Lecture 5: Invariants and Logic
Reminder: contracts

Associated with an individual feature:
- Preconditions
- Postconditions

Associated with a class:
- Class invariant
Contracts

remove_all_segments

-- Remove all stations except the south end.

ensure

only_one_left: count = 1
both_ends_same: south_end = north_end

extend(s: STATION)

-- Add s at end of line.

ensure

new_station_added: i_th(count) = s
added_at_north: north_end = s
one_more: count = old count + 1
Contracts

\[\text{deposit}(v : \text{INTEGER})\]
\[\quad -- \text{Add } v \text{ to account.}\]
\[\text{require}\]
\[\quad \text{positive: } v > 0\]
\[\text{do}\]
\[\quad \ldots\]
\[\text{ensure}\]
\[\quad \text{added: } balance = \text{old balance} + v\]
\[\text{end}\]
Class invariants

The invariant expresses consistency requirements between queries of a class

\[ \text{invariant} \]

\[
\text{south}\_\text{is}\_\text{first} : \text{south}\_\text{end} = i\_\text{th}\ (1) \\
\text{north}\_\text{is}\_\text{last} : \text{north}\_\text{end} = i\_\text{th}\ (\text{count})
\]
Applications of contracts

1. Getting the software right

2. Documenting it; in particular, documenting APIs

3. Testing & debugging

(More to come!)

Run-time effect: under compiler control (see Projects -> Settings under EiffelStudio)
Contracts outside of Eiffel

Java: Java Modeling Language (JML), iContract etc.

C#: Spec# (Microsoft Research extension)

UML: Object Constraint Language

Python

C++: Nana

etc.
Programming is reasoning.
Logic is the science of reasoning.

We use logic in everyday life:

"Socrates is human.
All humans are mortal.

Therefore Socrates must be mortal."
Reasoning and programming

Logic is the basis of

- **Mathematics**: proofs are only valid if they follow the rules of logic.

- **Software development**:
  - **Conditions in contracts**: “$x$ must not be zero, so that we can calculate $\frac{x+7}{x}$.”

  - **Conditions in program actions**: “If $i$ is positive, then execute this instruction” (to be introduced in a later lecture)
A condition is expressed as a boolean expression. It consists of

- **Boolean variables** (identifiers denoting boolean values)
- **Boolean operators** (**not**, **or**, **and**, **=**, **implies**)

and represents possible

- **Boolean values** (truth values, either **True** or **False**)
Examples of boolean expressions
(with `rain_today` and `cuckoo_sang_last_night` as boolean variables):

- `rain_today`
  (a boolean variable is a boolean expression)
- `not rain_today`
- `(not cuckoo_sang_last_night) implies rain_today`

(Parentheses group sub-expressions)
# Negation (not)

For any boolean expression \( e \) and any values of variables:

- Exactly one of \( e \) and \( \text{not} \ e \) has value \( \text{True} \)
- Exactly one of \( e \) and \( \text{not} \ e \) has value \( \text{False} \)
- One of \( e \) and \( \text{not} \ e \) has value \( \text{True} \) (Principle of the Excluded Middle)
- Not both of \( e \) and \( \text{not} \ e \) have value \( \text{True} \) (Principle of Non-Contradiction)

<table>
<thead>
<tr>
<th>( a )</th>
<th>not ( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
Disjunction (or)

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a , \text{or} , b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

- or operator is non-exclusive
- or operator is commutative

**Disjunction principle:**

- An or disjunction has value True except if both operands have value False
Conjunction (and)

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \text{ and } b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

The **and** operator is **commutative**.

**Duality** of **and** and **or**: properties of either operator yield properties of other (negating + swapping **True** and **False**)

**Conjunction principle:**
- An **and** conjunction has value **False** except if both operands have value **True**
Complex expressions

Build more complex boolean expressions by using the boolean operators

Example:

\[ a \text{ and } (b \text{ and } (\text{not } c)) \]
Truth assignment and truth table

Truth assignment for a set of variables: particular choice of values (True or False), for every variable

A truth assignment satisfies an expression if the value for the expression is True

A truth table for an expression with $n$ variables has

- $n + 1$ columns
- $2^n$ rows
### Combined truth table for basic operators

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>not $a$</th>
<th>$a$ or $b$</th>
<th>$a$ and $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Tautologies

**Tautology**: a boolean expression that has value *True* for every possible truth assignment

Examples:

