EECS3311 Software Design Fall 2019
Project
Building an Analyzer for a Simple Programming Language

Chen-Wei Wang

Due: **11:59PM, Wednesday, December 4**

This project has similarity with a previous project.
It is considered as violating academic honesty if you use code posted online by students who completed this course. We will run an automated check on your final submission; do not take chances.

1 Policies

- **Your (submitted or un-submitted) solution to this lab exercise (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., a non-private Github repository) 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 1 or 2** for this lab. Group members may be from different sections.
- When you submit your lab, you claim that it is **solely** the work of your group. 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 may 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. Follow this tutorial series on setting up a **private** Github repository for your Eiffel projects.
- The deadline is **strict** with no excuses: you receive **0** for not making your electronic submission in time. Emailing your solutions to the instruction or TAs will not be acceptable.
- You are free to work on this lab on your own machine(s), but you are responsible for testing your code at a Prims lab machine before the submission.

Contents

2 Required Readings

- **Tutorial Videos on ETF**

  We no longer distribute executables of the `etf` tool (for generating a starter project). You can get access to the ETF generator via either the Prism Lab machines, or the virtual box image.

- Tutorial on ETF: a Bank Application
- You may find the composite/visitor patterns useful, for which there is a tutorial: `https://www.youtube.com/playlist?list=PL5dxAmCmjv_4z5eXGW-ZBgsS2WZTyBHY2`.
- You can also find an abundance of resources on DbC, TDD, ESpec tests, and Eiffel code examples from these two sites:
  - `http://eiffel.eecs.yorku.ca`
  - `https://wiki.eecs.yorku.ca/project/eiffel/`

3 Working as a Group of One or Two

You **must** work either alone or with one parter. That is, only teams with one or two members are allowed. Members of a team may come from different sections.

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.

5 Grading Criterion of this Project

- When grading your submission (separate from the report), your ETF project will be compiled and built from scratch, and then executed on a number of acceptance tests (similar to `at01.txt`, `at02.txt`, ... given to you).
- Each acceptance test is considered as **passing** **only if** the output generated by your program is character-by-character identical to that generated by the *oracle*. No partial marks will be given to a test case even if the output difference is as small as a single character.
- It is therefore critical for you to always switch to the command-line mode of your ETF project and use either `diff` or `meld` to compare its output and that of the *oracle*.

6 Problems

In this project, you are asked to complete, under the Eiffel Testing Framework (ETF), the design and implementation of: **1)** a small programming language; and **2)** its two associated functionalities (i.e., `Java`-like code generation and type checking).

In Section 6.1 and Section 6.2, we formulate the programming and expression languages using context-free grammars. We adopt the following conventions:

- Italic words that start with a capital letter such as *Program* and *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.
- Words or symbols coloured in red and bold faces are *terminals*. Table 1 (p4) defines the corresponding ASCII character(s) and meaning for each terminal.

ASCII Character(s)	Meaning
`class`	starting keyword for a class declaration
`int`, `boolean`	pre-defined primitive types
`void`	return type of a command
`(`	starting delimiter for: **1)** declarations of parameters of a routine; or **2)** an expression
`)`	ending delimiter for: **1)** declarations of parameters of a routine; or **2)** an expression
{	starting delimiter for a class declaration or a routine implementation
}	ending delimiter for a class declaration or a routine implementation
`,`	delimiter between two parameters in a routine header
`;`	delimiter between two assignments in a routine implementation
`0` .. `9`	numerical digit
`True`	boolean constant
`False`	boolean constant
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division, which returns the integer quotient
`%`	modulo, which returns the integer remainder
`&&`	logical conjunction
`||`	logical disjunction
`==`	equality
`=`	variable assignment
`>`	greater than

`<`
`-`
	less than
	negative sign

`!`	logical negation

Table 1: Terminals: ASCII Representations and Meanings

6.1 Language of Classes and Features

Figure 1 shows the context-free grammar of the small Java-like programming language that you will design and implement. The non-terminal *Expression* is discussed separately in Section 6.2.
A **program** consists of a (possibly empty) list of class declarations. Each **class declaration** includes the name of a class and, enclosed in a matching pair of curly brackets ({ }), a (possibly empty) list of feature

```
Program                ::=  ClassDeclaration*

ClassDeclaration       ::=  class  Name  {
                                (AttributeDeclaration | RoutineDeclaration)*
                            }

AttributeDeclaration   ::=  Type  Name ;

RoutineDeclaration     ::=  Type  Name  Parameters  {
                                Assignment*
                            }
Parameters             ::=  ( )
                        |   ( Type  Name  (,  Type  Name)* )

Assignment             ::=  Name  =  Expression ;
Type                   ::=  int | boolean | void | Name
```

