ETH Zürich
Chair of Software Engineering

Software Verification

Bertrand Meyer

Lecture 2: Axiomatic semantics

Axiomatic semantics

Floyd (1967), Hoare (1969), Dijkstra (1978)

Purpose:

- Describe the effect of programs through a theory of the underlying programming language, allowing proofs

Prelude: an exercise (by Leslie Lamport)

http://research.microsoft.com/en-us/um/people/lamport/pubs/teaching-concurrency.pdf

Consider N processes numbered from 0 through N-1 where process i executes

$x[i] := 1;$
$y[i] := x[i-1]$ -- Subtraction modulo N

and stops. All $x[i]$ are initially 0; the reads and writes of each $x[i]$ are atomic.

(Continued on next slide)

Exercise (continued)

-- All $x[i]$ initially 0
$x[i] := 1$;
$y[i] := x[i-1]$ -- mod N

This algorithm satisfies the following property:

After every process has stopped, $y[i]$ equals 1 for at least one process i.

It is easy to see why: the last process i to write $y[i]$ must set it to 1.

But this is not a rigorous argument! The last process does not set $y[i]$ to 1 "*because*" it was the last process to write to y. It does not *know* that it is the last one to do so. More generally, what a process does depends only on the current state, not on some insider knowledge of what other processes did before it!

The algorithm satisfies the property **because it maintains an invariant**. Do you see that invariant?

What is a theory?

(Think of any mathematical example, e.g. elementary arithmetic)

A theory is a mathematical framework for proving properties about a certain object domain

Such properties are called theorems

Components of a theory:

- Grammar (e.g. BNF), defines well-formed formulae (WFF)
- Axioms: formulae asserted to be theorems
- Inference rules: ways to prove new theorems from previously obtained theorems

Notation

Let f be a well-formed formula

Then

$$\vdash f$$

expresses that f is a theorem

Inference rule

An inference rule is written

$$\frac{f_1, \quad f_2, \ldots, \; f_n}{f_0}$$

It expresses that if $f_1, \quad f_2, \ldots \; f_n$ are theorems, we may infer f_0 as another theorem

Example inference rule

"Modus Ponens" (common to many theories):

$$\frac{p, \quad p \Rightarrow q}{q}$$

How to obtain theorems

Theorems are obtained from the axioms by zero or more* applications of the inference rules.

*Finite of course

Example: a simple theory of integers

Grammar: Well-Formed Formulae are boolean expressions

- $i1 = i2$
- $i1 < i2$
- $\neg$ b1
- b1 $\Rightarrow$ b2

where b1 and b2 are boolean expressions, i1 and i2 integer expressions

An integer expression is one of

- 0
- A variable n
- f' where f is an integer expression (represents "successor")

An axiom and axiom schema

$\vdash 0 < 0'$

$\vdash f < g \Rightarrow f' < g'$

An inference rule

$$\frac{P\,(0), \quad P\,(f) \Rightarrow P\,(f')}{P\,(f)}$$

The theories of interest

Grammar: a well-formed formula is a "Hoare triple"

Instructions

{P} *A* {Q}

Assertions

Informal meaning: *A*, started in any state satisfying P, will satisfy Q on termination

Software correctness (a quiz)

Consider

$\{P\}\ A\ \{Q\}$

Take this as a job ad in the classifieds

Should a lazy employment candidate hope for a weak or strong P? What about Q?

Two "special offers":

- 1. $\{False\}\ A\ \{...\}$
- 2. $\{...\}\ A\ \{True\}$

Application to a programming language: Eiffel

```
extend (new: G; key: H)
        -- Assuming there is no item of key key,
        -- insert new with key; set inserted.
    require
        key_not_present: not has (key)
    ensure
        insertion_done: item (key) = new
        key_present: has (key)
        inserted: inserted
        one_more: count = old count + 1
```

Partial vs total correctness

$$\{P\} \quad A \quad \{Q\}$$

Total correctness:

- *A*, started in any state satisfying P, will terminate in a state satisfying Q

Partial correctness:

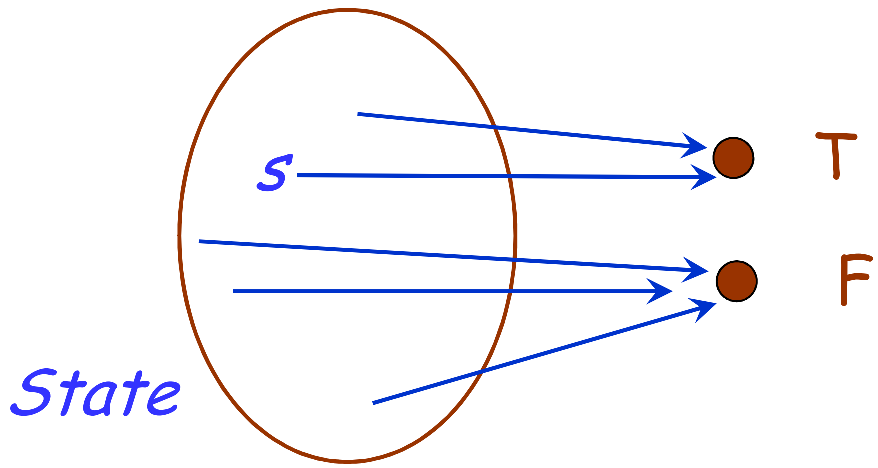
- *A*, started in any state satisfying P, will, *if it terminates*, yield a state satisfying Q

Axiomatic semantics

"Hoare semantics" or "Hoare logic": a theory describing the partial correctness of programs, plus termination rules

What is an assertion?