- $a \text{ or } (\neg a)$
- $\neg (a \text{ and } (\neg a))$
- $(a \text{ and } b) \text{ or } ((\neg a) \text{ or } (\neg b))$
Contradictions

**Contradiction**: a boolean expression that has value $\text{False}$ for every possible truth assignment

Examples:
- $a$ and $(\text{not } a)$

**Satisfiable**: for at least one truth assignment the expression yields $\text{True}$
- Any tautology is satisfiable
- No contradiction is satisfiable.
### Equivalence (=)

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

- $=$ operator is commutative ($a = b$ has same value as $b = a$)
- $=$ operator is reflexive ($a = a$ is a tautology for any $a$)

**Substitution:**

- For any expressions $u$, $v$ and $e$, if $u = v$ is a tautology and $e'$ is the expression obtained from $e$ by replacing every occurrence of $u$ by $v$, then $e = e'$ is a tautology
De Morgan’s laws

De Morgan’s Laws: Tautologies

- \( \overline{a \lor b} = \overline{a} \land \overline{b} \)
- \( \overline{a \land b} = \overline{a} \lor \overline{b} \)

More tautologies (distributivity):

- \( a \land (b \lor c) = (a \land b) \lor (a \land c) \)
- \( a \lor (b \land c) = (a \lor b) \land (a \lor c) \)
Syntax convention: binding of operators

Order of binding (starting with tightest binding): not, and, or, implies (to be introduced), =.

and and or are associative:

- $a \text{ and } (b \text{ and } c) = (a \text{ and } b) \text{ and } c$
- $a \text{ or } (b \text{ or } c) = (a \text{ or } b) \text{ or } c$

Style rules:
When writing a boolean expression, drop the parentheses:
• Around the expressions of each side of “=” if whole expression is an equivalence.
• Around successive elementary terms if they are separated by the same associative operators.
Implication (implies)

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a$ implies $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

$a$ implies $b$, for any $a$ and $b$, is the value of (not $a$) or $b$

In $a$ implies $b$: $a$ is antecedent, $b$ consequent

Implication principle:

- An implication has value True except if its antecedent has value True and its consequent has value False
- In particular, always True if antecedent is False
Implication in ordinary language

implies in ordinary language often means causation, as in “if ... then ...”

- “If the weather stays like this, skiing will be great this week-end"

- “If you put this stuff in your hand luggage, they won’t let you through.”
Misunderstanding implication

Whenever \( a \) is **False**, \( a \) implies \( b \) is **True**, regardless of \( b \):

- “If today is Wednesday, 2+2=5.”
- “If 2+2=5, today is Wednesday.”

Both of the above implications are **True**

Cases in which \( a \) is **False** tell us nothing about the truth of the consequent
Reversing implications (1)

It is not generally true that

\[ a \implies b = (\neg a) \implies (\neg b) \]

Example (wrong!):

- “All the people in Zurich who live near the lake are rich. I do not live near the lake, so I am not rich.”

\[ \text{live} \_ \text{near} \_ \text{lake} \implies \text{rich} \] [1]

\[ (\neg \text{live} \_ \text{near} \_ \text{lake}) \implies (\neg \text{rich}) \] [2]
Reversing implications (2)

Correct:

\[ a \implies b = (\neg b) \implies (\neg a) \]

Example:

- "All the people who live near the lake are rich. She is not rich, so she can't be living in Küsnacht"

\[ \text{live\_near\_lake} \implies \text{rich} = \]

\[ (\neg \text{rich}) \implies (\neg \text{live\_near\_lake}) \]
Semistrict boolean operators (1)

Example boolean-valued expression (\(x\) is an integer):

\[
\frac{x + 7}{x} > 1
\]

False for \(x \leq -7\)

Undefined for \(x = 0\)
Semistrict boolean operators (2)

BUT:

- Division by zero: \( x \) must not be 0.

\[(x /= 0) \text{ and } (((x + 7) / x) > 1)\]

False for \( x \leq -7 \)
False for \( x = 0 \)
BUT:

- Program would crash during evaluation of division

We need a non-commutative version of and (and or):

Semistrict boolean operators
Semistrict operators (and then, or else)

\[ a \text{ and then } b: \text{ has same value as } a \text{ and } b \text{ if } a \text{ and } b \text{ are defined, and has } \text{False} \text{ whenever } a \text{ has value } \text{False} \]

\[ a \text{ or else } b: \text{ has same value as } a \text{ or } b \text{ if } a \text{ and } b \text{ are defined, and has } \text{True} \text{ whenever } a \text{ has value } \text{True} \]

\[ (x \neq 0) \text{ and then } (((x + 7) / x) > 1) \]

Semistrict operators allow us to define an order of expression evaluation (left to right).

