Introduction to Programming

Bertrand Meyer

Lecture 9:
Control structures (end)

Last revised 9 December 2005

Announcement: classroom exercise

First classroom exercise in next week's Monday (5 December 2005) exercise session.
There is no assignment this week.

On the menu for today

Loops and their invariants
See what it takes to ensure that a loop terminates
Look at the general problem of loop termination
Examine lower-level control structures: “Goto” and flowcharts; see rationale for the “control structures of Structured Programming”
Study routines
Prove the undecidability of the Halting Problem
Loop

from
  Initialization  -- Compound

until
  Exit_condition  -- Boolean_expression

loop
  Body          -- Compound
end

Loop, full form

from
  Initialization  -- Compound

invariant
  Invariant_expression  -- Boolean_expression

variant
  Variant_expression  -- Integer_expression

until
  Exit_condition  -- Boolean_expression

loop
  Body          -- Compound
end

Looping over stations of a line

from
  fancy_line.start
until
  fancy_line.after

loop
  "Do something with fancy_line.item"
    fancy_line.forth
end
### Operations on a list

- **before**
- **item**
- **count**

**Commands**
**Queries**
(boolean)

- **start**
- **index**
- **forth**

(The cursor)

![Diagram of list operations](image)

---

### Looping over stations of a line

```plaintext
from 
fancy_line.start
until 
fancy_line.after
loop
  -- "Do something with fancy_line.item"

  fancy_line.forth
end
```

---

### Operations on a list (Repetition)

- **before**
- **item**
- **count**

**Commands**
**Queries**
(boolean)

- **start**
- **index**
- **forth**

(The cursor)

![Diagram of list operations](image)
Another loop syntax (from forum)

Displaying station names

```from
fancy_line.start
until
fancy_line.after
loop
   -- Display name of next station:
   Console.show(fancy_line.item)
   fancy_line.forth
end```

Computing the "maximum" of station names

```from
   fancy_line.start : Result := ""
until
   fancy_line.after
loop
   Result := greater(Result, fancy_line.item.name)
   fancy_line.forth
end```
Assignment

Result := "XYZ"

-- Change the value of Result to "XYZ"

Computing the "maximum" of station names

from
  fancy_line.start; Result := ""
until
  fancy_line.after
loop
  Result := greater (Result, fancy_line.item.name)
fancy_line.forth
end

In a function

highest_name: STRING is
  -- Alphabetically greatest station name of line
  do
    from
      fancy_line.start; Result := ""
    until
      fancy_line.after
    loop
      Result := greater (Result, fancy_line.item.name)
      fancy_line.forth
    end
  end
Postcondition?

```
highest_name: STRING is
  -- Alphabetically greatest station name of line
do
  from
  until fancy_line.end
  loop
    result := greater (result, fancy_line.item.name)
  end
end
```

Loop as approximation strategy

```
Result = name1
Result = Max (name1, name2)
Result = Max (name1, name2, ..., name)
```

The loop invariant

```
from
  fancy_line.start; Result := ""
invariant
  fancy.index := 1
  fancy.index := fancy.count + 1
  -- Result is the alphabetically highest of the
  -- names of previous stations
until
  fancy_line.end
loop
  result := greater (result, fancy_line.item.name)
  fancy.line.forth
end
```
Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause)
- when executed with the exit condition (until clause) not satisfied

The loop invariant

```plaintext
from
  fancy_line.start ; Result == ""
invariant
  fancy.index := 1
  fancy.index <= fancy.count + 1
  // Result is the alphabetically highest of the
  // names of previous stations
  until
    fancy_line.after
  loop
    Result := greater(Result, fancy_line.item.name)
    fancy_line.forth
  end
```

The loop invariant (better)

```
from
  fancy_line.start ; Result == ""
invariant
  index := 1
  index <= count + 1
  // If there are any previous stations,
  // Result is the alphabetically highest of their names
  until
    fancy_line.after
  loop
    Result := greater(Result, fancy_line.item.name)
    fancy_line.forth
  end
```
### The halting problem in practice

Some programs do not terminate in certain cases...

That's a bug!

