Introduction to Programming

Lab Session 5

(With material from the ETH Zurich course "Introduction to Programming")

September 20, 2016

INNOPOLIS UNIVERSITY

News

News

- Representatives meeting (week 8).

In this Lab

- Some logic.
- Object creation.
- Loops (variant and invariant)
- Exercises

Some logic[1]

[1] detailed description of the logics for this course will be taught in the Discrete Math course

Propositional Logic

- Constants: *True*, *False*
- Atomic formulae (propositional variables): P, Q, ...
- Logical connectives: **not**, **and**, **or**, **implies**, $=$
- Formulae: ϕ, χ, ... are of the form:
  - *True* | *False*
  - P
  - **not** ϕ | (ϕ **and** χ) | (ϕ **or** χ) | (ϕ **implies** χ) | ($\phi = \chi$)

To facilitate the reading of a formulae, parentheses can be omitted whenever there is not ambiguity.

Propositional Logic

Truth assignment and truth table

- Assigning a truth value to each propositional variable

Tautology

- *True* for all truth assignments
  - *(P* **or not** *P)* [2]
  - **not** *(P* **and not** *P)*
  - *((P* **and** *Q)* **or** *(***not** *P* **or not** *Q))*[3]

Contradiction

- *False* for all truth assignments
  - *(P* **and not** *P)*

[2] can also be rewritten as *P* **or not** *P*

[3] can also be rewritten as *(P* **and** *Q)* **or** *(***not** *P* **or not** *Q)*

Propositional Logic

Satisfiable

- *True* for at least one truth assignment.

Equivalent

- ϕ and χ are equivalent if they are satisfied under exactly the same truth assignments, or if $\phi = \chi$ is a tautology

Tautology/contradiction/satisfiable? (hands-on)

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable*

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable*

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*
- P **and not** P

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*
- P **and not** P *contradiction*

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*
- P **and not** P *contradiction*
- Q **implies** $(P$ **and not** $P)$

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*
- P **and not** P *contradiction*
- Q **implies** $(P$ **and not** $P)$ *satisfiable*

Tautology/contradiction/satisfiable? (hands-on)

- P **or** Q *satisfiable* (give an example)
- P **and** Q *satisfiable* (give an example)
- P **or not** P *tautology*
- P **and not** P *contradiction*
- Q **implies** $(P$ **and not** $P)$ *satisfiable* (give an example)

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

P	Q	P **implies** Q	**not** P **implies not** Q	**not** Q **implies not** P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

P	Q	P **implies** Q	**not** P **implies not** Q	**not** Q **implies not** P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

- P **implies** $Q = ($**not** P **implies not** $Q)$

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

P	Q	P **implies** Q	**not** P **implies not** Q	**not** Q **implies not** P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

- P **implies** $Q = ($**not** P **implies not** $Q)$ False

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

P	Q	P **implies** Q	**not** P **implies not** Q	**not** Q **implies not** P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

- P **implies** $Q = ($**not** P **implies not** $Q)$ False
- P **implies** $Q = ($**not** Q **implies not** $P)$

Equivalence (hands-on)

Do the following equivalences hold?

- $(P$ **implies** $Q) = ($**not** P **implies not** $Q)$
- $(P$ **implies** $Q) = ($**not** Q **implies not** $P)$

P	Q	P **implies** Q	**not** P **implies not** Q	**not** Q **implies not** P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

- P **implies** $Q = ($**not** P **implies not** $Q)$ False
- P **implies** $Q = ($**not** Q **implies not** $P)$ True

Useful equivalence

De Morgan laws

- **not** $(P$ **or** $Q) =$ **not** P **and** **not** Q
- **not** $(P$ **and** $Q) =$ **not** P **or** **not** Q

Implications

- P **implies** $Q =$ **not** P **or** Q
- P **implies** $Q =$ **not** Q **implies not** P