Figure 1: Context-Free Grammar: Programs (Classes and Features)

declarations. A feature declaration can either be an attribute declaration or a routine declaration. An **attribute declaration** includes the name of an attribute with its name and type. A **routine declaration** declares either a query (a.k.a. accessor/getter) or a command (a.k.a. mutator/setter), including the name of a routine, a (possibly empty) list of comma-separated parameter declarations enclosed in a matching pair of round parentheses (`( )`), a return type, and a (possibly empty) list of semi-colon-separated variable assignments as its implementation, enclosed in a matching pair of curly brackets ({ }). A **parameter declaration** includes the name and type of an input passed to the query or command in question.

Here is an example (see also `at09.txt` and `at10.txt` and their expected output files) of a syntactically correct (but not necessarily type-correct) program that can be produced by the above grammar in Figure 1:

```
class A {
  B b;
  A q1(int i) {
    A Result = null;
    Result = b.a;
    return Result;
  }
}
class B {
  A a;
  void c1() {
    a = b.a;
  }
}
```

Are both occurrences of the attribute call chain `b.a` in the above example type-correct? The first occurrence of `b.a` (in the context of class `A` and query `q1`) is type-correct because `b` is an existing attribute in the context class `A`, and `a` is an existing attribute in `b`'s type (i.e., `B`). On the other hand, the second occurrence of `b.a` (in the context of class `B` and command `c1`) is not type-correct because `b` is not an existing attribute in the context class `B`.

Question. When is an attribute call chain `a1.a2.....`$\mathtt{a}_n$ type-correct in general? From left to right, attribute `a1` exists in the current context class of the routine in question, attribute `a2` exists in `a1`'s type, attribute `a3` exists in `a2`'s type, ..., and $\mathtt{a}_n$ exists in $\mathtt{a}_{n-1}$'s type.

The RHS (right-hand side, or source) of a variable assignment is defined by the grammar of an *Expression* (discussed in Section 6.2).

6.2 Language of Expressions

Figure 2 shows the context-free grammar of the small expression language that you will design and implement. An expression is specified only as the RHS of some variable assignment (in the context of some routine and class).

$$
\begin{array}{lcl}
\mathit{Expression} & ::= & \mathit{IntegerConstant} \\
 & | & \mathit{BooleanConstant} \\
 & | & \textbf{(}\ \mathit{BinaryOp}\ \textbf{)} \\
 & | & \textbf{(}\ \mathit{UnaryOp}\ \textbf{)} \\
 & | & \mathit{CallChain} \\
 & & \\
\mathit{IntegerConstant} & ::= & (\,\textbf{1} \mid \textbf{2} \mid \textbf{3} \mid \textbf{4} \mid \textbf{5} \mid \textbf{6} \mid \textbf{7} \mid \textbf{8} \mid \textbf{9})(\textbf{0} \mid \textbf{1} \mid \textbf{2} \mid \textbf{3} \mid \textbf{4} \mid \textbf{5} \mid \textbf{6} \mid \textbf{7} \mid \textbf{8} \mid \textbf{9})* \\
 & & \\
\mathit{BooleanConstant} & ::= & \textbf{True} \\
 & | & \textbf{False} \\
 & & \\
\mathit{BinaryOp} & ::= & \mathit{Expression} + \mathit{Expression} \\
 & | & \mathit{Expression}\ \textbf{-}\ \mathit{Expression} \\
 & | & \mathit{Expression}\ \textbf{*}\ \mathit{Expression} \\
 & | & \mathit{Expression}\ /\ \mathit{Expression} \\
 & | & \mathit{Expression}\ \%\ \mathit{Expression} \\
 & | & \mathit{Expression}\ \&\&\ \mathit{Expression} \\
 & | & \mathit{Expression}\ \textbf{||}\ \mathit{Expression} \\
 & | & \mathit{Expression} == \mathit{Expression} \\
 & | & \mathit{Expression} > \mathit{Expression} \\
 & | & \mathit{Expression} < \mathit{Expression} \\
 & & \\
\mathit{UnaryOp} & ::= & \textbf{-}\ \mathit{Expression} \\
 & | & \textbf{!}\ \mathit{Expression} \\
 & & \\
\mathit{CallChain} & ::= & \mathit{Name}(\textbf{.}\mathit{Name})*
\end{array}
$$