Predicate (boolean-valued function) on the set of computation states



True: Function that yields T for all states

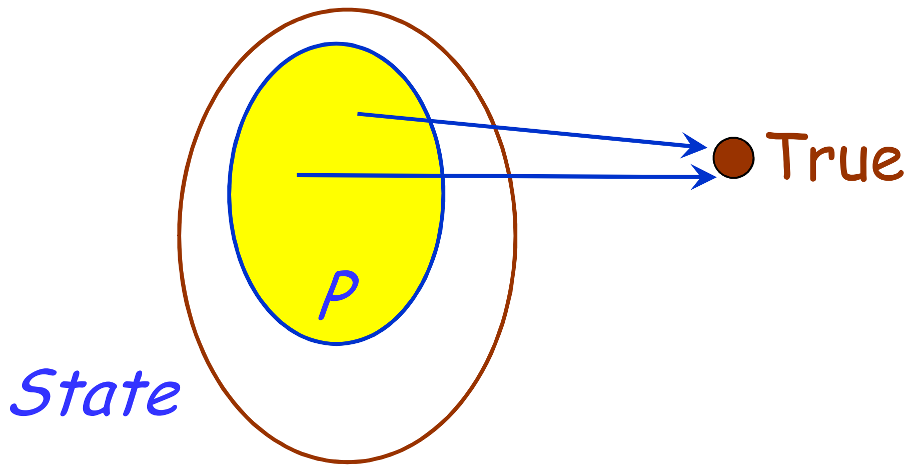
False: Function that yields F for all states

P **implies** Q: means $\forall\ s: State,\ P(s) \Rightarrow Q(s)$

and so on for other boolean operators

Another view of assertions

We may equivalently view an assertion P as a subset of the set of states (the subset where the assertion yields True):



True: Full *State* set

False: Empty subset

implies: subset (inclusion) relation

and: intersection **or**: union

The case of postconditions

Postconditions are often predicates on two states

Example (Eiffel, in a class *COUNTER*):

```
increment
        require
                count >= 0
        ...
        ensure
                count  =  old count + 1
```

Elementary mathematics

Assume we want to prove, on integers

$$\{x > 0\} \; A \; \{y \geq 0\} \qquad [1]$$

but have actually proved

$$\{x > 0\} \; A \; \{y = z \,\hat{}\, 2\} \qquad [2]$$

We need properties from other theories, e.g. arithmetic

"EM": Elementary Mathematics

The mark [EM] will denote results from other theories, taken (in this discussion) without proof

Example:

$y = z \hat{}\ 2$ **implies** $y \geq 0$ [EM]

Rule of consequence

$$\frac{\{P\}\ A\ \{Q\}, \quad P' \textbf{ implies } P, \quad Q \textbf{ implies } Q'}{\{P'\}\ \ A\ \ \{Q'\}}$$

Rule of conjunction

$$\frac{\{P\}\ A\ \{Q\},\quad \{P\}\ A\ \{R\}}{\{P\}\ \ A\ \ \{Q \textbf{ and } R\}}$$

Axiomatic semantics for a programming language

Example language: Graal (from *Introduction to the theory of Programming Languages*)

Scheme: give an axiom or inference rule for every language construct

Skip

$$\{P\} \;\; \textbf{skip} \;\; \{P\}$$

Abort

$\{False\}$ **abort** $\{P\}$

Sequential composition

$$\frac{\{P\}\ A\ \{Q\}, \qquad \{Q\}\ B\ \{R\}}{\{P\}\ \ A\ ;\ B\ \ \{R\}}$$

Assignment axiom (schema)

$$\{P\,[e\,/\,x]\} \qquad x := e \qquad \{P\}$$

$P\,[e/x]$ is the expression obtained from P by replacing (substituting) every occurrence of x by e.

Substitution

$x\,[x/x] =$

$x\,[y/x] =$

$x\,[x/y] =$

$x\,[z/y] =$

$3 * x + 1\,[y/x] =$

Applying the assignment axiom

$\{y > z - 2\}\ x := x + 1\ \{y > z - 2\}$

$\{2 + 2 = 5\}\ x := x + 1\ \{2 + 2 = 5\}$

$\{y > 0\}\ x := y\ \{x > 0\}$

$\{x + 1 > 0\}\ x := x + 1\ \{x > 0\}$

Limits to the assignment axiom

No side effects in expressions!

```
asking_for_trouble (x: in out INTEGER): INTEGER
    do
        x := x + 1;
        global := global + 1;
        Result := 0
    end
```

Dothe following hold?

```
{global = 0}  u := asking_for_trouble (a)   {global = 0}
{a = 0}       u := asking_for_trouble (a)   {a = 0}
```

Conditional rule

$$\frac{\{P \textbf{ and } c\}\ A\ \{Q\}, \qquad \{P \textbf{ and not } c\}\ B\ \{Q\}}{\{P\}\quad \textbf{if } c \textbf{ then } A \textbf{ else } B \textbf{ end } \{Q\}}$$