Interesting for programming when undefined objects may cause program crashes
Ordinary vs. Semistrict boolean operators

Use

- Ordinary boolean operators (and and or) if you can guarantee that both operands are defined
- **and then** if a condition only makes sense when another is true
- **or else** if a condition only makes sense when another is false

Example:

- “If you are not single, then your spouse must sign the contract”
  
  `is_single or else spouse_must_sign`
Semistrict implication

Example:
- “If you are not single, then your spouse must sign the contract.”

\((\text{not is\_single}) \implies \text{spouse\_must\_sign}\)

Definition of **implies**: in our case, **always semistrict**!
- \(a \implies b = (\text{not } a) \text{ or else } b\)
### Programming language notation for boolean operators

<table>
<thead>
<tr>
<th>Eiffel keyword</th>
<th>Common mathematical symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>~ or ¬</td>
</tr>
<tr>
<td>or</td>
<td>∨</td>
</tr>
<tr>
<td>and</td>
<td>∧</td>
</tr>
<tr>
<td>=</td>
<td>⇔</td>
</tr>
<tr>
<td>implies</td>
<td>⇒</td>
</tr>
</tbody>
</table>
Propositional and predicate calculus

Propositional calculus:

property $p$ holds for a single object

Predicate calculus:

property $p$ holds for several objects
Generalizing or

\(G\): group of objects, \(p\): property

**or:** Does *at least one* of the objects in \(G\) satisfy \(p\)?

Is at least one station of Line 8 an exchange?

- \(\text{Station\_Balard.is\_exchange or Station\_Lourmel.is\_exchange or Station\_Boucicaut.is\_exchange or ... (all stations of Line 8)}\)

Existential quantifier: *exists*, or \(\exists\)

\[\exists \ s : \text{Stations\_8} \mid s.\text{is\_exchange}\]

"There exists an \(s\) in \text{Stations\_8} such that \(s.\text{is\_exchange}\) is true"
**Generalizing and**

**and:** Does *every* object in $G$ satisfy $p$? Are all stations of Tram 8 exchanges?

- $\text{Station\_Balard}.\text{is\_exchange}$ and
- $\text{Station\_Lourmel}.\text{is\_exchange}$ and
- $\text{Station\_Boucicaut}.\text{is\_exchange}$ and ...

(all stations of Line 8)

**Universal quantifier:** *for all*, or $\forall$

$\forall \ s: \text{Stations\_8} \ | \ s.\text{is\_exchange}$

“For all $s$ in $\text{Stations8} \ | \ s.\text{is\_exchange}$ is true”
Existentially quantified expression

Boolean expression:

\[ \exists s : SOME_SET \mid s.\text{some\_property} \]

- True if and only if at least one member of \( SOME_SET \) satisfies property \( \text{some\_property} \)

Proving

- True: Find one element of \( SOME_SET \) that satisfies the property
- False: Prove that no element of \( SOME_SET \) satisfies the property (test all elements)
Universally quantified expression

Boolean expression:

\[ \forall s: SOME_SET \mid s.some_{\text{property}} \]

- \textbf{True} if and only if every member of \textit{SOME\_SET} satisfies property \textit{some\_property}

Proving

- \textbf{True}: Prove that every element of \textit{SOME\_SET} satisfies the property (test all elements)
- \textbf{False}: Find one element of \textit{SOME\_SET} that does not satisfy the property
Duality

Generalization of DeMorgan’s laws:

\[ \overline{\exists s : \text{SOME\_SET} \mid P} = \forall s : \text{SOME\_SET} \mid \overline{P} \]

\[ \overline{\forall s : \text{SOME\_SET} \mid P} = \exists s : \text{SOME\_SET} \mid \overline{P} \]
Empty sets

\[ \exists s : \text{SOME\_SET} \mid \text{some\_property} \]
If \text{SOME\_SET} is empty: always False

\[ \forall s : \text{SOME\_SET} \mid \text{some\_property} \]
If \text{SOME\_SET} is empty: always True
Reading assignment for next week

Chapter 6 (object creation)
  Read corresponding slides (from Thursday)

Start reading chapter 7 (control structures)
What we have seen

- Logic as a tool for reasoning
- Boolean operators: truth tables
- Properties of boolean operators: don’t use truth tables!
- Predicate calculus: to talk about logical properties of sets
- SemistRICT boolean operators