Equality on Boolean expressions

- $(P = Q) = (P$ **implies** $Q)$ **and** $(Q$ **implies** $P)$

Predicate Logic

- Domain of discourse: D
- Variables: $x : D$
- Functions: $f : D^n \to D$
- Predicates: $P : D^n \to \{True, False\}$
- Logical connectives: **not**, **and**, **or**, **implies**, $=$
- Formulae: ϕ, χ, ... are of the form:
  - $P(x, \dots)$
  - **not** ϕ | (ϕ **and** χ) | (ϕ **or** χ) | (ϕ **implies** χ) | (ϕ **=** χ)
  - $\forall x | \phi$
  - $\exists x | \phi$

To facilitate the reading of a formulae, parentheses can be omitted whenever there is not ambiguity.

Existential and universal quantification

There exists a human whose name is Bill Gates

$$\exists h : Human \mid h.name = \text{``}BillGates''$$

All persons have a name

$$\forall p : Person \mid p.name / = Void$$

Some people are students

$$\exists p : Person \mid p.is\_student$$

The age of any person is at least 0

$$\forall p : Person \mid p.age >= 0$$

Nobody likes Rivella

$$\forall p : Person \mid not\ p.likes(Rivella)$$

or

$$not\ \exists p : Person \mid p.likes(Rivella)$$

Semi-strict operations

Semi-strict operators (**and then**, **or else**)

- a **and then** b
  has same value as a **and** b if a and b are defined, and has value **False** whenever a has value **False**.

  $$text \ /= \ \textbf{Void and then} \ \ text.contains \ (\text{“Joe”})$$

- a **or else** b
  has same value as a **or** b if a and b are defined, and has value **True** whenever a has value **True**.

  $$list = \textbf{Void or else} \ \ list.is\_empty$$

Object creation

Creation procedures

- Instruction **create** x will initialise all the fields of the new object attached to x with default values
- What if we want some specific initialisation? E.g., to make object consistent with its class invariant?

Creation procedures

- Instruction **create** x will initialise all the fields of the new object attached to x with default values
- What if we want some specific initialisation? E.g., to make object consistent with its class invariant?

```
Class CUSTOMER
...
    id : STRING
invariant
    id /= Void
```

Creation procedures

- Instruction **create** *x* will initialise all the fields of the new object attached to *x* with default values
- What if we want some specific initialisation? E.g., to make object consistent with its class invariant?

```
Class CUSTOMER
...
    id : STRING
invariant
    id /= Void
```

Use creation procedure: **create** *a_customer.set_id ("13400002")*

Class *CUSTOMER*

```
class CUSTOMER

create set_id

feature

    id : STRING
            -- Unique identifier for Current.

    set_id (a_id : STRING)
            -- Associate this customer with 'a_id'.
        require
            id_exists : a_id /= Void
        do
            id := a_id
        ensure
            id_set : id = a_id
        end

invariant
    id_exists : id /= Void
end
```

Class *CUSTOMER*

```
class CUSTOMER

create set_id

feature

    id : STRING
            -- Unique identifier for Current.

    set_id (a_id : STRING)
            -- Associate this customer with 'a_id'.
        require
            id_exists : a_id /= Void
        do
            id := a_id
        ensure
            id_set : id = a_id
        end

invariant
    id_exists : id /= Void
end
```

List one or more creation procedures

May be used as a regular command and as a creation procedure.

Is established by *set_id*

Class *CUSTOMER*

```
class CUSTOMER

create set_id                List one or more creation procedures

feature

    id : STRING
            -- Unique identifier for Current.
                                         May be used as a regular command and
    set_id (a_id : STRING)                  as a creation procedure.
            -- Associate this customer with ‘a_id’.
        require
            id_exists : a_id /= Void
        do
            id := a_id
        ensure
            id_set : id = a_id
        end

invariant
    id_exists : id /= Void                  Is established by set_id
end
```