Figure 2: Context-Free Grammar: Expressions (used as the RHS of assignments)

Here are some syntactically correct (not necessarily type-correct) expressions that can be produced by the above grammar in Figure 2:

- `123`
- `True`
- `(- 10)`
- `(2 + 4)`
- `(((2 + 4) < 3) && (! (78 % False)))`
- `b.a`

Of course, grammar in Figure 2 can produce expressions that we would consider as:

- Type-correct, e.g., `(2 + 4)`
- Not type-correct, e.g. `((2 + 4) < 3) && !(78 % False)`

6.3 Task 1: Design of Language Structure

Your first task is to design and implement Eiffel classes and features (in the `model` cluster) to represent the above programming and expression languages.

6.4 Task 2: Design of Language Operations

Your second task is to design and implement Eiffel classes and features (in the `model` cluster) to support two operations on the programming and expression languages: **1)** pretty printing; and **2)** analysis.

6.4.1 Pretty Printing

Given a program that is syntactically correct, with respect to the above grammar, your implemented pretty printer should output it as a Java-like program. Pay attention to the following details:

- Each unary or binary operation is 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 examples:

- `(- 10)`
- `(2 + 4)`
- `(((2 + 4) < 3) && (! (78 % False)))`

Again, each query has a reserved variable `Result` (case-sensitive) whose type corresponds to the query's return type. For each query, users of your tool do not specify via some ETF command/event: **1)** declaration of the reserved variable `Result`; **2)** initialization of `Result` to the appropriate default value (i.e., 0 for integers, false for boolean, and null for reference type); and **3)** `return` of `Result`. Instead, your tool, when generating code, is required to generate **1)** and **2)** as always the first and last lines, respectively, of the query's implementation. Therefore, the `return` statement is not included in the above grammar in Figure 1 (p5).

6.4.2 Analysis (Type Checking)

Given a program that is syntactically correct, with respect to the above grammars in Figure 1 and Figure 2, your implemented tool should decide whether or not each of the variables assignments (in the context of some routine and some class) is type-correct. A variable assignment is type-correct if the type of its LHS matches that of its RHS:

- The LHS of a variable assignment may either refer to an existing attribute of the current class, or the reserved variable `Result` if the current routine in question is a query.
- It is considered as type-correct if the RHS of a variable assignment refers to parameter(s) of the routine in question.
- An expression (which appears as the RHS of a variable assignment) is considered as type-correct if and only if each (unary or binary) operator is applied to operand(s) of the right type.

For example, the following program is type-correct:

```
class A {
  int i;
  int q1(B b) {
    int Result = 0;
    Result = b.a.i;
    return Result;
  }
}
class B {
  A a;
}
```

7 Assumptions

For the simplicity of this project, we assume that:

1. A **type** can be either of the following: `INTEGER` for integers, `BOOLEAN` for boolean values, and the name of some declared class for referene values. When users invoke ETF events/commands from an acceptance test file (e.g,. `at01.txt`), they write `"INTEGER"` and `"BOOLEAN"`, but when your tool generates Java-like code, it translate those types to, respectively, `"int"` and `"boolean"`.

2. Each query has a reserved variable `Result` (case-sensitive) whose type corresponds to the query's return type. For each query, users of your tool do not specify via some ETF command/event: **1)** declaration of the reserved variable `Result`; **2)** initialization of `Result` to the appropriate default value (i.e., 0 for integers, false for boolean, and null for reference type); and **3)** `return` of `Result`. Instead, your tool, when generating code, is required to generate **1)** and **2)** as always the first and last lines, respectively, of the query's implementation. Therefore, the `return` statement is not included in the above grammar in Figure 1 (p5).

3. A **variable assignment** (in the context of a routine and a class) has its LHS (left-hand side, or target) being either the name of an attribute in the current class, or a reserved variable `Result` if the routine in question is a query. That is, a parameter variable, if any, cannot appear as the LHS of a variable assignment.

4. Users of your tool do not choose `INTEGER` (a pre-defined primitive type), `BOOLEAN` (a pre-defined primitive type), or `Result` (which is reserved for variable assignments in queries) for the name of a class, a feature (attribute, query, or command), or a parameter.

