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"

Prove the undecidability of the Halting Problem

A loop example: Levenshtein distance

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 ---
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

def distance(source, target: STRING): INTEGER
    -- Minimum number of operations to turn source into target
    local dist: ARRAY_2(INTEGER)  -- Indexed from zero
    i, j, new, 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)
    from i := 1 until i > source.count loop
        from j := 1 until j > target.count loop invariant
            if source[i] = target[j] then
                new := dist[i-1, j-1]
            else
                del := dist[i-1, j]; ins := dist[i, j-1]; subst := dist[i-1, j-1]
                new := minimum (del, ins, subst) + 1
            end
            dist[i, j] := new
            i := i + 1
        end
        Result := dist(source.count, target.count)
    end
end

Levenshtein, continued
The loop invariant

from fancy.start; Result := ''

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

from fancy.start; Result := ''

invariant
  fancy.index := 1
  fancy.index <= fancy.count + 1
  -- Result is the alphabetically highest of the
  -- names of previous stations
  until
    fancy.after
  loop
    Result := greater(Result, fancy.item.name)
    fancy.forth
  end
The loop invariant (better)

```
from
  fancy.start ; Result := ""
invariant
  index := 1
  index <= count + 1
  \-- If there are any previous stations, \-- Result is the alphabetically highest of their names
until
  fancy.after
loop
  Result := greater (Result, fancy.item.name)
  fancy.forth
end
```

Loop as approximation strategy

```
Result = name_1

Result = Max (name_1, name_2)

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

Result = Max (name_1, name_2, ..., name_i, ..., name_n)
```

In a function

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

Postcondition?

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

ensure Result /= Void and then not Result.empty
end

The effect of the loop

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

Quiz: what’s the invariant?

xxx (a, b: INTEGER; INTEGER is
  require a > 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
  end
  Result := m
end
Loop semantics rule

The effect of a loop is the combination of:

- Its invariant
- Its exit condition

How do we know a loop terminates?

```plaintext
from
  fancy.start; Result := 1
invariant
  index := 1
  index <= count + 1
  -- If there are any previous stations,
  -- Result is the alphabetically highest of their names
until
  fancy.after
loop
  Result := greater(Result, fancy.item.name)
  fancy.forth
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.start; Result := ""

invariant
  index >= 1
  index <= count + 1
  -- If there are any previous stations,
  -- Result is the alphabetically highest of their names

variant
  fancy.count - fancy.index + 1

until
  fancy.after

loop
  Result := greater(Result, fancy.item.name)

  fancy.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 😞 😞 😞

Paradoxes in logic

The Russell paradox

- Some sets are members of themselves, e.g. the set of all infinite sets is infinite
- Others are not members of all sets, e.g. the set of all finite sets is not finite
- Consider the set of all sets that are not members of themselves

The barber paradox (Russell)

- In Zurich there is a hairdresser that does the hair of all the inhabitants who do not do their own hair
- Who does her hair?
The Grelling paradox

An adjective in English is:
- "Autological" if it applies to itself (e.g. "polysyllabic")
- "Heterological" otherwise
- What is the status of "Heterological"?

Another form:
The first proposition appearing in green on this page is false

The liar’s paradox

(Very old!)
- Epaminondas says all Cretans are liars
- Epaminondas is a Cretan

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
```

Then we can write

```eiffel
terminates(file_name: STRING): BOOLEAN
-- Does program in file file_name terminate?
do
... Your algorithm here ...
end
```

Then...

Root procedure of the system:

```eiffel
what_do_you_think
-- Terminate only if not
do
from until not terminates("/usr/home/myfile")
loop
end
```

Then copy the program text to /usr/home/myfile
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

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”

```
  M I C H A E L   JACKSON
  ----------------------
       E N D S   H
```

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

```pascal
distance (source, target: STRING): INTEGER
  -- Minimum number of operations to turn source into target
  local
dist: ARRAY_2 [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

```pascal
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
      del := dist[i-1, j]; ins := dist[i, j-1]; subst := dist[i-1, j-1]
      new := min (del, ins, subst) + 1
    end
    dist[i, j] := new
    i := i + 1
  end
end
result := dist (source, count, target, count)
```

What we have seen

Loop correctness: invariant, variant
Termination
Concept of undecidability
The halting problem and its undecidability

An example algorithm using non-trivial loops, which looks mysterious until you have seen the invariant:
Levenshtein distance