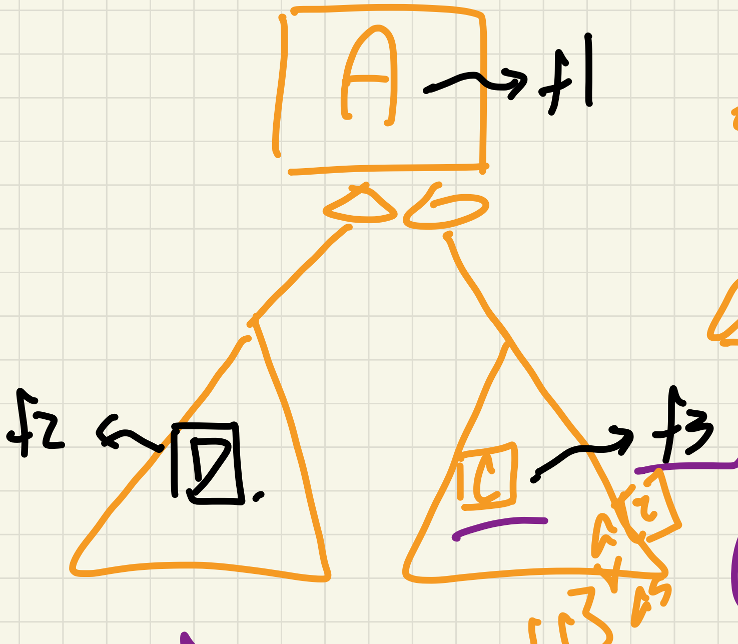
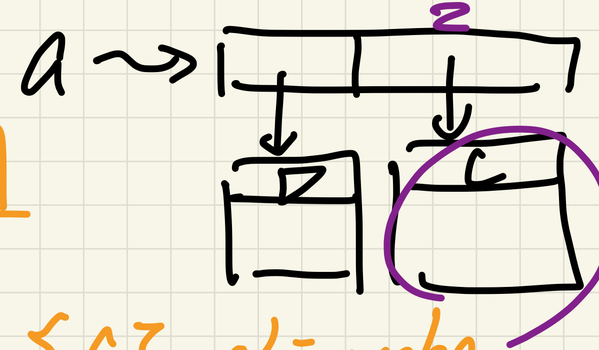
Exam Review I

Monday December 9





obj : A

create {C} obj.make

@ a: ARRAY[A]

a[i] := ?

across a is obj
loop
obj.? f1
end

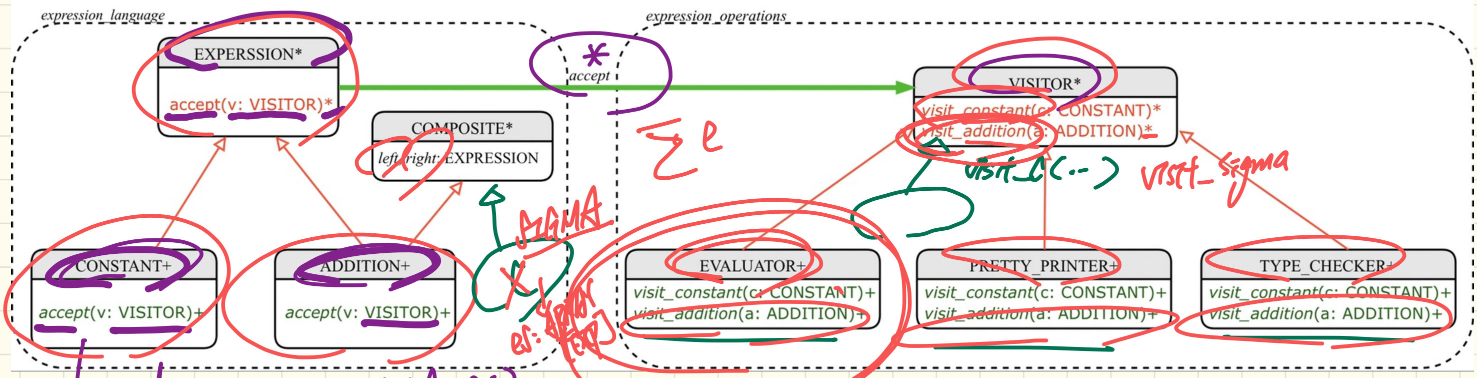
a[i].f3

i

ST: A

check attached {C} a[i] as c_obj then
c_obj.f3
end

Visitor Design Pattern: Architecture



do
v.visit_constant(current)
end

How to Use Visitors

```
test_expression_evaluation: BOOLEAN
  local add, c1, c2: EXPRESSION ; v: VISITOR
  do
    create {CONSTANT} c1.make (1) ; create {CONSTANT} c2.make (2)
    create {ADDITION} add.make (c1, c2)
    create {EVALUATOR} v.make
    add.accept (v)
    check attached {EVALUATOR} v as eval then
     Result := eval.value = 3
    end
  end
```