Yours had better terminate in all cases

Use variants

---

### In a function

```plaintext
highest_name: STRING is
   -- Alphabetically greatest station name of line
   do
      from fancy_line.start : Result := 
      until fancy_line.after
      loop
         Result := greater(Result, fancy_line.item.name)
         fancy_line.forth
      end
   end
```

---

### Postcondition?

```plaintext
highest_name: STRING is
   -- Alphabetically greatest station name of line
   do
      from fancy_line.start : Result := 
      until fancy_line.after
      loop
         Result := greater(Result, fancy_line.item.name)
      fancy_line.forth
      ensure
         Result /= Void and then not Result.empty
      end
```
Loop as approximation strategy

\[ \text{Result} = \text{name}_1 \]
\[ \text{Result} = \text{Max}(\text{name}_1, \text{name}_2) \]
\[ \text{Result} = \text{Max}(\text{name}_1, \text{name}_2, \ldots, \text{name}_n) \]

The loop invariant

\begin{verbatim}
from
  fancy_line.start, Result := ""
invariant
  Fancy_index >= 1
  Fancy_index <= Fancy.count + 1
  -- Result is the alphabetically highest of the
  -- names of previous stations
until
  Fancy Line after
loop
  Result := greater(Result, Fancy_line.item.name)
  Fancy_line.forth
end
\end{verbatim}

Loop invariant

(Do not confuse with class invariant)

Property that is:

- Satisfied after initialization (from clause)
- Preserved by every loop iteration (loop clause) when executed with the exit condition (until clause) not satisfied
The loop invariant

```plaintext
from fancy_line.start; Result :=

invariant
fancy.index := 1
fancy.index <= fancy.count + 1
-- Result is highest of previous station names
until
fancy_line.after
loop
Result := greater(Result, fancy_line.item.name)

fancy_line.forth
end
```

The loop invariant (better)

```plaintext
from fancy_line.start; Result :=

invariant
index := 1
index <= count + 1
-- If there are any previous stations,
-- Result is the alphabetically highest of their names
until
fancy_line.after
loop
Result := greater(Result, fancy_line.item.name)

fancy_line.forth
end
```

The effect of the loop

```plaintext
from fancy_line.start; Result :=

invariant
index := 1
index <= count + 1
-- Result is highest of previous station names
until
fancy_line.after
loop
Result := greater(Result, fancy_line.item.name)

fancy_line.forth
end
```

Invariant satisfied after initialization

Exit condition satisfied at end

Invariant satisfied after each iteration

At end: invariant and exit condition
- All stations visited (fancy_line.after)
- Result is highest of their names
Intermezzo: Levenshtein distance

- Also called "Edit distance"
- Purpose: to compute the smallest set of basic operations
  - Insertion
  - Deletion
  - Replacement

  that will turn one string into another

Levenshtein distance

- "Michael Jackson" to "Mendelssohn"

<table>
<thead>
<tr>
<th>Operation</th>
<th>S</th>
<th>D</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Levenshtein distance algorithm

\[
distance\text{\ (source, target: STRING, INTEGER)}
\]

\[
\text{\ -- Minimum number of operations to turn source into target}
\]

\[
\text{\ local}
\]

\[
dist\text{\ ARRAY_2[INTEGER]}
\]

\[
\text{\ i, j, new, deletion, insertion, substitution: INTEGER}
\]

\[
do\text{\ create dist, make (source, count, target, count)}
\]

\[
\text{\ from } i := 0 \text{\ until } i \text{\ > source, count loop}
\]

\[
dist[i, 0] := i; \ / i \neq 1
\]

\[
end\text{\ from } j := 0 \text{\ until } j \text{\ > target, count loop}
\]

\[
dist[0, j] := j; \ / j \neq 1
\]

\[
end\text{\ -- (Continued) }
\]
Levenshtein, continued

from i := 1 until j > source.count loop
  from j := 1 until j > target.count invariant
  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
    end j := j + 1
  end
  Result := dist(source, count, target, count)

Loop semantics rule

The effect of a loop is the combination of:

- Its invariant
- Its exit condition

Quiz: what's the invariant?

xxx (a is INTEGER, b is INTEGER)
  return
  a = 0; b = 0
  local m, n: INTEGER
  do from
    invariant m = a; n = b
    variant
    loop
      if m = n then
        m = m - n
      else
        n = n - m
      end
    end
  end
  Result = m
How do we know a loop terminates?

```
from
fancy_line.start; Result :="

invariant
index := 1
index := count + 1
-- If there are any previous stations,
-- Result is the alphabetically highest of their names
until
fancy_line.after
loop
Result := greater(Result, fancy_line.item.name)
```

