An Overview of Eiffel

Richard Paige

July 13, 1998

1 Introduction

This document contains a brief overview and introduction to the syntax and semantics of the Eiffel programming language. It is not a complete description of the language: it does not discuss the ISE Eiffel libraries, multiple inheritance, or genericity in full detail. It should not take the place of a legitimate language reference, e.g., [2, 3, 4]. It does, however, introduce the basic syntax of the language which you will need to use in your assignments and project.

2 Classes and Objects

In Eiffel, the fundamental language construct is the **class**. All Eiffel programs are made up of classes. A class is used in the declaration and creation of **objects**. Eiffel programs manipulate objects to perform some kind of computation. You can think of a class as consisting of all objects with the same behaviour and properties.

In an Eiffel program, every object has a well defined **type** and belongs to a class. Objects are referred to by **names**, which are just strings of characters. A name can refer to different objects at different points during execution.

A name in an Eiffel program is declared as having a type. The declaration

```
x : T
```

introduces a name `x` of type `T`. However, after declaration, `x` does not refer to any objects: we say that `x` is not bound to an object. Name `x` can be bound to any object that *type conforms* to type `T` (more on this in a moment). There are only three ways to achieve this binding.

1. **Assignment instruction:** the assignment `x:=y` binds `x` to the object to which `y` is currently bound (providing there is type conformance).
2. **A creation instruction:** the creation instruction `!!x` creates a new object of type `T`, and binds `x` to it. This is similar to `new` in C++.
3. **Routine call:** we will discuss this later.

When `x` is declared, but before any object has been explicitly bound to it, `x` is bound to nothing, expressed as the object `void`. We can see if an object is bound to `void` by writing `x=void`.

2.1 Basic Types

Eiffel allows programmers to define new types as classes. It also has a number of **basic types** (sometimes called **embedded types**) built-in. The basic types include

```
BOOLEAN, CHARACTER, INTEGER, REAL
```

Objects of these basic types are predefined, and so we can refer, in our programs, to standard objects like `true, false`, or `-37`. Names of basic types are initially bound to default values (`false` in the case of `BOOLEAN`, `0` for `INTEGER`). The basic types come predefined with the following operators:

```
BOOLEAN    not, or, and, implies, or else, and then
INTEGER    +, -, *, //, \\, ^, <, >, <=, >=
REAL       +, -, *, /, ^, <, >, <=, >=
```

The `or else` and `and then` operators are short-circuiting. Note that `//` is integer division and `\\` is modulus. Standard expression notation can be used in Eiffel.

3 Eiffel Instructions

Computations in Eiffel are specified using *expressions* (which calculate a value of some type) and *instructions*. The basic instructions in Eiffel are summarized in this section.

3.1 Creation and assignment

We have already seen the two most important instructions. The creation instruction generates a new object of some type, and binds a name to that object.

```
!!x
```

The type of the object bound to `x` will type conform to the type of `x`. Assignment is used to alter the binding of a name. `x:=y` binds name `x` to the object referenced by `y`.

3.2 Sequencing

A concatenation of instructions is merely a sequence.

```
instruction1
instruction2
...
instructionk
```

The instructions are executed one after the other. Semicolons can separate the instructions if desired.

3.3 Conditional

The conditional instruction carries out an instruction if some boolean condition is *true*. Conditionals have the following form.

```
if b1 then
    c1
elseif b2 then
    c2
 ...
elseif bk then
    ck
else
    ce
end
```

Each of the `b`s is a `BOOLEAN` expression and the `c`s are compound instructions. The `else` branch can be omitted, as can the `elseif`s.

3.4 Iteration

The iteration instruction carries out some instruction repeatedly, until a boolean condition is fulfilled. Iterations in Eiffel have the following form (there is only one form of iteration in Eiffel, unlike other languages).

```
from
   c1       -- loop initialization, executed once
until
   b        -- exit condition
loop
   c2       -- body of the loop
end
```

First, `c1` is executed. Then, expression `b` is evaluated. If it is `false`, then `c2` is executed and `b` is re-evaluated. If `b` is `true`, then the iteration is complete.

Comments in Eiffel are introduced by the double dash, `--`, and continue until the end of the line.