5. Users of your tool do not have any use of `Result` in a command. For example, you can assume that programs like below will not be specified by users (or tested on your tool):

```
class Account {
  int balance;
  void set_balance(int amount) {
    int Result = amount;
    balance = Result;
  }
}
```

8 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 programs they wish to print (Java-like code) and analyze. 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

type VAR_NAME     = STRING -- variable names are strings
type FEATURE_NAME = STRING -- feature (attribute, query, command) names are strings
type CLASS_NAME   = STRING -- class names are strings

type_check
  -- Is the specified program type-correct?

generate_java_code
  -- For the program specified, generate Java-like code.

add_class(cn: CLASS_NAME)
  -- Add a new class with name ‘cn‘.

add_attribute(
        cn: CLASS_NAME;    -- context class
        fn: FEATURE_NAME;  -- name of attribute
        ft: CLASS_NAME     -- attribute type
)
  -- Add to class ‘cn‘ a new attribute ‘fn‘ with type ‘ft‘.

add_command(
        cn: CLASS_NAME;                                  -- context class
        fn: FEATURE_NAME;                                -- name of command
        ps: ARRAYTUPLEpn: VAR_NAME; ft: CLASS_NAME  -- parameters
)
  -- Add to class ‘cn‘ a new command ‘fn‘ with a list of parameters ‘ps‘.
  -- Each parameter is a tuple with parameter name ‘pn‘ and type ‘ft‘.

add_query(
        cn: CLASS_NAME;                                   -- context class
        fn: FEATURE_NAME;                                 -- name of query
        ps: ARRAYTUPLEpn: VAR_NAME; pt: CLASS_NAME;  -- parameters
        rt: CLASS_NAME                                    -- return type
)
  -- Add to class ‘cn‘ a new query ‘fn‘ with a list of parameters ‘ps‘
  -- and return type ‘rt‘.
  -- Each parameter is a tuple with parameter name ‘pn‘ and type ‘ft‘.

add_assignment_instruction (cn: CLASS_NAME; fn: FEATURE_NAME; n: VAR_NAME)
  -- Assign to variable with name ‘n‘, in the context of routine ‘fn‘ in class ‘cn‘.
  -- Here ‘n‘ should either be ‘Result‘ (in the context of a query),
  -- or an attribute name in the current class.

-- Below are all events related to specifying expressions for Assignment RHS.

add_call_chain(chain: ARRAYVAR_NAME)
  -- Add a chain of calls to attributes (not queries or commands).

-- Events of users adding constants
```

```
bool_value (c: BOOLEAN)
int_value (c: INTEGER)

-- Events of users adding binary arithmetic operations
addition
subtraction
multiplication
quotient
modulo

-- Events of users adding binary logical operations
conjunction
disjunction

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

-- Events of users adding unary numerical or logical operations
numerical_negation
logical_negation
```

9 Specifying Expressions as the RHS of Assignments

The RHS of each variable assignment (in the context of some routine in some class) is an *Expression* (see Figure 2). We 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:

```
      MULTIPLICATION
       +---^---+
   ADDITION    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 RHS of each variable assignment as an *Expression*, using the agreed abstract user interface, by gradually specifying sub-expressions via a *pre-order* traversal of the tree[1]:

```
multiplication
addition
int_value (1)
int_value (2)
int_value (3)
```

In-between the events, since the expression being specified is not completed yet, your software must warn the users that both `type_check` (which performs type checking) is not allowed (but `generate_java_code` is allowed at any time). On the other hand, after all 5 events above have occurred, users would expect that the occurrence of event `type_check` reports that the specified expression is type-correct. All these small details are related to how your software should inform users of the current state of the tool, before and after the occurrence of each event. We will discuss how you should display the state of your tool in the next section.

[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.

10 Outputting the Abstract State

For the purpose of using your implemented tool, users need to be informed of:

- If the `generate_java_code` event/command is invoked, then output should be a Java-like program.
- If the `type_check` event/command is invoked, then output the type checking result for each class:
  - If all assignment instructions contained in a class (in all routines) are type-correct, then output that the class is type-correct.
  - Otherwise, output all assignment instructions that are not type-correct.
- Otherwise:
  - If the RHS of an assignment instruction is not currently being specified, then first output a `Status` message, either `OK` or some error, to give feedback on the last-executed event/command.

    Table 2 (p12) summarizes the list of status messages that your software, when appropriate, must report. All possible errors should be reflected as feature preconditions of your tool (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, the corresponding error message is reported back to the user, whereas the state of your tool *must* remain unchanged.