Visitor Design Pattern: Implementation

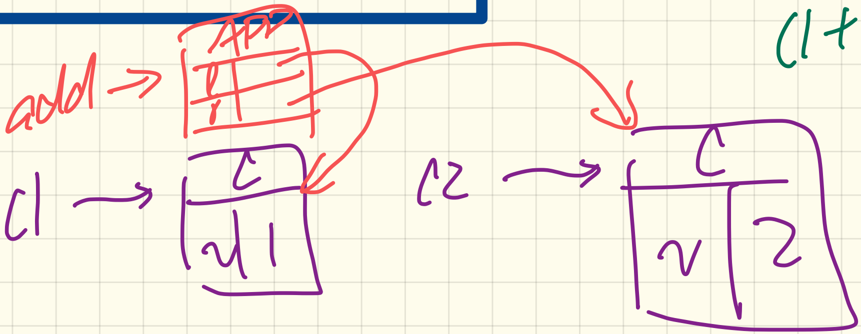
```
1  test_expression_evaluation: BOOLEAN
2    local add, c1, c2: EXPRESSION ; v: VISITOR
3    do
4      create {CONSTANT} c1.make (1) ; create {CONSTANT} c2.make (2)
5      create {ADDITION} add.make (c1, c2)
6      create {EVALUATOR} v.make
7      add.accept (v)
8      check attached {EVALUATOR} v as eval then
9        Result := eval.value = 3
10     end
11   end
```