3.5 Procedures and functions

A procedure is a compound instruction associated with a name and possibly a list of parameters. Procedures have the following form in Eiffel.

```
pname( a1:T1; a2:T2; ...; an:Tn) is
  local
       -- declaration of local variables
  do
    c  -- body of pname
  end
```

`pname` is the name given to the procedure being defined. The `a`s are the names of the arguments to the procedure. The arguments have the types given by the `T`s. The `local` clause is optional: it list the names and types of any local variables. The body of the procedure, `c`, is the instructions that are to be executed every time `pname` is called. Assignment to the arguments within `c` is not permitted.

To call the procedure `pname`, we write

```
pname(e1,e2,...,en)
```

where the `e`s are expressions of appropriate types; they are called the actual arguments.

Functions are analogous to procedures, except that they return a value. The text of a function definition must declare the type of this returned value. The general form of a function definition is as follows.

```
fname( a1:T1; a2:T2; ...; an:Tn ) : T  is
  local
       -- declaration of local variables
  do
    c  -- body of fname
  end
```

We have declared that the function `fname` will return an object of type `T` when it is complete. There must be a mechanism by which the body `c` specifies the value to be returned. This is achieved by an assignment of the form

```
result := expression
```

in the body `c`. The predeclared name `result` may only be used in the body of a function (or in the function's **ensure** clause, which we'll see later), and the value it receives in the last such assignment carried out in `c` is the value returned.

Functions and procedures may be recursive in Eiffel.

4 Classes

In Eiffel, one writes classes rather than programs. The text of an Eiffel class specifies the features (attributes and methods) of each object belonging to the class. Text for a class will have the following form.

```
class CNAME
creation
   -- the names of the creation procedures
feature
   -- the declarations or definitions of
   -- all class features
end
```

The `creation` clause is optional. If present, it names one or more commands appearing in the `feature` clause. One of these creation procedures must be called every time an object of the given class is created. The job of this command is to initialize the object. To call the creation procedure, simply tag the name (and arguments) of the procedure after the creation instruction, e.g.,

```
!!x.make(e1,e2)
```

The compiler will complain if a creation procedure is specified, but is not used during creation.

Though Eiffel has creation procedures, it has no 'destruction' procedures. Destruction is handled automatically by a garbage collector.

Features of a class may be of the following forms.

- **Attributes:** i.e., lists of names of entities that belong to any instance of the class
- **Queries:** i.e., functions that belong to the class

- **Commands:** i.e., procedures that belong to the class

Here is an example.

```
class LIST
creation make
feature
   count : INTEGER
   first : NODE

   empty : BOOLEAN is
     do
       result := (count=0)
     end

   add( x : ELEMENT ) is
     local
        n : NODE
     do
        if not has(x) then
          !!n
          n.set_item(x)
          n.set_next(first)
          first := n
          count := count+1
        end
     end

   make is
     do
        count := 0
        first := void
     end
end LIST
```

Features `count` and `first` are attributes: objects of types `INTEGER` and `NODE`, respectively. `empty` is a `BOOLEAN` query, while `add` and `make` are commands. `make` is also a creation procedure.

4.1 Accessing features

As hinted at already, features of a class are accessed using dot notation. Suppose we declare

```
x : LIST
```

Then to access feature `add` applied to object `x`, we write

```
x.add(y)
```

(where `y` conforms to type `ELEMENT`). We will see shortly how to prevent clients from accessing features of a class.

> **ASIDE.** If a name `x` is not bound to any object (i.e., `x=void`) then it is a run-time error to evaluate any feature of `x` by the dot notation.

It is not possible for one object to alter an attribute of another object via an assignment. So writing, e.g., `x.first := n` is an error. The only entity able to alter the attributes of an object is the object itself, via its features.

4.2 Equality

The notion of equality in Eiffel is somewhat different from what you may have seen in other languages. In every class, there is a predefined (i.e., you do not have to write it) boolean function `equal(x,y)` that returns `true` when

- both arguments are of the same type, and
- both arguments are equal attribute for attribute

Thus, `equal(x,y)` does a feature-by-feature comparison of objects. Note that this is *not* the same as `x=y`, which is `true` if and only if `x` and `y` refer to the same object. Figure 4.2 illustrates the situation.

