar can produce expressions that we would consider as:

- Not type-correct, e.g. `({True, False} \/ {2+3, 4})`
- Not evaluable, e.g., `(7 / 0)`

Task 1

Your first task is to design and implement Eiffel classes and features to represent this expression language.

5.2 Operations on the Expression Language

Task 2

Your second task is to design and implement Eiffel classes and features to support three operations on the expression language: **1)** pretty printing; **2)** type checking; and **3)** evaluation.

5.2.1 Pretty Printing

Given an expression that is syntactically correct, with respect to the above grammar, your implemented pretty printer should write it out as a string. Pay attention to the following details:

- Each unary or binary operation is surrounded by a matching pair of round parentheses
- Each set enumeration is not surrounded by a matching pair of round parentheses.
- Each unary operator is followed by a single space.
- Each binary operator is preceded by a single space and followed by a single space.

For example: `(- 10)`, `(+ {1})`, `({True} \ {False})`, *etc.*

5.2.2 Type Checking

Given an expression that is syntactically correct, with respect to the above grammar, your implemented type checker should decide whether or not it is type-correct. An expression is considered as type-correct if and only if each (unary or binary) operator is applied to operand(s) of the right type.

Here are example expressions that are both syntactically correct and type-correct:

- `(1 + 2)`
- `(+ {1, 2, 3})`
- `(&& {True, False})`
- `(|| {True, False})`
- `({1, 2, 3} \/ {2})`
- `({1, 2, 3} \ {2})`

Here are example expressions that are syntactically correct but not type-correct:

- `(1 + True)`
- `(3 + {1, 2, 3})`
- `(+ {1, 2, True})`
- `(&& {1, 2})`
- `(|| True)`
- `({1, 2, 3} \/ {True, False, True})`

5.2.3 Evaluation

Given an expression that is both syntactically and type-correct, with respect to the above grammar, your implemented evaluator should aim to simplify the expression by evaluating parts that can be evaluated.

Here are example of evaluating type-correct expressions:

- `(1 + 2)` evaluates to `3`
- `(+ {1, 2, 3})` evaluates to `6`
- `(&& {True, False})` evaluates to `False` [generalized and]
- `(|| {True, False})` evaluates to `True` [generalized or]
- `({1, 2, 3} \/ {2})` evaluates to `{1, 2, 3}`

- `({1, 2, 3} \ {2})` evaluates to `{1, 3}`
- `{1, 2, 3, 4}` evaluates to `{1, 2, 3, 4}`

In the above last example, there are no members in the set enumeration that can be further simplified, so the evaluation result is simply the original set enumeration.

6 Abstract User Interface

Customers need not know details of your design of classes and features. Instead, there is an agreed interface for customers to specify the expressions they wish to print, type check, and evaluate. This is why we are using ETF: customers only need to be familiar with the list of *events*, defined in the plain text file below that is used to generate the customized ETF for your project. We assume the following abstract user interface:

```
system analyzer

-- Events of users requesting for processing the expression (entered so far)
type_check
evaluate
reset

-- Events of users adding constants
boolean_constant (c: BOOLEAN)
integer_constant (c: INTEGER)

-- Events of users adding binary arithmetic operations
add
subtract
multiply
divide

-- Events of users adding binary logical operations
conjoin
disjoin
imply

-- Events of users adding binary relational operations
equals
greater_than
less_than

-- Events of users adding binary set operations
union
intersect
difference

-- Events of users adding unary arithmetic operations
negative

-- Events of users adding unary logical operations
negation

-- Events of users adding unary composite operations
sum
```

```
for_all
there_exists

-- Events of users starting/finishing entering set enumerations
open_set_enumeration
close_set_enumeration
```

We also assume that given a syntactically correct expression, customers know about its corresponding parse tree. For example, the parse tree for the expression `((1 + 2) * 3)` looks like:

```
        MULTIPLY
       +---^---+
      ADD      3
   +---^---+
   1       2
```

Notice that delimiters (i.e., left and right parentheses) are not necessary to be included in the parse tree, but they are useful in disambiguating the order of evaluation.

