Lecture 4:
More Algorithmic Constructs

Aims:

- To investigate reductions from one construct to another; and
- To look at further aspects of the DEC CAFF language:
  - nested control structures;
  - dangling-else constructions; and
  - procedures.

4.0.1 Loop Construct Reductions

- You may be worried that your programming language does not have all four unbounded iteration constructs. For example, Java has just two of them. But you needn't worry.

  Given even just one of the four constructs, you can translate an algorithm that uses the other three kinds of loop into an algorithm that uses just the one kind of loop that you have plus sequencing and conditionals.

- Here we assume we have pre-test, exit-when-**true** constructs (**while** loops). We show how to translate the other three.

```
until B
{   C
}
```
becomes
```
while ¬B
{   C
}
```

```
do
{   C
}
until B
```
becomes
```
C;
while ¬B
{   C
}
```



```
do
{   C
}
while B
```
becomes
```
C;
while B
{   C
}
```

- Similarly, in the unlikely event that your programming language doesn't offer any bounded loop constructions (such as **for** -loops), again you needn't worry. They can be reduced to unbounded iteration.
- Here we look at how to reduce a simple **for** loop to a **while** loop.

```
for V := E1 upto E2
   C;
```
becomes
```
V := E1;
HIGHEST := E2;
while V ≤ HIGHEST
{   C;
    V := V + 1;
}
```

. . . provided C does not alter the contents of V in any way.

4.1 Nested Control Structures

- We have already noted that the sequence, conditional and iteration constructs are built out of other commands (e.g. the arms of conditionals and the body of loops are commands). This gives the possibility of nesting one control structure inside another.

  This makes reasoning about algorithms and programs harder and requires more care on our part.

  Let's start by looking at the nesting of **if** commands within other **if** commands.

4.1.1 Nested if Commands

- Here is a problem that illustrates some of the issues.

Problem 4.1.
Parameters: *An integer, ms, denoting marital status (0 for single, 1 for married); and a positive integer s, denoting salary.*
Returns: *The income tax this person will pay.*

Here are some fictitious and highly simplified rules for determining your tax.

If you are single

If your salary is over	But not over	The tax rate is
€0	€20,000	15%
€20,000	€50,000	28%
€50,000		45%

If you are married

If your salary is over	But not over	The tax rate is
€0	€35,000	15%
€35,000	€85,000	28%
€85,000		45%

- Here's a fragment of pseudocode to compute your tax, t.

```
if ms = 0
{   if s ≤ 20000
    {   t := s × 0.15
    }
    else if s ≤ 50000
    {   t := s × 0.28
    }
    else
    {   t := s × 0.45
    }
}
else
{   ...
}
```

We have a conditional that tests the marital status. Then in both the **if** and **else** branches of that **if** command we have another conditional. These conditional themselves contain further conditionals nested in their **else** branches.

Class Exercise

- This looks the same, but is wrong. Why?

```
if ms = 0
{   if s ≤ 50000
    {   t := s × 0.28
    }
    else if s ≤ 20000
    {   t := s × 0.15
    }
    else
    {   t := s × 0.45
    }
}
else
{   ...
}
```

- You really only have the freedom to change the order of tests if they are mutually exclusive.

```
if s ≥ 0 ∧ s ≤ 20000
{   t := s × 0.15
}
else if s > 20000 ∧ s ≤ 50000
{   t := s × 0.28
}
else if s > 50000
{   t := s × 0.45
}
```

...is the same as...

```
if s > 20000 ∧ s ≤ 50000
{   t := s × 0.28
}
else if s ≥ 0 ∧ s ≤ 20000
{   t := s × 0.15
}
else if s > 50000
{   t := s × 0.45
}
```

...and even the same as...

```
if s > 20000 ∧ s ≤ 50000
{    t := s × 0.28
}
if s ≥ 0 ∧ s ≤ 20000
{    t := s × 0.15
}
if s > 50000
{    t := s × 0.45
}
```

...which, if you look at it again carefully, you wlll see has no nesting. It's just a sequence of separate **if** commands. It too can be re-ordered as much as you like — again because the conditions are strong enough to give mututal exclusion.

4.1.2 The Dangling-else Problem

4.1.2.1 Class Exercise

- What is printed by the following DEECAFF algorithm?

```
x := 1;
y := 1;
if x = 2
   if y = 2
      display"Both x and y are 2";
else
   display"x was not 2";
```

- Confusion can arise when a command mixes one-armed and two-armed conditionals. It can be unclear to which **if** an **else** belongs. This is called the *dangling-**else** problem*.
- This is a source of numerous errors when programming in, e.g., Java. Students often assume that their indentation will make things clear to the computer. But the computer takes no note of indentation. It will have some simple rule for deciding to which **if** each **else** should be associated. The most usual rule is: an **else** always belongs to the closest **if** (unless braces are used to overrule this convention).