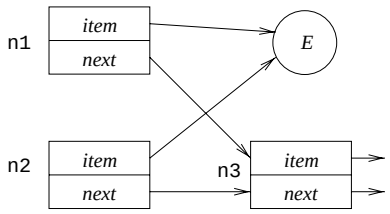

Fig. 1: Equality

In Figure 4.2, we have names `n1`, `n2`, and `n3` referring to the objects as shown. Clearly, `n1 /= n2` (the two names refer to two distinct objects!). But `equal(n1,n2)` is `true`, because the features of both objects agree.

Note that `=` and `equal` agree on the basic types in Eiffel.

4.3 Visibility

A class may have features that should be accessible to clients (i.e., features that should be in the interface of the class). A class may also have features that should be inaccessible and invisible to clients; these features are implementation-oriented ones. Eiffel offers the ability to **export** some features, and hide others. This is done by the **feature** clause in a class. There may be an arbitrary number of **feature** clauses, optionally qualified with a list of classes to which the features are to be exported.

```
feature { C1, C2, ..., Ck }
```

All features between this `feature` clause and the next (or the end of the class) are exported only to classes `C1, C2, .., Ck`. Thus, these and only these classes may use the exported features.

If the `feature` clause is not qualified by a list of classes, then the following features are exported to all classes. This is equivalent to writing

```
feature { ANY }
```

To make a feature invisible to all classes, we write

```
feature { NONE }
```

This is a very strong restriction: the following features are also invisible to other objects of the same class!

5 Assertions and Contracts

Eiffel, unlike most languages, has built-in and powerful support for writing **assertions**. An assertion is a boolean expression on the state of the program that is evaluated when it is reached during execution. If an assertion evaluates to *true*, execution continues; otherwise, execution may halt, or an exception may be raised (see the references for details on exception handling).

There are five kinds of assertions in Eiffel.

1. **Preconditions:** assertions that must be true when a routine is called.
2. **Postconditions:** assertions that must be true when a routine returns.
3. **General assertions:** (via the `check` clause) assertions that must be true when execution reaches them.
4. **Class invariants:** assertions that must be maintained true by all instances of a class.
5. **Loop invariants:** (which we will not discuss here)

Preconditions and postconditions are associated with routines of a class. The syntax of a procedure has the following format.

```
pname( arglist ) is
   require
       -- the preconditions
   local
       -- local variables
   do
       -- body of pname
   ensure
       -- the postconditions
   end
```

The preconditions must be true when the routine is called, and the postconditions must have been established (by the body of the routine) when the routine terminates. The `require` and `ensure` clauses may appear in functions, as well. Here is an example.

```
gcd( m,n : INTEGER ) : INTEGER is
   require  m>=0 and then n>=0
   do
     if n=0 then
       result := m
     else
       result := gcd(n, m\\n)
     end
   end
```

Assertions are limited to using the syntax of Eiffel's boolean expressions; no logical quantifiers are permitted. Calls to queries of an object are permitted in assertions.

5.1 Class invariants

The boolean expressions in a class invariant define conditions that must be true of all objects of the class at 'stable' times (essentially, upon exit from a method of a class). During execution of a method, the invariant may be violated. Every method (excepting the creation procedures) can assume, when it starts to execute, that its preconditions are met *and* the class invariant is satisfied. After execution, the postcondition must be met *and* the class invariant must be satisfied.

The text of a class is expanded by an **invariant** clause.

```
class CLASS_NAME
creation
   -- list of creation procedures
feature
   -- list of attributes and methods
invariant
   -- boolean expressions of class invariant
end
```

The creation procedures cannot assume that the invariant is true when it is called, though the creation procedure must establish the invariant.

Here is an example of a class invariant for a `BANK_ACCOUNT` class.

```
class BANK_ACCOUNT feature
    balance : INTEGER
    -- features here

    invariant
       balance >= 0
end -- BANK_ACCOUNT
```

6 Inheritance

Inheritance has been discussed in lecture and in your textbook. Eiffel supports both single and multiple inheritance. We discuss single inheritance here, and leave multiple inheritance to the references. Each class may contain an inheritance clause, as follows.

```
class CNAME
inherit
   -- list of classes from which CNAME inherits
creation
   -- creation procedure names
feature
   -- features and attributes
invariant
   -- class invariant
end
```

The `inherit` clause may contain arbitrarily many names of other classes.