Assuming that customers have the above parse tree in mind, they will specify the expression, using the agreed abstract user interface, by gradually specifying sub-expressions via a *pre-order* traversal of the tree[1]:

```
multiply
add
integer_constant (1)
integer_constant (2)
integer_constant (3)
```

In-between the events, since the expression being specified is not completed yet, your software must warn the users that both `type_check` and `evaluate` are not allowed. On the other hand, after all 5 events above have occurred, users would expect: **1)** that the occurrence of event `type_check` reports that the specified expression is type-correct; and **2)** that the occurrence of event `evaluate` reports that the result is `9`. All these small details are related to how your software should inform users of the current state of the analyzer, before and after the occurrence of each event. We will discuss how you should display the state of your analyzer in the next section.

7 Outputting the Abstract State

For the purpose of using your implemented analyzer, users need to be informed of: **1)** the expression currently being specified; and **2)** whether the last event that occurred was a success, or resulted in an error. These two pieces of information constitute the *abstract*[2] state of your analyzer.

As an example, consider the following use case where the user attempts to type check and evaluate the expression `((1 + 2) * 3)`:

```
  Expression currently specified: ?
  Report: Expression is initialized.
->multiply
  Expression currently specified: (? * nil)
  Report: OK.
->add
  Expression currently specified: ((? + nil) * nil)
```

[1]A pre-order traversal of the tree first visits the root, then recursively visits the left subtree, and then recursively visits the right subtree.

[2]The term "abstract" here suggests that we show only the relevant information to users, by filtering out all other (implementation-related) details of your software.

```
  Report: OK.
->integer_constant(1)
  Expression currently specified: ((1 + ?) * nil)
  Report: OK.
->type_check
  Expression currently specified: ((1 + ?) * nil)
  Report: Error (Expression is not yet fully specified).
->integer_constant(2)
  Expression currently specified: ((1 + 2) * ?)
  Report: OK.
->integer_constant(3)
  Expression currently specified: ((1 + 2) * 3)
  Report: OK.
->type_check
  Expression currently specified: ((1 + 2) * 3)
  Report: ((1 + 2) * 3) is type-correct.
->evaluate
  Expression currently specified: ((1 + 2) * 3)
  Report: 9
```

In the above expected use case of your analyzer, the occurrence of each event is preceded by "dash-greater-than" (`->`). The initial state of the analyzer (before the first event occurs) is one where the expression object is initialized. After the occurrence of each event, your software is also expected to display its post-state. Observe that for any two consecutive event occurrences, the post-state of the earlier event occurrence is at the same time the pre-state of the later event occurrence.

The string representation of the abstract state consists of two lines:

- The first line starts with the header "Expression currently specified: " and is followed by the pretty-printing of the expression that has so far been input by the user.

  The pretty printing of the expression should display both operators (see Table 1) and their operands. For operands that have not yet been specified, your software should print each one of them as `nil`. However, as a special case, to help users keep track of their progress, the next (sub-)expression that your software expects to be entered, as far as the assumed pre-order is concerned, should be printed as `?` instead. That is, at any one time, the pretty printing of the expression contains at most one `?`, but may contain multiple `nil`'s.

- The second line starts with the header "Report: " and is followed by a message of either: **1)** a success, if the event occurrence that resulted in the current state was a *legitimate* one; or **2)** an error. Table 2 summarizes the list of messages that your software, when appropriate, must report.

  All possible errors should be reflected as feature preconditions of your analyzer (in the *model* cluster). However, reporting the violations of these preconditions (as errors) must be done on the side of the abstract user interface (i.e., the corresponding descendant class of *ETF_COMMAND*). When an error occurs, only the corresponding error message is reported back to the user, whereas the state of your analyzer *must* remain unchanged.

8 Getting Started

First of all, make sure you have already acquired the basic knowledge about the Eiffel Testing Framework (ETF) as detailed in Section 2.

