Syntax of Eiffel: a Brief Overview

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Escape Sequences

Escape sequences are special characters to be placed in your program text.

- In Java, an escape sequence starts with a backward slash `\` e.g., `\n` for a new line character.
- In Eiffel, an escape sequence starts with a percentage sign `%` e.g., `%N` for a new line characgter.

See here for more escape sequences in Eiffel: `https://www.eiffel.org/doc/eiffel/Eiffel%20programming%20language%20syntax#Special_characters`

Commands, and Queries, and Features

- In a Java class:
  - **Attributes**: Data
  - **Mutators**: Methods that change attributes without returning
  - **Accessors**: Methods that access attribute values and returning
- In an Eiffel class:
  - Everything can be called a *feature*.
  - But if you want to be specific:
    - Use *attributes* for data
    - Use *commands* for mutators
    - Use *queries* for accessors

Naming Conventions

- Cluster names: all lower-cases separated by underscores

  e.g., `root`, `model`, `tests`, `cluster_number_one`
- Classes/Type names: all upper-cases separated by underscores

  e.g., `ACCOUNT`, `BANK_ACCOUNT_APPLICATION`
- Feature names (attributes, commands, and queries): all lower-cases separated by underscores

  e.g., `account_balance`, `deposit_into`, `withdraw_from`

Operators: Assignment vs. Equality

- In Java:
  - Equal sign = is for assigning a value expression to some variable. e.g., `x = 5 * y` changes `x`'s value to `5 * y`
    This is actually controversial, since when we first learned about =, it means the mathematical equality between numbers.
  - Equal-equal == and bang-equal `!=` are used to denote the equality and inequality.
    e.g., `x == 5 * y` evaluates to *true* if `x`'s value is equal to the value of `5 * y`, or otherwise it evaluates to *false*.
- In Eiffel:
  - Equal = and slash equal `/=` denote equality and inequality.
    e.g., `x = 5 * y` evaluates to *true* if `x`'s value is equal to the value of `5 * y`, or otherwise it evaluates to *false*.
  - We use `:=` to denote variable assignment.
    e.g., `x := 5 * y` changes `x`'s value to `5 * y`
  - Also, you are not allowed to write shorthands like `x++`, just write `x := x + 1`.

Attribute Declarations

- In Java, you write: `int i`, `Account acc`
- In Eiffel, you write: `i: INTEGER`, `acc: ACCOUNT`

  Think of `:` as the set membership operator $\in$:

  e.g., The declaration `acc: ACCOUNT` means object `acc` is a member of all possible instances of `ACCOUNT`.

Method Declaration

- **Command**

```
deposit (amount: INTEGER)
  do
    balance := balance + amount
  end
```

Notice that you don't use the return type `void`

- **Query**

```
sum_of (x: INTEGER; y: INTEGER): INTEGER
  do
    Result := x + y
  end
```

  - Input parameters are separated by semicolons `;`
  - Notice that you don't use `return`; instead assign the return value to the [redacted]

Operators: Logical Operators (1)

- Logical operators (what you learned from EECS1090) are for combining Boolean expressions.
- In Eiffel, we have operators that *EXACTLY* correspond to these logical operators:

	LOGIC	EIFFEL
Conjunction	$\wedge$	**and**
Disjunction	$\vee$	**or**
Implication	$\Rightarrow$	**implies**
Equivalence	$\equiv$	**=**

Review of Propositional Logic (1)

- A *proposition* is a statement of claim that must be of either *true* or *false*, but not both.
- Basic logical operands are of type Boolean: *true* and *false*.
- We use logical operators to construct compound statements.
  - Binary logical operators: conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equivalence (a.k.a if-and-only-if $\iff$)

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \iff q$
true	*true*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*
false	*true*	*false*	*true*	*true*	*false*
false	*false*	*false*	*false*	*true*	*true*

  - Unary logical operator: negation ($\neg$)

p	$\neg p$
true	*false*
false	*true*

Review of Propositional Logic: Implication

- Written as $p \Rightarrow q$
- Pronounced as "p implies q"
- We call p the antecedent, assumption, or premise.
- We call q the consequence or conclusion.
- Compare the *truth* of $p \Rightarrow q$ to whether a contract is *honoured*: $p \approx$ promised terms; and $q \approx$ obligations.
- When the promised terms are met, then:
  - The contract is *honoured* if the obligations are fulfilled.
  - The contract is *breached* if the obligations are not fulfilled.
- When the promised terms are not met, then:
  - Fulfilling the obligation (q) or not ($\neg q$) does *not breach* the contract.