    Then, summarize the list of declared classes. Each class is summarized by the numbers of attributes, queries, and commmands declared so far. Each attribute, query, or command is also presented with their headers (e.g,. attribute `i: INTEGER`, command `set_i(INTEGER, INTEGER)`). This output is not meant to be Java-like.
  - If the RHS of an assignment instruction is currently being specified, then also indicate the routine (and its context class) being implemented, as well as pretty-print the RHS expression being specified so far. For example:

    ```
    Routine currently being implemented: {A}.set_i
    Assignment being specified: i := ((? + nil) * nil)
    ```

    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.

    As soon the RHS expression of the current variable assignment becomes complete (i.e., no `?`), we stop showing the above two lines (`Routine currently being implemented` and `Assignment being specified`).

Message	Context
OK.	When there is no error.
Error (An assignment instruction is not currently being specified).	The RHS of a variable assignment is not currently being specified. Therefore, no expressions or sub-expressions can be added. **Applicable events**: All events related to the specification of *Expression* (e.g., `add_call_chain`, `addition`) **Reference**: `at01.txt`
Error (An assignment instruction is currently being specified for routine r in class c).	The RHS of a variable assignment is currently being specified and not yet completed. Therefore, no new class/attribute/routine/assignment can be added and type checking cannot be performed. **Applicable events**: `add_class`, `add_attribute`, `add_command`, `add_query`, `add_assignment_instruction`, `type_check`. **Reference**: `at02.txt`
Error (cn is already an existing class name).	The classes to be added already exists. **Applicable events**: `add_class` **Reference**: `at03.txt`
Error (c is not an existing class name).	The context class c specified is non-existing. **Applicable events**: `add_attribute`, `add_command`, `add_query`, `add_assignment_instruction`. **Reference**: `at03.txt`
Error (fn is already an existing feature name in class cn).	The feature to be added already exists in class cn. **Applicable events**: `add_attribute`, `add_command`, `add_query` **Reference**: `at03.txt`
Error (Parameter names clash with existing classes: $list$).	Parameter names in $list$ clash with existing class names. **Applicable events**: `add_command`, `add_query` **Reference**: `at04.txt`
Error (Duplicated parameter names: $list$).	Parameter names in $list$ are declared more than once. **Applicable events**: `add_command`, `add_query` **Reference**: `at05.txt`
Error (Parameter types do not refer to primitive types or existing classes: $list$).	Each parameter type in $list$ is neither a primitive type (i.e., `INTEGER` or `BOOLEAN`) nor a reference type (i.e., reference to some declared class). **Applicable events**: `add_command`, `add_query` **Reference**: `at06.txt`
Error (Return type does not refer to a primitive type or an existing class: rt).	The return type of an attribute or a query is neither a primitive type (i.e., `INTEGER` or `BOOLEAN`) nor a reference type (i.e., reference to some declared class). **Applicable events**: `add_attribute`, `add_query` **Reference**: `at06.txt`
Error (fn is not an existing feature name in class cn).	The context feature fn specified does not exist in class cn. **Applicable events**: `add_assignment_instruction` **Reference**: `at07.txt`
Error (Attribute fn in class cn cannot be specified with an implementation).	The feature to be added with an implementation is actually an attribute in class cn. **Applicable events**: `add_assignment_instruction` **Reference**: `at07.txt`
Error (Call chain is empty).	When the call chain of attributes is empty **Applicable Events**: `add_call_chain` **Reference**: `at07.txt`

Table 2: Messages: String Values of System Status

The above information constitutes the *abstract*[2] state of your tool. **For output, each indentation level consists of two white spaces.** As an example, consider the following use case where the user attempts to specify, type-check, and generate Java-like code for the following program:

```
class A {
  int i;
  void set_i(int x, int y) {
    i = ((1 + 2) * 3);
    i = (i + (x - y));
  }
}
```

The expected process of specifying, type-checking, and generating Java-like code for the above program is as follows (as a result of running `at08.txt` using the batch mode of ETF; you can also see this expected output output in `at08.expected.txt`):

```
  Status: OK.
  Number of classes being specified: 0
->add_class("A")
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 0
      Number of queries: 0
      Number of commands: 0
->add_attribute("A","i","INTEGER")
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 0
->add_command("A","set_i",<<["x", "INTEGER"], ["y", "INTEGER"]>>)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
->add_assignment_instruction("A","set_i","i")
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
```