Download the **`analyzer.zip`** file from the course moodle page and unzip it. The text file `analyzer_events.txt` is for you to generate the ETF project for your analyzer application. The five input files (e.g., `at1.txt`,

Message	Context
`Expression is initialized.`	When the analyzer is first started.
`OK.`	When the analyzer is successfully reset, or when the last expression input by the user was a success.
`Error (Expression is not yet fully specified).`	When the user attempts to type check or evaluate, but the expression being specified has not yet been completed.
`Error (Expression is already fully specified).`	When the user attempts to add a new sub-expression, or to close a set enumeration, but the expression being specified has been completed already.
`Error (Expression is not type-correct).`	When the user attempts to evaluate the expression that has been completely specified but is not type-correct.
`Error (Initial expression cannot be reset).`	When the user attempts to reset the expression immediately after it is initialized or re-initialized.
`Error (Divisor is zero).`	When the user attempts to evaluate a division where the divisor is zero.
`Error (Set enumeration is not being specified).`	When the user attempts to close a set enumeration, but there is currently not a pending set enumeration.
`Error: (Set enumeration must be non-empty).`	When the user attempts to close a pending set enumeration, but no member expressions have been specified for that set enumeration.

Table 2: Messages: String Values and When to Report Them

`at2.txt`, *etc.*) are **simple** use cases for you to test your software. The five expected output files (e.g., `at1.expected.txt`, `at2.expected.txt`, *etc.*) contain outputs that your software must produce to match. You are advised to, before start coding, study the given expected output files carefully, in order to obtain certain reasonable sense of how your analyzer is supposed to behaviour.

All your development will go into this downloaded directory, and when you make the submission, you must submit this directory. To begin your development, follow these steps:

1. Open a new command-line terminal. Change the current directory into this downloaded directory, type the following command to generate the ETF project:

   ```
   etf -new analyzer_events.txt .
   ```

   Notice that there is a dot (`.`) at the end to denote the current directory.

2. There are two `analyzer` directories: one is the top-level directory that contains all generated ETF code and your development; the other is the sub-directory that contains the code of your model of analyzer. **When you submit, make sure that you submit the top-level `analyzer` directory.**

3. Open the generated project in Eiffel Studio by typing:

   ```
   estudio17.05 analyzer.ecf &
   ```

4. Once the generated project compiles successfully in Eiffel Studio, go to the *ROOT* class in the *root* cluster. Change the implementation of the *switch* feature as:

```
switch: INTEGER
      -- Running mode of ETF application.
   do
      Result := etf_cl_show_history
   end
```

This overrides the default GUI mode of the generated ETF. To make it take effect, re-compile the project in Eiffel Studio.

5. Switch back to the terminal and type the following command:

   ```
   EIFGENs/analyzer/W_code/analyzer
   ```

   Then you should see this output (rather than launching the default GUI of ETF):

   ```
   Error: a mode is not specified
   Run 'EIFGENs/analyzer/W_code/analyzer -help' to see more details
   ```

6. As you develop your ETF project for the analyzer, launch the batch mode of the executable. For example:

   ```
   EIFGENs/analyzer/W_code/analyzer -b at1.txt
   ```

   This prints the output to the terminal. To redirect the output to a file, type:

   ```
   EIFGENs/analyzer/W_code/analyzer -b at1.txt at1.actual.txt
   ```

   The `at1.actual.txt` file stores the *actual* output from your current software, and your goal is to make sure that `at1.actual.txt` is identical to `at1.expected.txt` by typing:

   ```
   diff at1.expected.txt at1.actual.txt
   ```

   Of course, the actual output file produced by the default project is far from being identical to the expected output file.

7. You should first aim to have your software produce outputs that are identical to those of the five expected output files (i.e., `at1.expected.txt`, `at2.expected.txt`, *etc.*).

8. Then, as you develop further for your ETF project, create as many acceptance test files of your own as possible. Examine the outputs and make sure that they are consistent with the requirements as stated in this document.