p	q	$p \Rightarrow q$
true	*true*	*true*
true	*false*	*false*
false	*true*	*true*
false	*false*	*true*

Review of Propositional Logic (2)

- **Axiom**: Definition of $\Rightarrow$

$$p \Rightarrow q \equiv \neg p \vee q$$

- **Theorem**: Identity of $\Rightarrow$

$$true \Rightarrow p \equiv p$$

- **Theorem**: Zero of $\Rightarrow$

$$false \Rightarrow p \equiv true$$

- **Axiom**: De Morgan

$$\begin{array}{rcl} \neg(p \wedge q) & \equiv & \neg p \vee \neg q \\ \neg(p \vee q) & \equiv & \neg p \wedge \neg q \end{array}$$

- **Axiom**: Double Negation

$$p \equiv \neg\, (\neg\, p)$$

- **Theorem**: Contrapositive

$$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$$

Review of Predicate Logic (1)

- A *predicate* is a *universal* or *existential* statement about objects in some universe of disclosure.
- Unlike propositions, predicates are typically specified using *variables*, each of which declared with some *range* of values.
- We use the following symbols for common numerical ranges:
  - $\mathbb{Z}$: the set of integers
  - $\mathbb{N}$: the set of natural numbers
- Variable(s) in a predicate may be *quantified*:
  - *Universal quantification* :
    All values that a variable may take satisfy certain property.
    e.g., Given that i is a natural number, i is *always* non-negative.
  - *Existential quantification* :
    Some value that a variable may take satisfies certain property.
    e.g., Given that i is an integer, i *can be* negative.

Review of Predicate Logic (2.1)

- A **universal quantification** has the form $(\forall X \mid R \bullet P)$
  - X is a list of variable *declarations*
  - R is a *constraint on ranges* of declared variables
  - P is a *property*
  - $(\forall X \mid R \bullet P) \equiv (\forall X \bullet R \Rightarrow P)$
    e.g., $(\forall X \mid True \bullet P) \equiv (\forall X \bullet True \Rightarrow P) \equiv (\forall X \bullet P)$
    e.g., $(\forall X \mid False \bullet P) \equiv (\forall X \bullet False \Rightarrow P) \equiv (\forall X \bullet True) \equiv True$
- *For all* (combinations of) values of variables declared in X that satisfies R, it is the case that P is satisfied.
  - $\forall i \mid i \in \mathbb{N} \bullet i \geq 0$ [*true*]
  - $\forall i \mid i \in \mathbb{Z} \bullet i \geq 0$ [*false*]
  - $\forall i, j \mid i \in \mathbb{Z} \wedge j \in \mathbb{Z} \bullet i < j \vee i > j$ [*false*]
- The range constraint of a variable may be moved to where the variable is declared.
  - $\forall i : \mathbb{N} \bullet i \geq 0$
  - $\forall i : \mathbb{Z} \bullet i \geq 0$
  - $\forall i, j : \mathbb{Z} \bullet i < j \vee i > j$

Review of Predicate Logic (2.2)

- An ***existential quantification*** has the form $(\exists X \mid R \bullet P)$
  - X is a list of variable *declarations*
  - R is a *constraint on ranges* of declared variables
  - P is a *property*
  - $(\exists X \mid R \bullet P) \equiv (\exists X \bullet R \wedge P)$
    e.g., $(\exists X \mid True \bullet P) \equiv (\exists X \bullet True \wedge P) \equiv (\forall X \bullet P)$
    e.g., $(\exists X \mid False \bullet P) \equiv (\exists X \bullet False \wedge P) \equiv (\exists X \bullet False) \equiv False$
- *There exists* a combination of values of variables declared in X that satisfies R and P.
  - $\exists i \mid i \in \mathbb{N} \bullet i \geq 0$ [*true*]
  - $\exists i \mid i \in \mathbb{Z} \bullet i \geq 0$ [*true*]
  - $\exists i, j \mid i \in \mathbb{Z} \wedge j \in \mathbb{Z} \bullet i < j \vee i > j$ [*true*]
- The range constraint of a variable may be moved to where the variable is declared.
  - $\exists i : \mathbb{N} \bullet i \geq 0$
  - $\exists i : \mathbb{Z} \bullet i \geq 0$
  - $\exists i, j : \mathbb{Z} \bullet i < j \vee i > j$