Class `CNAME` acquires all features from the classes from which it inherits (attributes and methods). All invariants of the parent classes are taken over by the child class (they are `and`ed to the new invariant of

the child class). The export policy of features in parent classes are inherited by child classes. This can be overridden using the **export** clause.

6.1 Renaming and redefining

The full form of the **inherit** clause is somewhat more complicated than we have shown here. The child class may *rename* features from a parent class, and may *redefine* features from a parent class. Renaming is useful in avoiding name clashes (which can arise in multiple inheritance), and if you want a new name for a feature in a child class. Redefining is useful if a feature in a parent class doesn't quite do what is required for a child class.

Here is an example of redefinition. Suppose we have a class `EMPLOYEE` with feature `months_pay`. We want to create a class `SECRETARY`, inheriting from `EMPLOYEE`, but with a different definition of the feature `months_pay`.

```
class SECRETARY
inherit
  EMPLOYEE
    redefine months_pay end

creation make
feature
   hours_worked : REAL
   hourly_wage : REAL

   months_pay : REAL is
     do
       result := hourly_wage * hours_worked
     end

-- more here
end SECRETARY
```

A salesperson might be paid according to a different scheme.

```
class SALESPERSON
inherit
  EMPLOYEE
    redefine months_pay end

creation make
feature
   salary, bonus : REAL

   months_pay : REAL is
     do
       result := salary + bonus
     end
-- more here
end
```

The method `months_pay` in class `EMPLOYEE` is redefined in the child classes.

Renaming of features is done in a similar fashion. An example of a rename clause (omitting the enclosing class details) looks like the following.

```
inherit
   GRAPH
     rename
        add_vertex as add_class,
        add_edge   as add_relation,
        ...
```

The name `add_vertex` in class `GRAPH` is renamed to `add_class` in the child class. The parent class is unchanged.

7 Type Conformance

Recall that the assignment `x:=y` is only permitted when the type of `y` conforms to the type of `x`. We can now define what type conformance means. First, we define two terms:

- a class *A* is an **ancestor** of class *B* if *A* and *B* are the same class, or *A* is an ancestor of a parent of *B*.
- a class *B* is a **descendent** of class *A* if *A* is an ancestor of *B*.

Class *B* **conforms** to class *A* if and only if *B* is a descendent of *A*.

8 Generic Classes

A generic class is one that is parameterized by a type. An example is the standard Eiffel class `ARRAY`. It is parameterized by the abstract type `G`. The array can hold elements of type `G`. We write this as

```
ARRAY[G]
```

We can use the generic class to declare arrays of different types. To declare an array of integers, we write

```
x : ARRAY[INTEGER]
```

i.e., we fill in the generic type `G` with the embedded type `INTEGER`. To declare an array of arrays of reals, we write

```
y : ARRAY[ARRAY[REAL]]
```

Other standard generic classes include: `SET, STACK, QUEUE, LIST, SEQUENCE`, and so on.

You can write your own generic classes. For example, you can write a generic `NODE` class as follows.

```
class NODE[G]
feature
   item : G
   next : NODE[G]

   set_item( x:G ) is
     do item := x end
```

```
    set_next( n:NODE[G] ) is
      do next := n end
end
```

The class is parameterized by the abstract type `G`; nodes hold elements of type `G`. Instead of using a concrete type (like `INTEGER`) throughout the implementation of `NODE`, you use `G`. The class can be instantiated like `ARRAY`, above.

```
n : NODE[INTEGER]
```

References

[1] B. Meyer. *Object-oriented Software Construction*, Second Edition, Prentice-Hall, 1997.

[2] R. Rist and R. Terwilliger. *Object-Oriented Programming in Eiffel*, Prentice-Hall, 1995.

[3] R. Switzer. *Eiffel: An Introduction*, Prentice-Hall, 1993.

[4] P. Thomas and R. Weedon. *Object-Oriented Programming in Eiffel*, Second Edition, Addison-Wesley, 1998.