9. About one week before the project due date, you will be given an *oracle* program for you to test if your software and the oracle produce identical outputs on all of your acceptance test files. You may want to write a shell script program to automate the workflow of generating and comparing outputs. **You certainly want to finish all your development before the oracle program is made available to you, so that if you find any inconsistencies of outputs, you still have sufficient time to debug and fix.**

9 Modification of the Cluster Structure

You must not change signatures of any of the classes or features that are generated by the ETF tool. You may only add your own clusters or classes to the *model* cluster as you consider necessary. However, when you add a new cluster, it is absolutely critical for you to make sure that a **relative path** (i.e., a path that is relative to the current project directory `.` and does not start with `/`) is specified to add that cluster in the project setting. **Specifying an absolute path in your project will make your submitted project fail to compile when being graded, and this will result in an immediate zero for your marks with no excuses.** So please, make sure you pay extra attention to all clusters that you add to the project.

10 Submission

10.1 Checklist before Submission

1. Similar to the assignment, put a `team.txt` file in the `docs` directory by including the CSE login names of yourself, and of your team parter(s) (if you choose to work as a group of two or three). Here is an example of the contents of `team.txt` (with two members in the team):

```
cse123456
cse654321
```

2. Make sure the *ROOT* class in the *root* cluster has its *switch* feature defined as:

```
switch: INTEGER
      -- Running mode of ETF application.
   do
      Result := etf_cl_show_history
   end
```

3. Compile and print off a report including:

   - A cover page that clearly indicates: 1) course; 2) semester; 3) names; 4) CSE logins of the team member(s); and 5) CSE login of the submitting account;
   - Three BON diagrams for your design, which should contain exactly 3 pages:
     - Page 1 details the relationships between all relevant classes. All classes are shown in the concise view.
     - Page 2 details the architecture of your design that models the expression language structure. You must show the critical class(es) in the expanded view.
     - Page 3 details the architecture of your design that models the three language operations. You must show the critical class(es) in the expanded view.
   - With no page limit, explain in details how your design (not the design of the generated ETF project) for the analyzer obeys the following design principles:
     - Information Hiding (what is hidden and may be changed? what is not hidden and stable?)
     - Single Choice Principle
     - Open-Close Principle
     - Uniform Access Principle
   - You must also include the draw.io XML source file of your bon diagram and its exported PDF in the `docs` directory when you make your electronic submission. **If the TA cannot find, in the *docs* directory, the draw.io XML source and PDF files of your BON diagrams, you will immediately lose 50% of your marks for that part of the project.**

10.2 Submitting Your Work

Both hard-copy and electronic submissions are required.

- **Hard-Copy Submission**

  1. By the due date, drop the print-out of the report into the EECS3311 dropbox. **You will receive zero mark for the report if the TA cannot collect it from the dropbox.**

- **Electronic Submission**

  1. You are expected to submit from a Prism lab terminal.
  2. Each team must make their submission from only a single CSE account.
  3. There are two `analyzer` directories: one is the top-level directory that contains all generated ETF code and your development; the other is the sub-directory that contains the code of your model of analyzer. **When you submit, make sure you submit the top-level `analyzer` directory.**
  4. Go to the directory containing the top-level `analyzer` project directory:

     4.1 Run the following command to remove the `EIFGENs` directory:

     ```
     eclean analyzer
     ```

     4.2 Run the following command to make your submission:

     ```
     submit 3311 project analyzer
     ```

     A check program will be run on your submission to make sure that you pass the basic checks (e.g., the code compiles, passes the given tests, *etc*). After the check is completed, feedback will be printed on the terminal, or you can type the following command to see your feedback:

     ```
     feedback 3311 project
     ```

     In case the check feedback tells you that your submitted project has errors, you must fix them and re-submit. Therefore, you may submit for as many times as you want before the submission deadline, to at least make sure that you pass all basic checks.

     Note. You will receive zero for submitting a project that cannot be compiled.

11 Questions

There might be unclarity, typos, or even errors in this document. It is **your responsibility** to bring them up, early enough, for discussion. Questions related to the project requirements are expected to be posted on the on-line course forum. It is also **your responsibility** to frequently check the forum for any clarifications/changes on the project requirements.

12 Amendments