Assertions

Assertions

- Boolean expressions or predicates that evaluate to **true** or **false** in every state
- In a program they express **constraints** on the state **that must be true** at that point
- Associate with
  - » **Individual program statements**
  - » **functions**
  - » **classes**

Assertions & Correct Programs

How to write correct programs and know it

– Harlan Mills

- Specify clearly, precisely and succinctly
  - » **What is expected and guaranteed by each component – class, function and statement**
- The essence of documentation
- Essential for debugging
- Aids in fault tolerance

Assertion Language Symbols

- Arithmetic operators

  + – * / ^ (exponent)

  // div (integer division)
  \\ mod (modulus / remainder)

- Relational operators

  = ≠ ≤ ≥ < >

- Boolean operators & logic

  ∧ and ∨ or ⊕ xor ¬ ~ not

  → implies ↔ iff

Assertion Language Symbols – 2

- Semi-strict **and** and **or** – Eiffel only for practical and efficiency reasons
  - » **Also called lazy evaluation in other programming languages**

and then

A and then B **Evaluate B only if A is true**

or else

A or else B **Evaluate B only if A is false**

Assertion Language Symbols – 3

- Predicate logic

  ∀ **forall** ∃ **exists (there exists)**

  | **such that**

  • **it is the case that (it holds that)**

- Set operators

  ∈ **member_of** ∉ **not_member_of**

  ⊃ ⊇ ⊂ ⊆ **contains**
  ⊄ **does_not_contain**

  ∩ **intersection** ∪ **union**

  #S number of members of the set S

Assertion Language Special Symbols

- Special variables related to program semantics

Result – result of a function but only in ensure assertions

Current @ – current object

Void – not attached

Variable before and after values

- Mathematical notation

  Unlimited context

  name

  value of the variable **name** before its value is changed

  name'

  value of the variable **name** after a its value has changed

- Eiffel notation

  Limited context

  name

  value of the variable **name** after a routine terminates

  old name

  value of the variable **name** before a routine starts

Quantified Expression

- Used to express properties about sets of objects

Quantifier Range_Expr [| Restriction] • Property

Quantifier ∀ forall ∃ exists (there exists)

Range_Expr var_name : set_of_values

Restriction Boolean expression or, recursively, a quantified expression

Property Boolean expression or, recursively, a quantified expression

Range Expression examples

- Type range – each value is of a given type

  v : VEHICLE

- Sequence range – each value is in a sequence

  k : low .. high

- Member range – each value is a member in a set

  c $\in$ children

Textual Notation example

```
class CITIZEN feature

   name, sex, age : VALUE
   spouse : CITIZEN
   children, parents : SET[CITIZEN]

   single : BOOLEAN ensure Result iff ( spouse = Void ) end

   divorce
      require  not single
      ensure  single and ( old spouse ) . single
   end

invariant
      single or spouse.spouse = Current
      parents.count = 2
      for_all  c member_of children  it_holds
         ( exists  p  member_of  c.parents  it_holds  p = Current)

end
```

Mathematical Notation example

```
class CITIZEN feature
    name, sex, age : VALUE
    spouse : CITIZEN
    children, parents : SET[CITIZEN]
    single : BOOLEAN ensure Result ↔ ( spouse = Void ) end
    divorce
        require  ~ single
        ensure single  ∧ spouse . Single
     end
invariant
        single ∨ spouse . spouse = @
        parents . count = 2
        ∀ c ∈ children  •  (∃ p ∈ c . parents • p = @)
}
```

Specifying Members of a Set

- Set enumeration – list the members

  $S = \{ a, e, i, o, u \}$

  The set of vowels in the English alphabet

- Set comprehension – logically specify members
  Notice that the forall is implicit not explicit

  $\{ x , y : Integer \mid (0 < x < 10) \land (1 \leq y \leq 9) \cdot x^3 + y^3 \}$

  The set of the sums of pairs of the cubes of single digit integers greater than zero

Pre-Conditions

- Statement syntax
  - » **require** **boolean expression**
- Where within function/procedure
  - » **write just before the local clause, if it exists**

```
nonZero ( row , col : INTEGER ) : BOOLEAN
-- Result true if non-zero element at <row, col>
require  0 < row and row <  MaximumRow + 1
          0 < col and col < MaximumCol + 1
do
...
end
```

Post-Conditions

- Statement syntax
  - » **ensure boolean expression**
- Where within function/procedure
  - » **write just before the end of body**

```
NonZero ( row , col : INTEGER ) : BOOLEAN
-- Result true if non-zero element at <row, col>
do
...
ensure  Result =
    ( search_by_row(row, col) /= void and
      search_by_row(row, col).data /= 0 )
end
```

State changes

- Show relationship between initial and final values
- At the end of the body the final values are in effect
- Refer to initial values using the keyword **old**

```
addElement ( element : TYPE )
require  size < Capacity
do

...

ensure size = old size + 1
end
```

Assertions are tagged

- Tag names are used to identify assertions

```
addElement ( element : TYPE )
require enough_space: size < Capacity
do

...

ensure one_larger: size = old size + 1
end
```