Class *CUSTOMER*

```
class CUSTOMER

create set_id                List one or more creation procedures

feature

    id : STRING
            -- Unique identifier for Current.
                                            May be used as a regular command and
    set_id (a_id : STRING)                    as a creation procedure.
            -- Associate this customer with 'a_id'.
        require
            id_exists : a_id /= Void
        do
            id := a_id
        ensure
            id_set : id = a_id
        end

invariant
    id_exists : id /= Void                  Is established by set_id
end
```

Class *CUSTOMER*

```
class CUSTOMER

create set_id                          List one or more creation procedures

feature

    id : STRING
            -- Unique identifier for Current.
                                        May be used as a regular command and
    set_id (a_id : STRING)                  as a creation procedure.
            -- Associate this customer with 'a_id'.
        require
            id_exists : a_id /= Void
        do
            id := a_id
        ensure
            id_set : id = a_id
        end

invariant
    id_exists : id /= Void             Is established by set_id
end
```

Loop invariant and variant.

Loop: basic form (from previous lab)

```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Loop: basic form (from previous lab)

```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Compound.

Loop: basic form (from previous lab)

```
from
    initialisation
until
    exit_condition
loop
    body
end
```

Compound.

Boolean Expression.

Loop: basic form (from previous lab)

from
    initialisation ← Compound.
until
    exit_condition ← Boolean Expression.
loop
    body ← Compound.
end

Loop: more general form (from previous lecture)

```
from
    initialisation
invariant
    inv
until
    exit_condition
loop
    body
variant
    var
end
```

Loop: more general form (from previous lecture)

```
from
    initialisation
invariant
    inv
until
    exit_condition
loop
    body
variant
    var
end
```

Compound.

Loop: more general form (from previous lecture)

from
    initialisation
invariant
    inv
until
    exit_condition
loop
    body
variant
    var
end

Compound.

Boolean Expression (optional).

Loop: more general form (from previous lecture)

from
    initialisation ← Compound.
invariant
    inv ← Boolean Expression (optional).
until
    exit_condition ← Boolean Expression.
loop
    body
variant
    var
end

Loop: more general form (from previous lecture)

```
from
    initialisation
invariant
    inv
until
    exit_condition
loop
    body
variant
    var
end
```

Compound.

Boolean Expression (optional).

Boolean Expression.

Compound.

Loop: more general form (from previous lecture)

from
    initialisation ← Compound.
invariant
    inv ← Boolean Expression (optional).
until
    exit_condition ← Boolean Expression.
loop
    body ← Compound.
variant
    var ← Integer Expression (optional).
end

Invariant and variant

Loop invariant (do not confuse with class invariant)

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all n elements in a list, on each iteration i ($i = 1..n$) the sum of first i elements is obtained".

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all n elements in a list, on each iteration i ($i = 1..n$) the sum of first i elements is obtained".

Loop variant

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all n elements in a list, on each iteration i ($i = 1..n$) the sum of first i elements is obtained".

Loop variant

- integer expression that is non-negative after execution of **from** clause and after each execution of **loop** clause and strictly decreases with each iteration;

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all n elements in a list, on each iteration i ($i = 1..n$) the sum of first i elements is obtained".

Loop variant

- integer expression that is non-negative after execution of **from** clause and after each execution of **loop** clause and strictly decreases with each iteration;
- a loop with a correct variant can not be infinite

Invariant and variant

Loop invariant (do not confuse with class invariant)

- holds before and after the execution of loop body;
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all n elements in a list, on each iteration i ($i = 1..n$) the sum of first i elements is obtained".

Loop variant

- integer expression that is non-negative after execution of **from** clause and after each execution of **loop** clause and strictly decreases with each iteration;
- a loop with a correct variant can not be infinite (why?)