Conditional rule: example proof

Prove:

{$m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd(x, y) = gcd(m, n)$}

if $x > y$ **then**
    $x := x - y$
else
    $y := y - x$
end

{$m, n, x, y > 0$ **and** $gcd(x, y) = gcd(m, n)$}

Loop rule (partial correctness)

$$\frac{\{P\}\ A\ \{I\}, \qquad \{I \text{ and not } c\}\ B\ \{I\},}{\{P\}\ \textbf{from}\ A\ \textbf{until}\ c\ \textbf{loop}\ B\ \textbf{end}\ \{I \text{ and } c\}}$$

Slight variant

$$\frac{\{P\}\ A\ \{I\}, \qquad \{I \textbf{ and not } c\}\ B\ \{I\},\ (I \textbf{ and } c) \textbf{ implies } Q}{\{P\}\ \textbf{from } A \textbf{ until } c \textbf{ loop } B \textbf{ end } \{Q\}}$$

Loop rule (partial correctness, another form)

$$\frac{\{P\}\ A\ \{I\},\quad \{I \textbf{ and not } c\}\ B\ \{I\},\quad \{(I \textbf{ and } c) \textbf{ implies } Q\}}{\{P\}\quad \textbf{from}\ A\ \textbf{until}\ c\ \textbf{loop}\ B\ \textbf{end}\quad \{Q\}}$$

Loop termination

Must show there is a variant:

Expression v of type INTEGER such that
(for a loop **from** A **until** c **loop** B **end** with precondition P):

1. {P} A {v ≥ 0}

2. ∀ v0 > 0:
   {v = v0 **and not** c} B {v < v0 **and** v ≥ 0}

Computing the maximum of an array

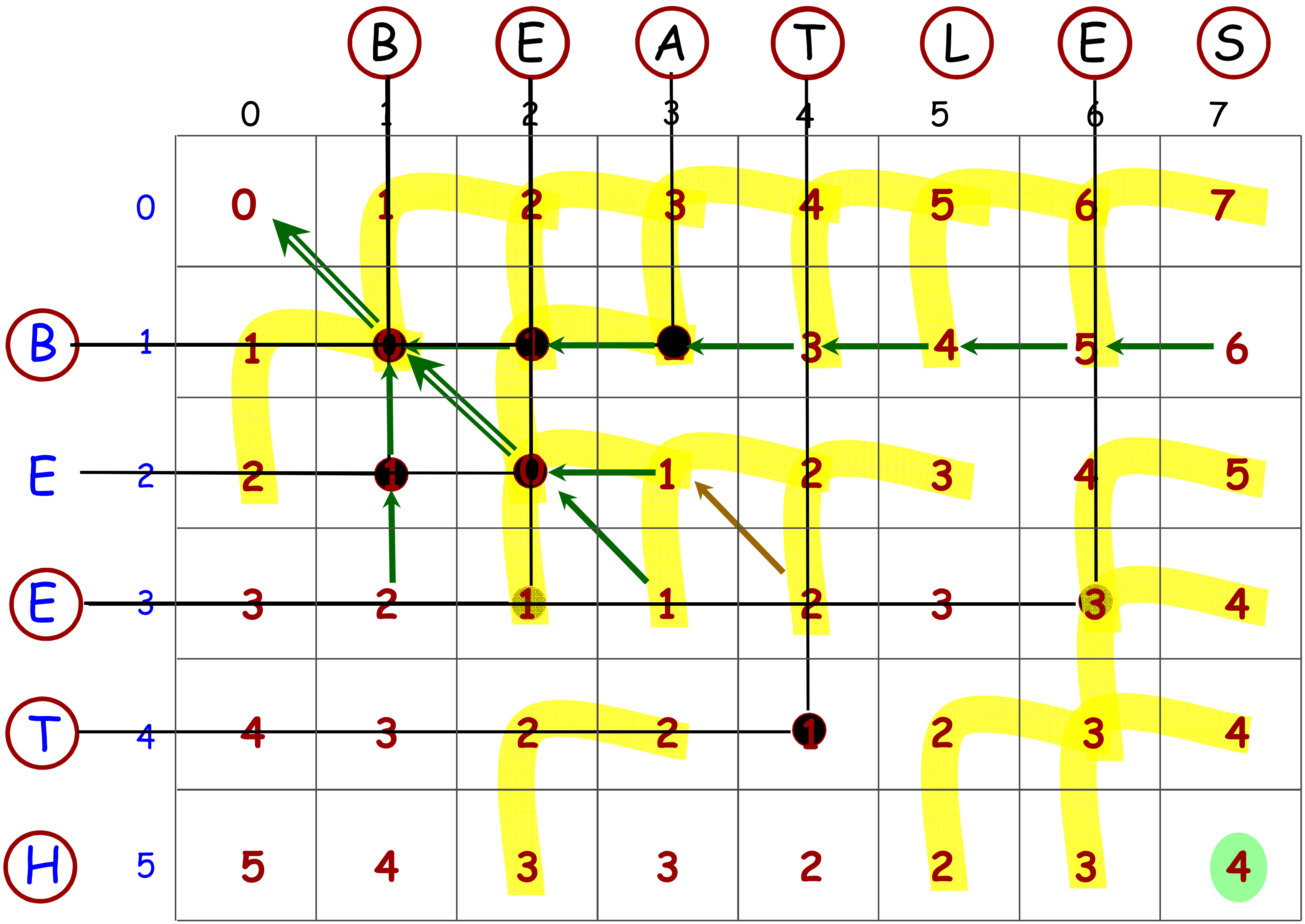
```
from
        i := 0 ; Result := a [1]
until

        i = a.upper
loop
        i := i + 1
        Result := max (Result , a [i])
end
```