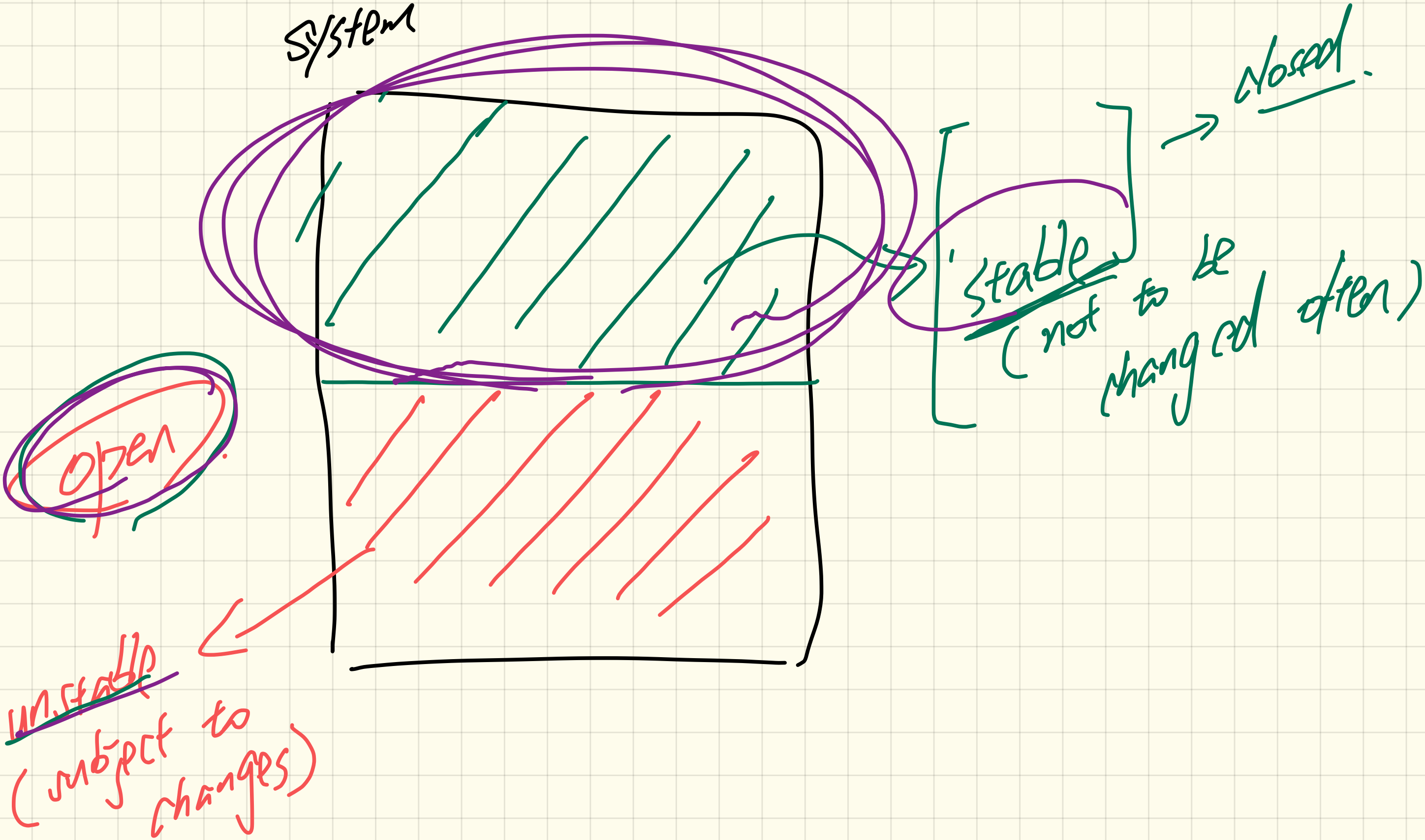
composite

visitor

Write a fragment of code which builds: $(1+2)+(3+4)$

Visualizing Line 4 to Line 6




system
closed.
stable
(not to be changed often)
open
unstable
(subject to changes)

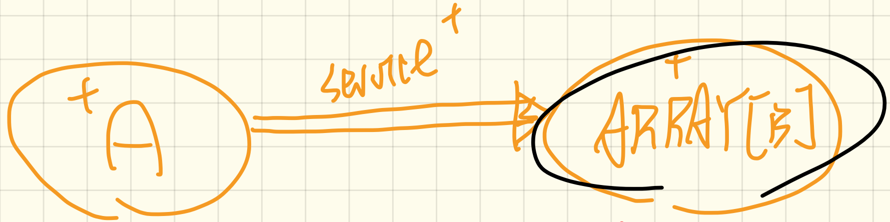
```
class A
    service: ARRAY[B]
    :
end
```

supplier

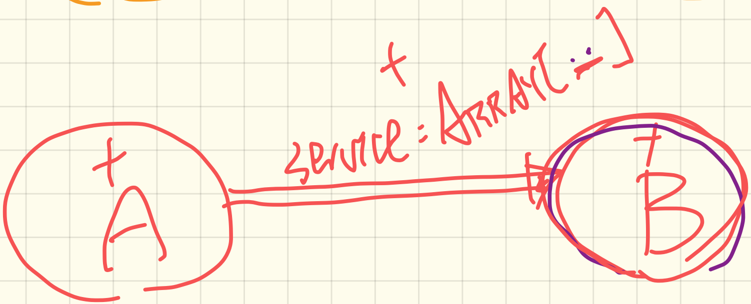
supplier

```
class B
    :
end
```

①



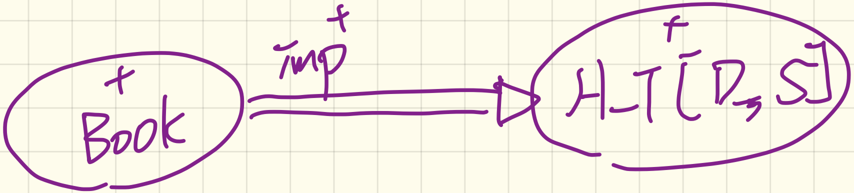
②



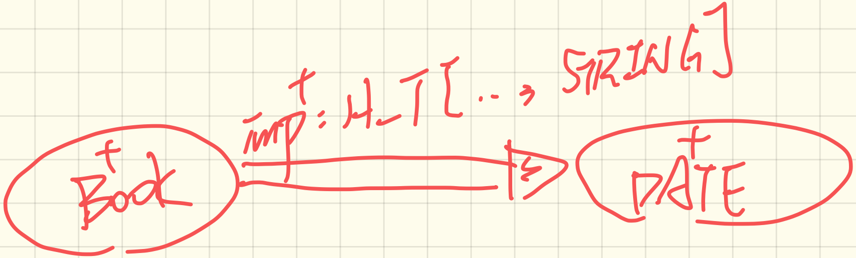
class Book
imp : HASH_TABLE[DATE, STRING]