Sum of the first n integers (hands-on)

What are the loop invariants and variants here?

```
sum (n: INTEGER): INTEGER
        -- Compute the sum of the numbers from 0 to 'n'
    require 0 <= n
    local i: INTEGER
    do
        from
            Result := 0
            i := 1
        invariant
            ???
            ???
        until
            i > n
        loop
            Result := Result + i
            i := i + 1
        variant
            ???
        end
    ensure Result = (n * (n + 1)) // 2
    end
```

Sum of the first n integers (hands-on)

What are the loop invariants and variants here?

```
sum (n: INTEGER): INTEGER
        -- Compute the sum of the numbers from 0 to 'n'
    require 0 <= n
    local i: INTEGER
    do
        from
            Result := 0
            i := 1
        invariant
            1 <= i and i <= n+1
            ???
        until
            i > n
        loop
            Result := Result + i
            i := i + 1
        variant
            ???
        end
    ensure Result = (n * (n + 1)) // 2
    end
```

Sum of the first n integers (hands-on)

What are the loop invariants and variants here?

```
sum (n: INTEGER): INTEGER
        -- Compute the sum of the numbers from 0 to 'n'
    require 0 <= n
    local i: INTEGER
    do
        from
            Result := 0
            i := 1
        invariant
            1 <= i and i <= n+1
            Result = (i * (i - 1)) // 2
        until
            i > n
        loop
            Result := Result + i
            i := i + 1
        variant
            ???
        end
    ensure Result = (n * (n + 1)) // 2
    end
```

Sum of the first n integers (hands-on)

What are the loop invariants and variants here?

```
sum (n: INTEGER): INTEGER
        -- Compute the sum of the numbers from 0 to 'n'
    require 0 <= n
    local i: INTEGER
    do
        from
            Result := 0
            i := 1
        invariant
            1 <= i and i <= n+1
            Result = (i * (i - 1)) // 2
        until
            i > n
        loop
            Result := Result + i
            i := i + 1
        variant
            n - i + 1
        end
    ensure Result = (n * (n + 1)) // 2
    end
```

What does this function do? (hands-on)

```
??? (n: INTEGER): INTEGER
          -- ?????????
    require n >= 0
    local i: INTEGER
    do
        from
            Result := 1
            i := 2
        until
            i > n
        loop
            Result := Result * i
            i := i + 1
        end
    end
```

What does this function do? (hands-on)

```
??? (n: INTEGER): INTEGER
        -- ?????????
    require n >= 0
    local i: INTEGER
    do
        from
            Result := 1
            i := 2
        until
            i > n
        loop
            Result := Result * i
            i := i + 1
        end
    end
```

It calculates the factorial number of *n*.

Invariant and variant (hands-on)

What are the invariant and variant of the factorial loop?

```
from
    Result := 1
    i := 2
invariant
    ???
until
    i > n
loop
    Result := Result * i
    i := i + 1
variant
    ???
end
```

Invariant and variant (hands-on)

What are the invariant and variant of the factorial loop?

```
from
    Result := 1
    i := 2
invariant
    Result = factorial (i - 1)
until
    i > n
loop
    Result := Result * i
    i := i + 1
variant
    ???
end
```

Invariant and variant (hands-on)

What are the invariant and variant of the factorial loop?

```
from
    Result := 1
    i := 2
invariant
    Result = factorial (i - 1)
until
    i > n
loop
    Result := Result * i
    i := i + 1
variant
    n - i +2
end
```

Writing loops (hands-on)

Implement a function that calculates Fibonacci numbers, using a loop

```
fibonacci (n : INTEGER) : INTEGER
        -- n-th Fibonacci number
    require
        n_non_negative: n >= 0
    ensure
        first_is_zero: n = 0 implies Result = 0
        second_is_one: n = 1 implies Result = 1
        other_correct: n > 1 implies Result
                        = fibonacci (n - 1) + fibonacci (n - 2)
    end
```

Writing loops (solution)

```
do
    if n <= 1 then
        Result := n
    else
        from
            a := 0
            b := 1
            i := 1
        invariant
            a = fibonacci (i - 1)
            b = fibonacci (i)
        until i = n loop
            Result := a + b
            a := b
            b := Result
            i := i + 1
        variant
            n - i
        end
    end
end
```

Exercises

Exercises can be found in: `https://drive.google.com/open?id=0B1GMHm59JFjqMlBrWUVBUmg3YjA`.

Thank you!