Levenshtein distance

"Beethoven" to "Beatles"

	B	E	E	T	H	O	V	E	N
			A			L			S
Operation	–	–	R	–	D	R	D	–	R
Distance	0	0	1	1	2	3	4	4	5


B E A T L E S
0 1 2 3 4 5 6 7
B
E
E
T
H
0 1 2 3 4 5

Levenshtein distance algorithm

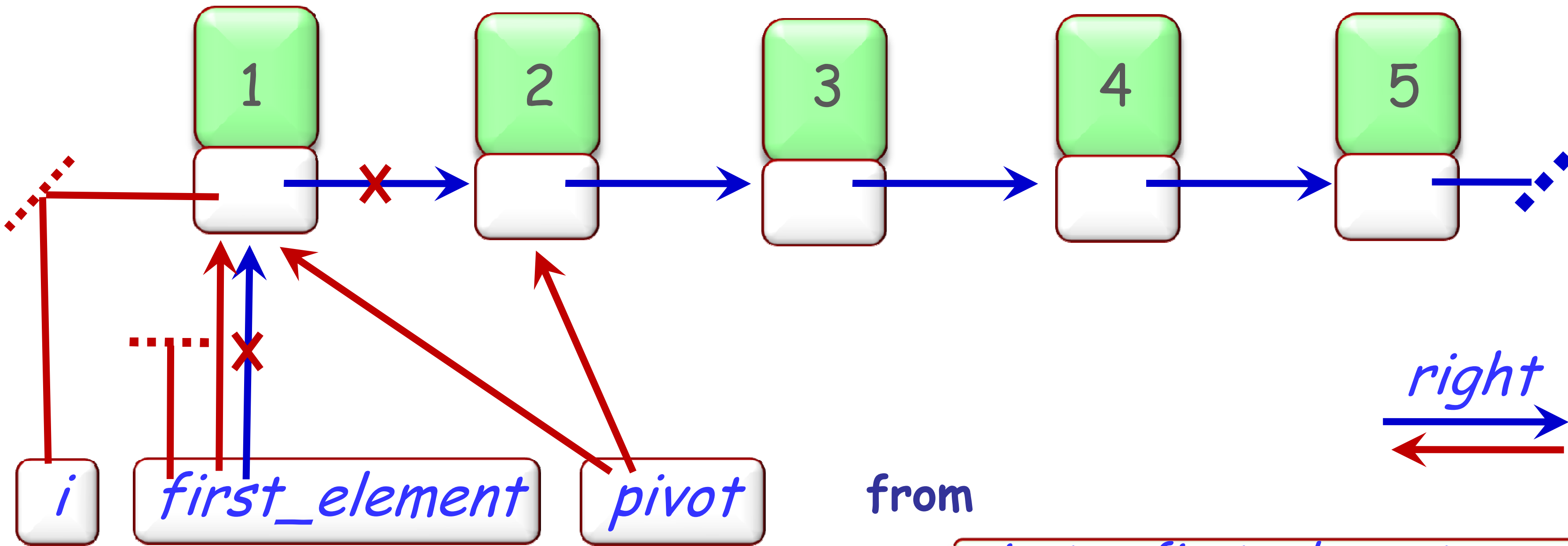
```
distance (source, target: STRING): INTEGER
        -- Minimum number of operations to turn source into target
    local
        dist: ARRAY_2 [INTEGER]
        i, j, del, ins, subst : INTEGER
    do
        create dist.make (source.count, target.count)
        from i := 0 until i > source.count loop
            dist [i, 0] := i ; i := i + 1
        end

        from j := 0 until j > target.count loop
            dist [0, j ] := j ; j := j + 1
        end
        -- (Continued)
```