Non-executable assertions

- Use **comments** if you cannot write an executable assertion
- Use already defined functions or custom written functions

```
insert_in_row(matElem : MATRIX_ELEMENT)
    -- Insert the matrix element in the current row "row"
  require …
  local …
  do …
ensure
  -- contains(MatrixElement(data, row, column)) at < row, column >
end
```

Loop Invariants & Loop Syntax

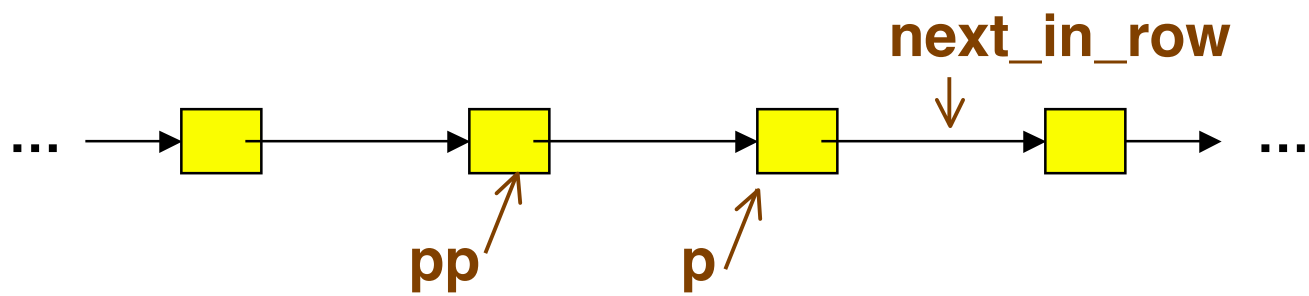
```
from
    init statements
invariant
    assertions for invariant
until
    exit condition
loop
    body statements
variant
    integer expression
end
```

(assertions for invariant)
- Can invoke Boolean functions
- Use agents to implement predicate calculus expressions

(integer expression)
- Always non negative
- Body decreases value on every iteration
- Ideally 0 on loop exit

Loop Invariant Example

- Inserting an element into a sorted singly linked list




```
row := matElem.row ; column := matElem.column
from   p := rowList @ row
invariant  ???
until
  p = void or p.column >= column
loop
  pp := p   ;   p := p.next_in_row
end
```

Loop Invariant Example – 2

- Using mathematical notation

invariant

predecessor_relation: $(pp = void \land p = head)$

$\lor\ (pp \neq void \land pp.next = p)$

predecessor_before_data: $pp \neq void \land pp.data < data$

data_less_than: $\forall\ k : head .. pp \bullet k.data < data$

Loop Invariant Example – 3

- Eiffel executable assertion.
- Column_less_than uses an agent to implement the invariant
  - > Agents and loop invariants are discussed in other slides

```
from   p := rowList @ row
invariant
  predecessor_relation :      (pp = void and p = rowList @ row)
                        or  (pp /= void and pp.next_in_row = p)

  predecessor_before_column: pp = void or pp.column < column

 -- forall k : rowList @ row ..  pp  ::  k.column  <  column
  data_less_than : column_limit( rowList @ row, pp,
                                         agent less_than(?, column) )
end
```

Check Assertion

- Within the body of a routine you can insert a **check** clause
- The **check** clause is executed and if an assertion is false then an exception occurs
- Used to remind the reader of a non obvious fact that could be deduced

```
If full then  error := overflow
else
  check
     representation_exists : representation /= Void
  end
  representation.put(x) ; error := none
end
```

Class Invariants

- Appear in the **invariant** clause just before the end of the class definition

```
class SPARSE_MATRIX
...
invariant
    actualRows <= maxRowCol
    actualCols <= maxRowCol
    -- forall row : maxNonzeroRow + 1 .. actualRows
    --          ::  empty ( rowList [ row ] )
    -- forall col : maxNonzeroCol + 1 .. actualCols
    --          ::  empty ( colList [ col ] )
end -- SPARSE_MATRIX
```

Class Invariants – 2

- Class invariants define which states of the ADT are valid

- True at stable times
  - » **After make (object creation)**
  - » **After every exported feature call**
    - > **Could be false during a feature call as various sub-states change**

- Invariant is implicitly a part of every pre and post condition

Class Invariants – examples

- See slides 9 & 10 in this set of slides
  - » **Relationship between parents and children**
  - » **Relationship between spouses**

- See Abstract data type documentation slides 18 .. 23
  - » **Relationship between first and last pointers in a circular queue and the length of the queue**

General Guideline

- Assertions may be written in many ways
  - » **Select the representation to be as clear and easy to understand as possible**
    - > **Point is to convey information, not provide a puzzle to be solved**
  - » **Use notation that is close to the meaning of the relationships involved – no need to restrict to first order predicate calculus**
    - > **Set notation**
    - > **Bag notation**
    - > **Sequence notation**

Assertion Monitoring

- Eiffel provides multiples levels of assertion monitoring
  - » **See the project settings & page 393**
- Always should be on during debugging
- Turn off as little as possible only if time is critical and the system can be trusted