end

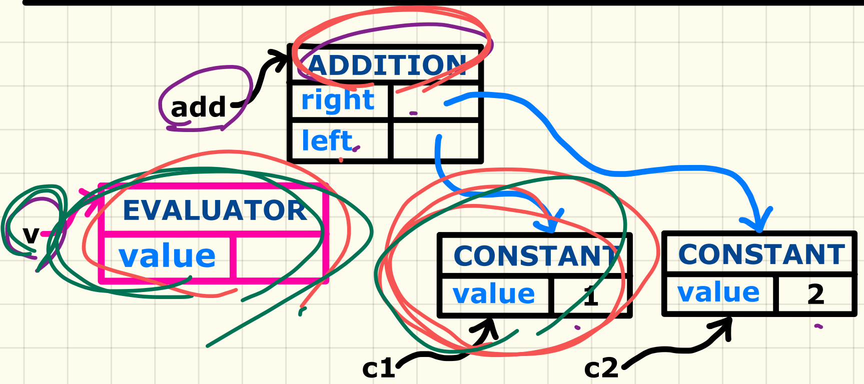
1



2



Executing Composite and Visitor Patterns at Runtime



Tracing add.accept(v)

Double Dispatch

add.accept(v)
↳ DT of add: ADDITION
⇒ call accept in ?
↳ DT of v: EVALUATOR
⇒ call visit_addition in ?

```
deferred class VISITOR
  visit_constant(c: CONSTANT) deferred end
  visit_addition(a: ADDITION) deferred end
end
```

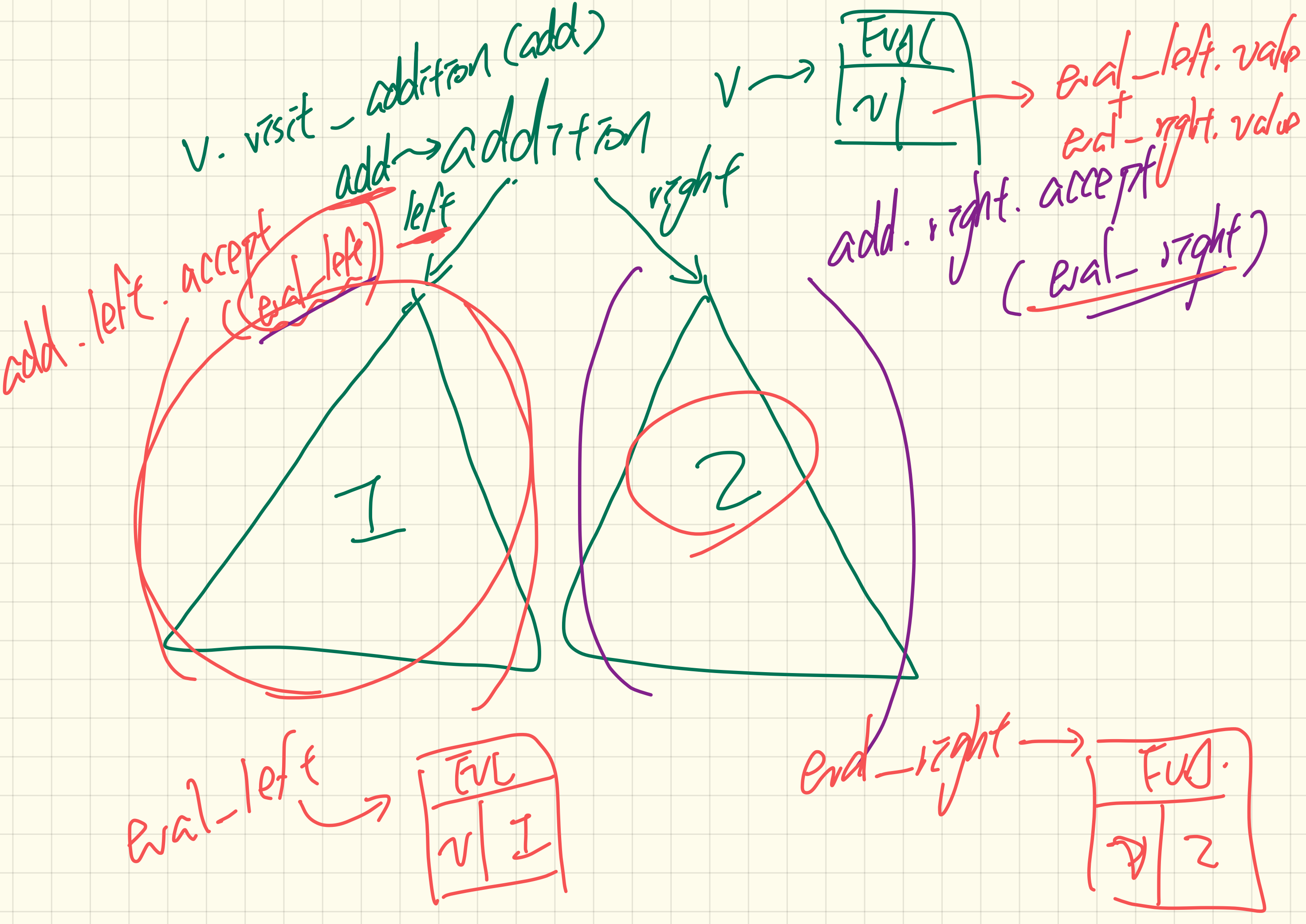
```
class EVALUATOR inherit VISITOR
  value : INTEGER
  visit_constant(c: CONSTANT)  do value := c.value end
  visit_addition(a: ADDITION)
    local eval_left, eval_right: EVALUATOR
    do a.left.accept(eval_left)
       a.right.accept(eval_right)
       value := eval_left.value + eval_right.value
    end
end
```