[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.

```
  Assignment being specified: i := ?
->multiplication
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := (? * nil)
->addition
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := ((? + nil) * nil)
->int_value(1)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := ((1 + ?) * nil)
->int_value(2)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := ((1 + 2) * ?)
->int_value(3)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
```

```
->add_assignment_instruction("A","set_i","i")
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := ?
->addition
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := (? + nil)
->add_call_chain(<<"i">>)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := (i + ?)
->subtraction
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := (i + (? - nil))
->add_call_chain(<<"x">>)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
  Routine currently being implemented: {A}.set_i
  Assignment being specified: i := (i + (x - ?))
->add_call_chain(<<"y">>)
  Status: OK.
  Number of classes being specified: 1
    A
      Number of attributes: 1
        + i: INTEGER
      Number of queries: 0
      Number of commands: 1
        + set_i(INTEGER, INTEGER)
->type_check
  class A is type-correct.
->generate_java_code
  class A {
    int i;
    void set_i(int x, int y) {
      i = ((1 + 2) * 3);
      i = (i + (x - y));
    }
  }
```

In the above expected use case of your tool, the occurrence of each event is preceded by "dash-greater-than" (->). The initial state of the tool (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.

11 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 tool application. The input files (e.g., at01.txt, at02.txt, *etc.*) are **example** use cases for you to test your software. The corresponding expected output files (e.g., at01.expected.txt, at02.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 tool 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 tool. **When you submit, make sure that you submit the top-level analyzer directory.**

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

```
estudio19.05 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_gui_show_history -- GUI mode
   Result := etf_cl_show_history
-- Result := unit_test            -- Unit Testing mode
  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 at01.txt
```

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

```
EIFGENs/analyzer/W_code/analyzer -b at01.txt > at01.actual.txt
```

   The `at01.actual.txt` file stores the *actual* output from your current software, and your goal is to make sure that `at01.actual.txt` is identical to `at01.expected.txt` by typing:

```
diff at01.expected.txt at01.actual.txt
```

   or typing:

```
meld at01.expected.txt at01.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 expected output files (i.e., `at01.expected.txt`, `at02.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 10 days 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 certainly want to finish much of 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.**

12 Modification of the Cluster Structure

You must not change signatures of any of the classes or features that are generated by the ETF tool (the only exception is the `ETF_MODEL` and `ETF_MODEL_CLASS` in the *model* cluster, for which you may make any modifications). 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.

13 Project Report

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 programming language structure. You must show the critical class(es) in the expanded view (with contracts).
  - Page 3 details the architecture of your design that models the language operations. You must show the critical class(es) in the expanded view (with contracts).
- With no page limit, explain in details how your design (not the design of the generated ETF project) for the programming language 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
  - Any other design principle that we discussed in lectures
- 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.**

14 Submission

14.1 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 (if you choose to work as a group of two). 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_gui_show_history  -- GUI mode
   Result := etf_cl_show_history
-- Result := unit_test  -- Unit Testing mode
  end
```

14.2 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. Produced outputs by your program must be **identical** to those produced by the oracle. You are responsible for testing enough input cases with the oracle give to you.
  3. Each team must make their submission from **only** a single CSE account.
  4. 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.**
  5. Go to the directory containing the top-level `analyzer` project directory:

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

     ```
     eclean analyzer
     ```

     5.2 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.

15 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.

16 Amendments

- We assume that users do not enter empty strings ("") as names of classes/attributes/commands/-queries.
- We assume that users do not declare routine parameter names that clash with names of existing features (attributes, commands, and queries). Symmetrically, users do not declare feature names that clash with names of any parameters.
- The order in which we print features (attributes, commands, and queries), when `generate_java_code` event/command is invoked, corresponds to the order in which these features were added. For example, the following events

  ```
  add_class("A")
  add_attribute("A", "att", "INTEGER")
  add_command("A", "cmd", <<>>)
  add_assignment_instruction("A", "cmd", "att")
  int_value(2)
  generate_java_code
  type_check
  ```

  would generate a fragment of Java code:

  ```
  class A {
    int att;
    void cmd() {
      att = 2;
    }
  }
  ```

  If we swap the order of additions of the attribute `att` and the command `cmd`:

  ```
  add_class("A")
  add_command("A", "cmd", <<>>)
  add_assignment_instruction("A", "cmd", "att")
  int_value(2)
  add_attribute("A", "att", "INTEGER")
  generate_java_code
  type_check
  ```

  Then the generated java code is:

  ```
  class A {
    void cmd() {
  ```

```
    att = 2;
  }
  int att;
}
```

However, both versions of class `A` are type-correct: like in Java and Eiffel, in a routine (query or command), you can refer to features that are declared before or after it.