Levenshtein distance algorithm

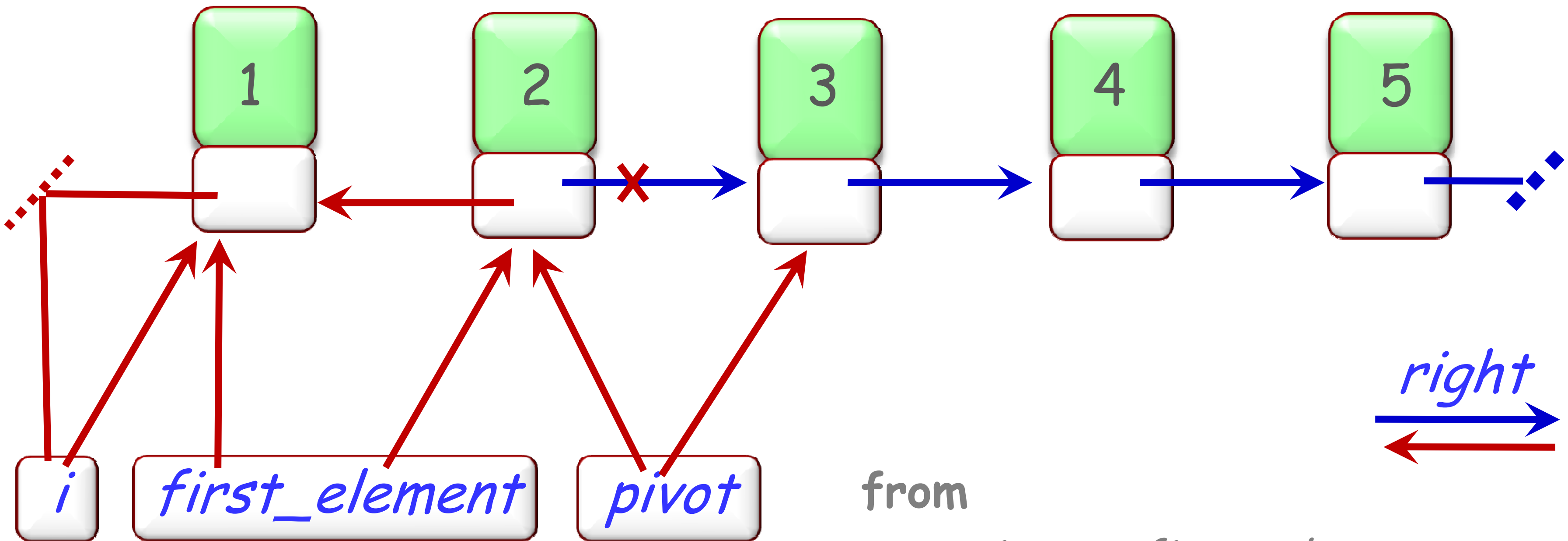
```
from i := 1 until i > source.count loop
  from j := 1 until j > target.count invariant
     ???
  loop
     if source [i ] = target [ j ] then
          dist [i, j ] := dist [ i -1, j -1]
     else
          deletion := dist [i -1, j ]
          insertion := dist [i , j - 1]
          substitution := dist [i - 1, j - 1]
          dist [i, j ] := minimum (deletion, insertion, substitution) + 1
     end
     j := j + 1
  end
  i := i + 1
end
Result := dist (source.count, target.count)
end
```

Reversing a list



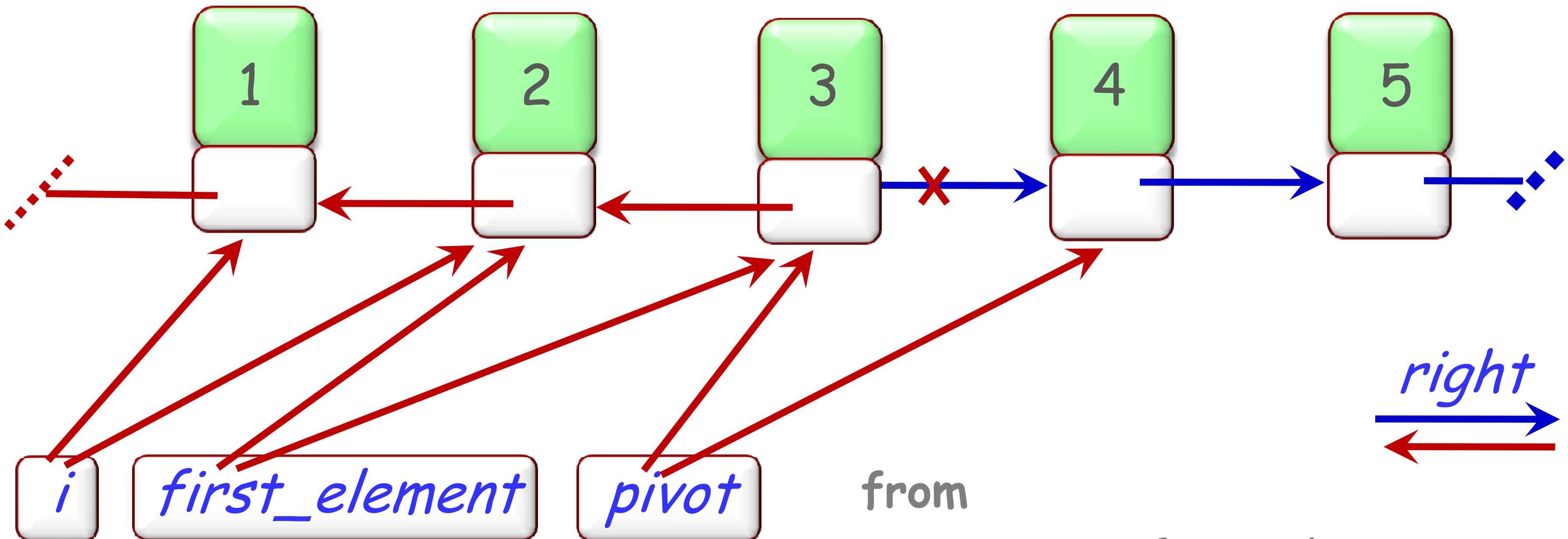
from
    pivot := *first_element*
    first_element := **Void**
until *pivot* = **Void** **loop**
    i := *first_element*
    first_element := *pivot*
    pivot := *pivot.right*
    first_element.put_right (*i*)
end