double dispatch

double dispatch

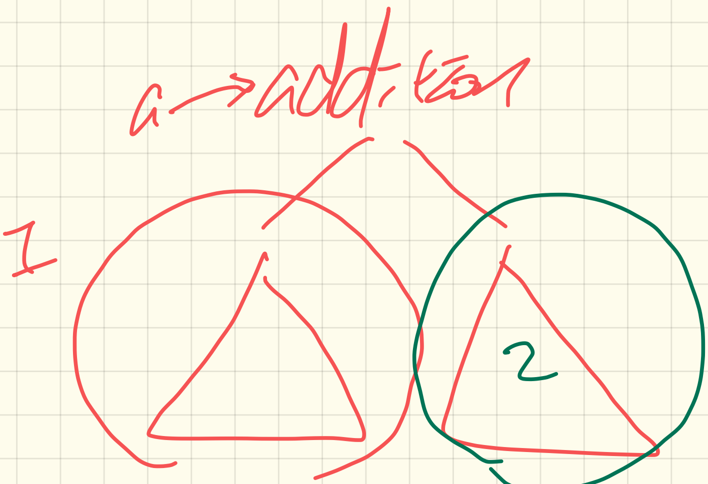
```
class CONSTANT inherit EXPRESSION
...
  accept(v: VISITOR)
    do
      v.visit_constant(Current)
    end
end
```

```
class ADDITION
inherit EXPRESSION COMPOSITE
...
  accept(v: VISITOR)
    do
      v.visit_addition(Current)
    end
end
```


v.visit_addition(add)
add → addition
left
right
v → Eval
v
eval_left.val + eval_right.val
add.left.accept(eval_left)
add.right.accept(eval_right)
1
2
eval_left → Eval
v
1
eval_right → Eval.
v
2

2 X

```
class EVALUATOR inherit VISITOR
  value: INTEGER
  visit_constant(c: CONSTANT) do value := c.value end
  visit_addition(a: ADDITION)
    local eval_left, eval_right: EVALUATOR
    do a.left.accept(eval_left) Current
       a.right.accept(eval_right) Current
       value := eval_left.value + eval_right.value
    end
end
```



```
class BANK
  accounts: ARRAY[ACCOUNT]
  withdraw_from (i: INTEGER; a: INTEGER)
      -- Withdraw amount 'a' from account stored as the 'i'th item in 'accounts'.
    require
      positive_amount: a > 0
      enough_balance: accounts.valid_index (i) and accounts [i].balance > a
    do
      accounts[i].withdraw (a)
    end
end
```

and

and then

&&

-1

-1

balance_cond: accounts[i].balance > a

→ valid_index: accounts.valid_index(i)

require

p1 $i >= 0$ and $i <= count$

p2 $a[i] > 0$

⋮

if there's dependency among preconditions, put the least-depending first.

