Introduction to Programming
Bertrand Meyer

Lecture 9:
Control structures (end)

Last revised 5 December 2005

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

invariant
  Invariant_expression   -- Boolean_expression
variant
  Variant_expression    -- Integer_expression

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
- index
- count

1. start
2. back
3. forth

 Commands
 Queries
 (boolean)

**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
- index
- count

1. start
2. back
3. forth

 (The cursor)

**Another loop syntax (from forum)**

```
for (count = 0; count < 500; count++)
    printf("we will throw paper airplanes in class.");
```

**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**

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

**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
  end
require
  Result /= Void and then not Result.empty
end
```

**Loop as approximation strategy**

```
Result = name_1
Result = Max (name_1, name_2)
Result = Max (name_1, name_2, ..., name)
Result = Max (name_1, name_2, ..., name)
```

**The loop invariant**

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

```
invariant
fancy_index := 1
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
```

---

3
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

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

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

fancy_line.start ; Result :: ""
until

fancy_line.after
loop

Result := greater(Result, fancy_line.item.name)

fancy_line.forth
end

ensure

Result /= Void and then not Result.empty

end
Loop as approximation strategy

Result = name_1
Result = Max (name_1, name_2)
Result = Max (name_1, name_2, ..., name_m)

The loop invariant

\[
\text{from } \quad \text{fancy\_line\_start }; \quad \text{Result := } \text{"\text{\textbf{\textit{}}}"} \\
\text{invariant} \\
\quad \text{fancy\_index} := 1 \\
\quad \text{fancy\_index} := \text{fancy\_count} + 1 \\
\quad \text{-- Result is the alphabetically highest of the} \\
\quad \text{-- names of previous stations} \\
\text{until} \\
\quad \text{fancy\_line\_after} \\
\text{loop} \\
\quad \text{Result := greater(\text{Result}, \text{fancy\_line\_item\_name})} \\
\quad \text{fancy\_line\_forth} \\
\text{end}
\]

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 (better)

\[
\text{from} \\
\quad \text{fancy\_line\_start }; \quad \text{Result := } \text{"\text{\textbf{\textit{}}}"} \\
\text{invariant} \\
\quad \text{index} := 1 \\
\quad \text{index} := \text{count} + 1 \\
\quad \text{-- If there are any previous stations,} \\
\quad \text{-- Result is the alphabetically highest of their names} \\
\text{until} \\
\quad \text{fancy\_line\_after} \\
\text{loop} \\
\quad \text{Result := greater(\text{Result}, \text{fancy\_line\_item\_name})} \\
\quad \text{fancy\_line\_forth} \\
\text{end}
\]

The effect of the loop

\[
\text{from} \\
\quad \text{fancy\_line\_start }; \quad \text{Result := } \text{"\text{\textbf{\textit{}}}"} \\
\text{invariant} \\
\quad \text{index} := 1 \\
\quad \text{index} := \text{count} + 1 \\
\quad \text{-- Result is highest of previous station names} \\
\text{until} \\
\quad \text{fancy\_line\_after} \\
\text{loop} \\
\quad \text{Result := greater(\text{Result}, \text{fancy\_line\_item\_name})} \\
\quad \text{fancy\_line\_forth} \\
\text{end}
\]
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 algorithm

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

begin
create dist, make (source, count, target, count)
from i = 0 until i > source, count loop
   dist[0, i] := i ; i := i + 1
end

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

Levenshtein, continued

from i := 1 until i > 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 := min (deletion, min (insertion, substitution)) + 1
      end
      dist[i, j] := new
      i := i + 1
   end
   j := j + 1

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?

ax(x, b: INTEGER) : INTEGER
require
b > 0 ; b > 0
local
m, n: INTEGER

do
   from
   invariant
   m = a ; n = b
   variant ??????
   until
   m = n
   loop
      if m > n then
         m := m - n
      else
         n := n - m
      end
   end
   Result := m
end


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)
end

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)
fancy_line.forth
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

`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

In programming languages: the Goto

test condition goto else_part
Compound_1
continue
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

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

**One-entry, one-exit**

- **Compound**
- **Loop**
- **Conditional**

---

**Quiz: what’s the invariant?**

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

**Quiz: what’s the invariant?**

```plaintext
while (a, b: INTEGER, INTEGER) do
  if a = b then
    break
  end
  local
    m, n: INTEGER
  do
    from
      m := a; n := b
    invariant -- ?????
    variant ??
  until
    m > n
  loop
    if m > n then
      m := m - n
    else
      n := n - m
  end
  Result := m
end
```

---

**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"

  ![Levenshtein distance example](image)

  Distance: 10
Levenshtein distance algorithm

```plaintext
distance(source: 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.length loop
      dist[i, 0] := i; i := i + 1
    end
    from j := 0 until j > target.length loop
      dist[0, j] := j; j := j + 1
    end
  end
  -- (Continued)
```

Levenshtein, continued

```plaintext
from i := 1 until i > source.length loop
  from j := 1 until j > target.length loop
    dist[i, j] := new
    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 := dist[i-1, j-1] + 1
    end
    dist[i, j] := new
    i := i + 1
  end
  j := j + 1
end
Result := dist(source.length, target.length)
```

Levenshtein, continued

```plaintext
from i := 1 until i > source.length loop
  from j := 1 until j > target.length loop
    dist[i, j] := new
    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 := dist[i-1, j-1] + 1
    end
    dist[i, j] := new
    i := i + 1
  end
  j := j + 1
end
Result := dist(source.length, target.length)
```

End of lecture 9