Reversing a list



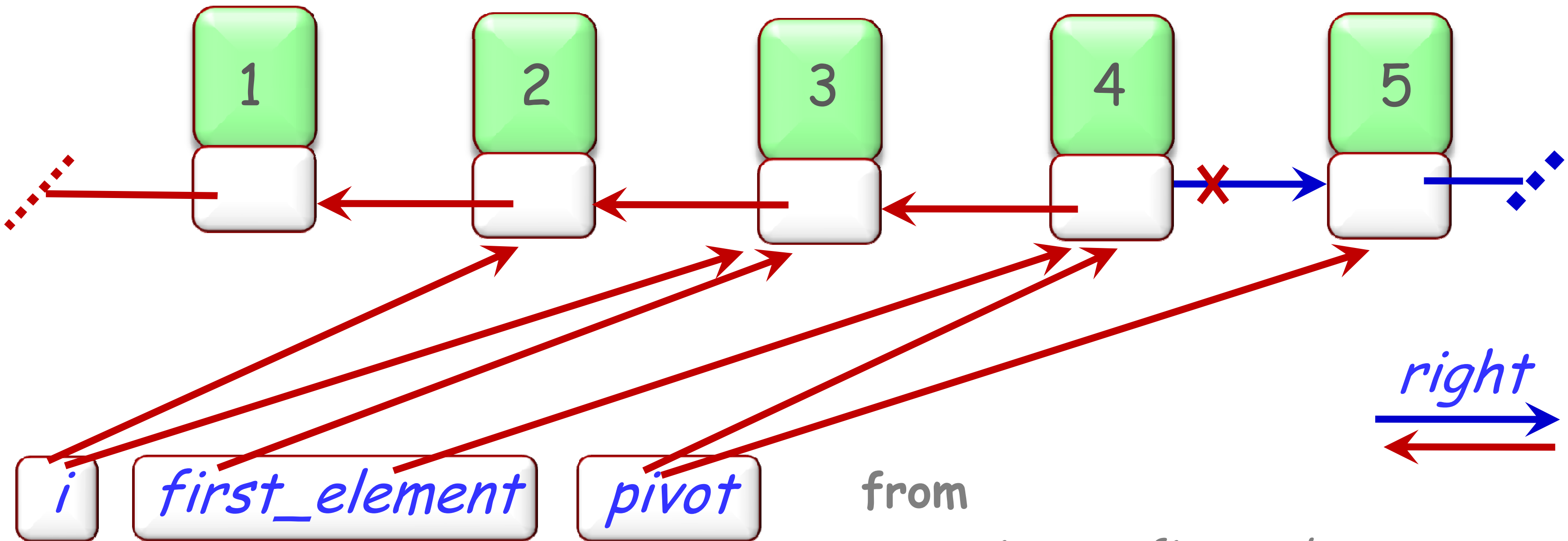

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



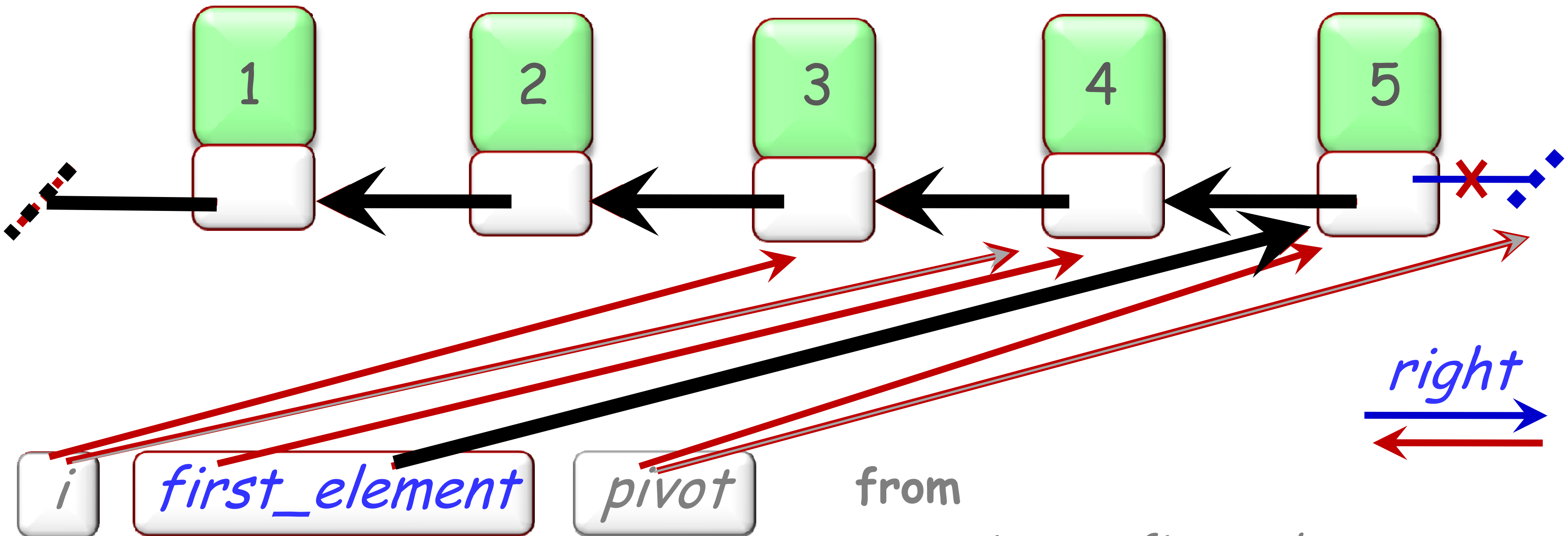

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



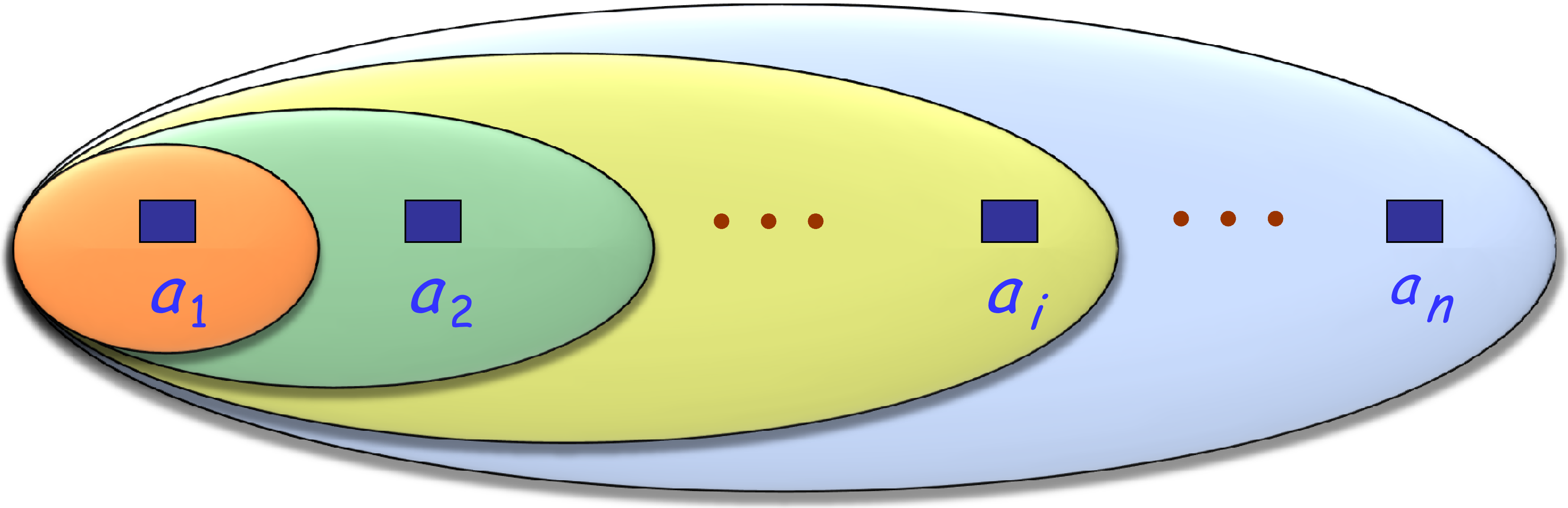

```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list




```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Loop as approximation strategy



Result = a_1 = Max (a_1 .. a_1)

Result = Max (a_1 .. a_2)

Loop body:

```
i := i + 1
Result := max (Result , a [i ])
```

Result = Max (a_1 .. a_i)

The loop invariant

Result = Max (a_1 .. a_n)

Loops as problem-solving strategy

A loop invariant is a property that:

- Is easy to **establish initially** (even to cover a trivial part of the data)
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**!

		0	1 B	2 E	3 A	4 T	5 L	6 E	7 S
	0	0	1	2	3	4	5	6	7
B	1	1	0	1					
E	2	2	1	0	1	2	3	4	5
E	3	3	2	1	1	2	3	3	4
T	4	4	3	2	2	1	2		
H	5								

Invariant: each *a* [*i*, *j*] is distance from *source* [1 .. *i*] to *target* [1 .. *j*]