Predicate Logic (3)

- Conversion between ∀ and ∃

$$(\forall X \mid R \bullet P) \iff \neg(\exists X \bullet R \Rightarrow \neg P)$$
$$(\exists X \mid R \bullet P) \iff \neg(\forall X \bullet R \Rightarrow \neg P)$$

- Range Elimination

$$(\forall X \mid R \bullet P) \iff (\forall X \bullet R \Rightarrow P)$$
$$(\exists X \mid R \bullet P) \iff (\exists X \bullet R \wedge P)$$

Operators: Logical Operators (2)

- How about Java?
  - Java does not have an operator for logical implication.
  - The `==` operator can be used for logical equivalence.
  - The `&&` and `||` operators only **approximate** conjunction and disjunction, due to the ***short-circuit effect*** *(SCE)*:
    - When evaluating `e1 && e2`, if `e1` already evaluates to *false*, then `e1` will **not** be evaluated.
      e.g., In `(y != 0) && (x / y > 10)`, the SCE guards the division against division-by-zero error.
    - When evaluating `e1 || e2`, if `e1` already evaluates to *true*, then `e1` will **not** be evaluated.
      e.g., In `(y == 0) || (x / y > 10)`, the SCE guards the division against division-by-zero error.
  - However, in math, we always evaluate both sides.
- In Eiffel, we also have the version of operators with SCE:

	short-circuit conjunction	short-circuit disjunction
Java	`&&`	`||`
Eiffel	**and then**	**or else**

Operators: Division and Modulo

	Division	Modulo (Remainder)
Java	`20 / 3` is 6	`20 % 3` is 2
Eiffel	`20 // 3` is 6	`20 \\ 3` is 2

Class Declarations

- In Java:

```
class BankAccount {
  /* attributes and methods */
}
```

- In Eiffel:

```
class BANK_ACCOUNT
  /* attributes, commands, and queries */
end
```

Class Constructor Declarations (1)

- In Eiffel, constructors are just commands that have been *explicitly* declared as **creation features**:

```
class BANK_ACCOUNT
-- List names commands that can be used as constru
create
  make
feature -- Commands
  make (b: INTEGER)
    do balance := b end
  make2
    do balance := 10 end
end
```

- Only the command `make` can be used as a constructor.
- Command `make2` is not declared explicitly, so it cannot be used as a constructor.

Creations of Objects (1)

- In Java, we use a constructor `Accont(int b)` by:
  - Writing `Account acc = new Account(10)` to create a named object `acc`
  - Writing `new Account(10)` to create an anonymous object
- In Eiffel, we use a creation feature (i.e., a command explicitly declared under `create`) `make (int b)` in class `ACCOUNT` by:
  - Writing `create {ACCOUNT} acc.make (10)` to create a named object `acc`
  - Writing `create {ACCOUNT}.make (10)` to create an anonymous object
- Writing

```
create {ACCOUNT} acc.make (10)
```

is really equivalent to writing

```
acc := create {ACCOUNT}.make (10)
```

Selections

```
if B₁ then
  -- B₁
  -- do something
elseif B₂ then
  -- B₂ ∧ (¬B₁)
  -- do something else
else
  -- (¬B₁) ∧ (¬B₂)
  -- default action
end
```

Loops (1)

- In Java, the Boolean conditions in `for` and `while` loops are **stay** conditions.

```
void printStuffs() {
  int i = 0;
  while( i < 10 ) {
    System.out.println(i);
    i = i + 1;
  }
}
```

- In the above Java loop, we *stay* in the loop as long as `i < 10` is true.
- In Eiffel, we think the opposite: we *exit* the loop as soon as `i >= 10` is true.