Loop variant

Integer expression that must:

- Be non-negative when after initialization (from)

- Decrease (i.e. by at least one), while remaining non-negative, for every iteration of the body (loop) executed with exit condition not satisfied

The variant for our loop

```
from
fancy_line.start; Result :="

invariant
index := 1
index := count + 1
-- If there are any previous stations,
-- Result is the alphabetically highest of their names
variant
fancy_line.count - fancy_line.index + 1
until
fancy_line.after
loop
Result := greater(Result, fancy_line.item.name)
```

end
The general termination problem

Can EiffelStudio find out if your program will terminate?

No, it can’t 😞

No other program, for any other realistic programming language, can! 😞 😞 😞

The halting problem and undecidability

("Entscheidungsproblem", Alan Turing, 1936.)

It is not possible to devise an effective procedure that will find out if an arbitrary program will terminate on arbitrary input

(or, for that matter, if an arbitrary program with no input will terminate)

The halting problem in Eiffel

Assume we have a routine

```eiffel
terminates(my_program: STRING): BOOLEAN
  -- Does my_program terminate?
  do
    ... Your algorithm here ...
  end
```


The halting problem in practice

Some programs do not terminate in certain cases...
That's a bug!
Yours had better terminate in all cases
Use variants

Control structures at the machine level

Unconditional branch:
BR label

Conditional branch, for example:
BEQ loc_a loc_b label

The equivalent of if-then-else

if a = b then Compound_1 else Compound_2 end

BEQ loc_a loc_b 111
101 ... Code for Compound_2 ...
BR 125
111 ... Code for Compound_1 ...
125 ... Code for continuation of program ...
Flowcharts

Test condition goto else_part

Compound_1

else_part: Compound_2

continue: Continuation of program...

"Goto considered harmful"

Dijkstra, 1968
Arbitrary Goto instructions lead to messy, hard to maintain programs ("spaghetti code")
The Goto today

Almost universally decried
Still exists in some programming languages
Also hides under various disguises, e.g. break

```
loop
  ...
  if c then break end
  ...
end
```

One-entry, one-exit

(Compound) (Loop) (Conditional)

Quiz: what’s the invariant?

```plaintext
*** (a, b INTEGER, c INTEGER) ...
requires
  a > 0; b > 0
local
  m, n: INTEGER
do
  from
    m = a; n = b
  invariant ...
  variant ...
  until
  m = n
  loop
    if m = n then
      end
    else
      n = n - m
    end
  end
  Result = m
```

```
Quiz: what’s the invariant?

```plaintext
module gcd(a, b: INTEGER) : INTEGER
  -- Greatest common divisor of a and b
  require
    a > 0; b > 0
  local
    m := INTEGER
  do
    from
    invariant ...
    variant ...
    until 
    loop m := n
      if m > n then
        m := m - n
      else
        n := n - m
      end
    end
    end
  Result = m
```

Intermezzo: Levenshtein distance

Also called “Edit distance”

Purpose: to compute the smallest set of basic operations

- Insertion
- Deletion
- Replacement

that will turn one string into another

Levenshtein distance

```
"Michael Jackson" to "Mendelssohn"

```

```
Operation          S     D     S     S   S   D   D   D   D   I
Distance           0 1 2 3 4     5 6 7 8 9 10
```

### Levenshtein distance algorithm

```plaintext
distance (source, target: STRING): INTEGER
  -- Minimum number of operations to turn source into target
local
  dist ARRAY_2 [INTEGER]
  i, j, new, deletion, insertion, substitution: 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, continued

```plaintext
from i := 1 until i > source, count loop
  from j := 1 until j > target, count invariant
    loop
      -- For all p ≤ i, q ≤ j, we can turn source [1..p] into target [1..q] in dist [p, q] operations
      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
  end

Result := dist (source, count, target, count)
```

### Levenshtein distance algorithm

```plaintext
distance (source, target: STRING): INTEGER
  -- Minimum number of operations to turn source into target
local
  dist ARRAY_2 [INTEGER]
  i, j, new, deletion, insertion, substitution: 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, continued

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

End of lecture 9