K R D I

K Keep

I Insert

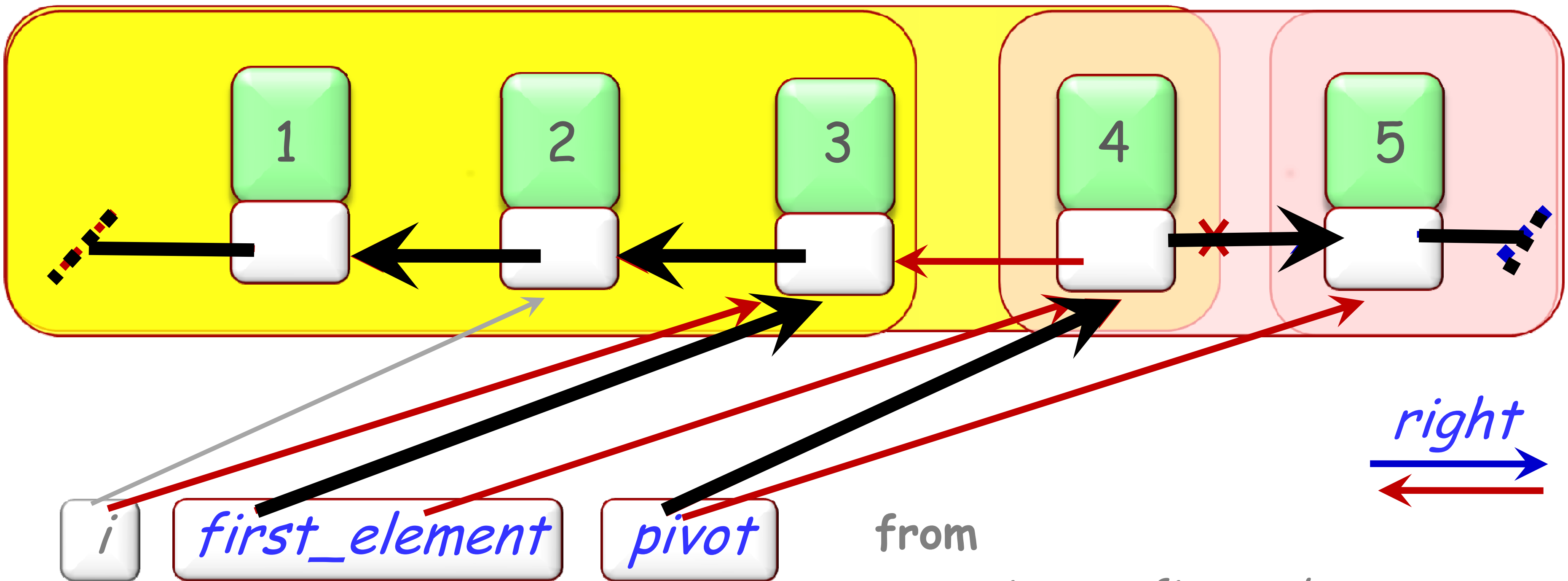
D Delete

R Replace

Levenshtein loop

```
from i := 1 until i > source.count loop
    from j := 1 until j > target.count invariant
        -- For all p : 1 .. i, q : 1 .. j–1, we can turn source [1 .. p ]
        -- into target [1 .. q ] in dist [p, q ] operations
    loop
        if source [i ] = target [ j ] then
            new := dist [ i -1, j -1]
    else
            deletion := dist [i -1, j ]
            insertion := dist [i , j - 1]
            substitution := dist [i - 1, j - 1]
            new := deletion.min (insertion.min (substitution)) + 1
        end
        dist [i, j ] := new
        j := j + 1
    end
    i := i + 1
end
Result := dist (source.count, target.count)
```

Reversing a list



Invariant: from *first_element* following right, initial items in inverse order; from *pivot*, rest of items in original order

```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Routines (1)

For:

f (x: T) **do** Body **end**

$$\frac{\{P\}\ \text{Body}\ \{Q\}}{\boxed{\{P\ [a/x]\}\ \ f\ (a)\ \ \{Q\ [a/x]\}}}$$

Routines (2)

For:

f (x: T) **do** Body **end**

$$\frac{(\forall a \mid \boxed{\{P\,[a/x]\}\ \ f\,(a)\ \ \{Q\,[a/x]\}})\ \ \textbf{implies}\ \ \{P\}\ Body\ \{Q\}}{\boxed{\{P\,[a/x]\}\ \ f\,(a)\ \ \{Q\,[a/x]\}}}$$

Hoare (1971)

The solution to the infinite regress is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself. Thus we are permitted to prove that the procedure body possesses a property, on the assumption that every recursive call possesses that property, and then to assert categorically that every call, recursive or otherwise, has that property. This assumption of what we want to prove before embarking on the proof explains well the aura of magic which attends a programmer's first introduction to recursive programming.

Procedures and Parameters: An Axiomatic Approach, in E. Engeler (ed.), *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, pp. 102-16 (1971).

Functions

The preceding rule applies to procedures (routines with no results)

Extension to functions?

Soundness and completeness

How do we know that an axiomatic semantics (or *logic*) is "right"?

- Sound: every deduced property holds of all corresponding program executions

- Complete: every property that holds of all program executions can be proved by the logic
  (Undecidable!)

A model

To examine soundness and completeness we need a model: a mathematical description of program executions

Basic model (programs without input):

$$\mathfrak{M}: \text{Instruction} \rightarrow (\text{State} \nrightarrow \text{State})$$

Partial functions

$$\text{State} \triangleq \text{Variable} \rightarrow \text{Value}$$

Also needed:

$$\mathcal{E}: \text{Expression} \rightarrow (\text{State} \nrightarrow \text{Value})$$

Example interpretations

$\mathfrak{M}$ [x := e] (s) = s ⊎ [x, $\mathcal{E}$ [e] (s)]

"Overriding union"

$\mathfrak{M}$ [i1 ; i2] (s) =

$\mathfrak{M}$ [**if** c **then** i1 **else** i2 **end**] (s) =

Notation

For an axiomatic theory *A*, a model *M* and a property p:

$M \models p$

means that p can be proved from *M*

$A \vdash p$

means that p can be proved from *A*