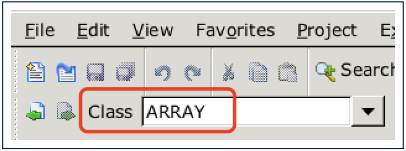
Loops (2)

In Eiffel, the Boolean conditions you need to specify for loops are **exit** conditions (logical negations of the stay conditions).

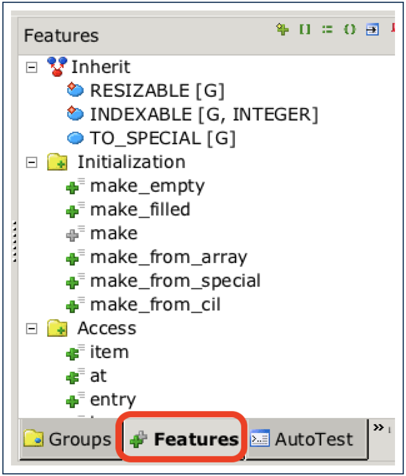
```
print_stuffs
  local
    i: INTEGER
  do
    from
      i := 0
    until
      i >= 10
    loop
      print (i)
      i := i + 1
    end -- end loop
  end -- end command
```

Library Data Structures

Enter a DS name.



Explore supported features.



Data Structures: Arrays

- Creating an empty array:

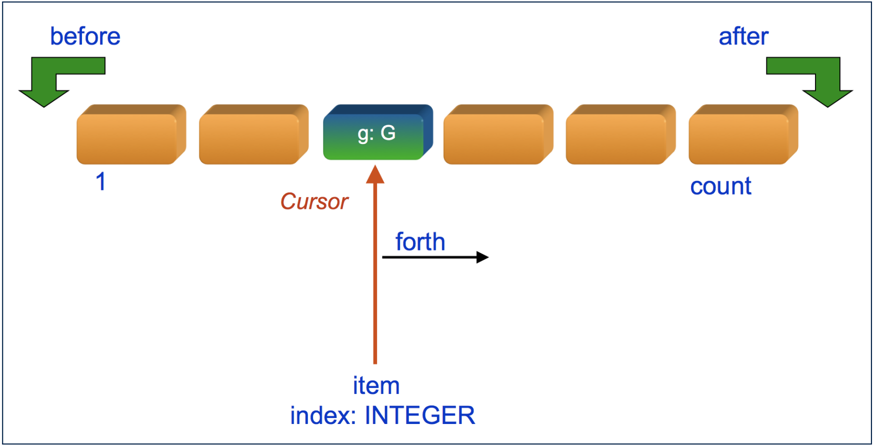
```
local a: ARRAY[INTEGER]
do create {ARRAY[INTEGER]} a.make_empty
```

  - This creates an array of `lower` and `upper` indices 1 and 0.
  - Size of array `a`: `a.upper - a.lower + 1`.

- Typical loop structure to iterate through an array:

```
local
  a: ARRAY[INTEGER]
  i, j: INTEGER
do
  ...
from
  j := a.lower
until
  j > a.upper
do
  i := a [j]
  j := j + 1
end
```

Data Structures: Linked Lists (1)



Data Structures: Linked Lists (2)

- Creating an empty linked list:

```
local
  list: LINKED_LIST[INTEGER]
do
  create {LINKED_LIST[INTEGER]} list.make
```

- Typical loop structure to iterate through a linked list:

```
local
  list: LINKED_LIST[INTEGER]
  i: INTEGER
do
  ...
from
  list.start
until
  list.after
do
  i := list.item
  list.forth
end
```

Using `across` for Quantifications

- ***across*** ... ***as*** ... ***all*** ... ***end***

  A Boolean expression acting as a universal quantification (∀)

```
1  local
2    allPositive: BOOLEAN
3    a: ARRAY[INTEGER]
4  do
5    ...
6    Result :=
7      across
8        a.lower |..| a.upper as i
9      all
10       a [i.item] > 0
11     end
```

  - **L8**: `a.lower |..| a.upper` denotes a list of integers.
  - **L8**: `as i` declares a list cursor for this list.
  - **L10**: `i.item` denotes the value pointed to by cursor `i`.

- **L9**: Changing the keyword **all** to ***some*** makes it act like an existential quantification ∃.

Index (1)

Index (2)