p1 and then p2

ensure

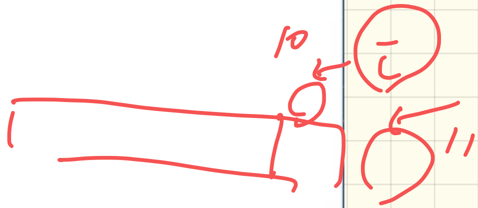
q1

q2

```
class DICTIONARY[V, K]
feature {NONE} -- Implementations
  values: ARRAY[K]
  keys: ARRAY[K]
feature -- Abstraction Function
  model: FUN[K, V]
feature -- Queries
  get_keys(v: V): ITERABLE[K]
    local i: INTEGER; ks: LINKED_LIST[K]
    do
      from i := keys.lower ; create ks.make_empty
      invariant ??
      until i > keys.upper
      do if values[i] ~ v then ks.extend(keys[i]) end
      end
      Result := ks.new_cursor
    ensure
      result_valid: ∀k | k ∈ Result • model.item(k) ~ v
      no_missing_keys: ∀k | k ∈ model.domain • model.item(k) ~ v ⇒ k ∈ Result
    end
```

lower 1

keys →



variant

i := i+1

i

keys.upper - i + 1

10 - 11 + 1

```
class DICTIONARY[V, K]
feature {NONE} -- Implementations
  values: ARRAY[K]
  keys: ARRAY[K]
feature -- Abstraction Function
  model: FUN[K, V]
feature -- Queries
  get_keys(v: V): ITERABLE[K]
    local i: INTEGER; ks: LINKED_LIST[K]
    do
      from i := keys.lower ; create ks.make_empty
      invariant ??
      until i > keys.upper
      do if values[i] ~ v then ks.extend(keys[i]) end
      end
      Result := ks.new_cursor
    ensure
      result_valid: ∀k | k ∈ Result • model.item(k) ~ v
      no_missing_keys: ∀k | k ∈ model.domain • model.item(k) ~ v ⇒ k ∈ Result
    end
```

PO2. Assuming LI to exit, not ready after the end of iteration, LI is maintained.

$\{\neg(i > keys.upper) \wedge ??\}$

$\{LI ??\}$

PO1: Init establishes the LI.

$\{True\}$ $i := keys.lower$; create ks.m_e $\{??\}$

Correct Loops: Proof Obligations

```
find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: ∀j | a.lower ≤ j < i • Result ≥ a[j]
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper − i + 1
    end
  ensure
    correct_result: ∀j | a.lower ≤ j ≤ a.upper • Result ≥ a[j]
  end
end
```

ทำไมถึงเป็นจริง

I

B

Initialization:

Before Termination:

Upon Termination:

Non-Negative Variant:

Decreasing Variant:

Prove

$1 \le j < 1$ $\quad \forall x \mid F(x) \cdot \bot(x) \equiv T$

$wp(S_1 ; S_2, R) = wp(S_1, wp(S_2, R))$

Establishment of Loop Invariant:

{ *True* }
i := a.lower
Result := a[i]
{ $\forall j \mid a.lower \le j < i \bullet Result \ge a[j]$ }

① Calculate $wp(i := a.lower ; Result := a[i],$
$\forall j \mid a.lower \le j < i \cdot Result \ge a[j])$

$= \{ wp \text{ rule on } ; \}$

$wp(i := a.lower, wp(Result := a[i],$
$\forall j \mid a.lower \le j < i \cdot Result \ge a[j]))$

$= \{ wp \text{ rule of } ; \}$

$wp(i := a.lower, \forall j \mid a.lower \le j < \cancel{i}^{\,a.lower} \cdot a[i] \ge a[j])$

$= \{ wp \text{ on } ; \}$

$\forall j \mid a.lower \le j < a.lower \cdot a[i] \ge a[j]$

$\equiv True$

from
:
invariant I ← loop invariant
until
B ← exit condition
→ constraint on loop counter.
loop
:
end
ensure R ← postcondition

PO: Upon termination, given that I is maintained, postcondition is established.

$$B \wedge I \Rightarrow R$$

from

until

loop

end

while (...) {

}

I := C.new_cursor

across C as ~~I~~ i

loop

end

names.count = old names.count + 1

integer

names.count = (old names.df.count + 1