- A properly laid out version of the previous algorithm makes it clear why there was no output:

```
x := 1;
y := 1;
if x = 2
   if y = 2
      display"Both x and y are 2";
   else
      display"x was not 2";
```

- The **else** belongs to the closest **if** , irrespective of the programmer's indentation.
  So the two versions are the same. The only difference is that first one had misleading indentation.
- The solution is to use braces to properly start and end the parts of the different branches.
  For example, this is what the programmer actually intended:

```
x := 1;
y := 1;
if x = 2
{    if y = 2
        display"Both x and y are 2";
}
else
   display"x was not 2";
```

- When you have mixed one-armed and two-armed conditionals, braces can disambiguate for you:

```
if B1
{    if B2
        C1;
     else
        C2;
}
```

```
if B1
{    if B2
        C1;
}
else
   C2;
```

- As we have been discussing, only the right-hand one *needs* the braces. But it may make you think harder and get things right more often if you include braces in both cases.
- This is one of the reasons why I use braces very liberally. As you've seen, I even put them round single commands. It is one of my defences against causing dangle-else problems.

4.1.3 Nested Loops

- Equally common is the nested loop.
- Here's an example that does matrix multiplication. You don't need to understand matrix multipication: this is just an example of nested loops.

  However, for those of you who are interested, here's a quick recap. Suppose a and b are two matrices such that the number of columns of a is equal to the number of rows of b. Say a is a $m \times p$ matrix and b is a $p \times n$ matrix. Then the result of multiplying them, c, is a $m \times n$ matrix. In c, the ij-entry is obtained by multiplying the elements of the ith row of a by the corresponding elements of the jth column of b and then adding.

  For example,

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix} = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$

  The top left-hand corner of the answer is 21 because $1 \times 5 + 2 \times 8 = 21$.

- Here's the algorithm:

Algorithm: MATRIXMULT(a, b)
Parameters: Two two-dimensional arrays of integers, $a[1 \ldots m][1 \ldots p]$ and $b[1 \ldots p][1 \ldots n]$.
Returns: The product of a and b.

```
{   create array c[1...m][1...n] with 0 in each cell
    for i := 1 upto m
    {   for j := 1 upto n
        {   for k := 1 upto p
            {   c[i][j] := c[i][j] + a[i][k] × b[k][j];
            }
        }
    }
}
return c;
```

  How many times does the assignment to $c[i][j]$ get executed?

- Although the example shows nested **for** loops, obviously any loop construct can be nested within any other loop construct.

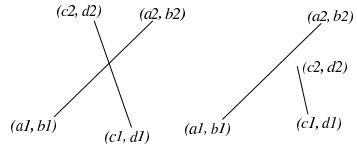
4.2 Procedures

- In DECAFF, we allow ourselves modules of code that we will call *procedures*. In programming languages, these are variously referred to as procedures, functions, modules, subroutines and methods. They allow us to break up lengthy algorithms into more manageable chunks. And they allow us to avoid duplication within an algorithm.

- Here's an example.

Problem 4.2.
Parameters: *Two line segments. One line segment has end-points (a_1, b_1) and (a_2, b_2). The other has end-points (c_1, d_1) and (c_2, d_2).*
Returns: *YES if the two line segments intersect; otherwise NO.*

- In the left-hand example, the answer is YES; in the right-hand example, the answer is NO:



- The mathematics of how we determine this don't matter. All we're really interested in here is illustrating the idea of a procedure.

Algorithm: SEGMENTINTERSECTION$(a_1, b_1, a_2, b_2, c_1, d_1, c_2, d_2)$

```
{   if DETERMINANT(a1, b1, a2, b2, c1, d1) =
                   DETERMINANT(a1, b1, a2, b2, c2, d2)   ∧
       DETERMINANT(c1, d1, c2, d2, a1, b1) =
                   DETERMINANT(c1, d1, c2, d2, a2, b2)
    {   return YES
    }
    else
    {   return NO
    }
}

procedure DETERMINANT(x1, y1, x2, y2, x3, y3)

{   return x1 × y2 − x2 × y1 + x3 × y1 − x1 × y3+
                x2 × y3 − x3 × y2
}
```

  (In fact, from a mathematical point of view, the algorithm isn't the complete story. Some extra tests are needed.)

- We're not going to use procedures very much in this module. Our algorithms are going to be short and sweet, so we won't have much need for them. Therefore, I don't want to get bogged down in matters of detail.

- But those of you who have an interest in programming languages might like to think about the following.

  Here is a very simple (very inefficient) sorting algorithm. It uses a procedure called *swap*. But if you coded this up in most programming languages, including Java, *swap* would not actually do what you want it to do. The question is: why? To answer the question, you need to find a book and read about parameter passing. The key phrases to look for are *pass-by-value* and *pass-by-reference*.

Algorithm: BUBBLESORT($a, lower, upper$)
Parameters: An array of integers, $a[lower \ldots upper]$,
$0 < lower \leq upper$.
Returns: The array sorted into ascending order.

```
{   for i := lower upto upper − 1
    {   for j := lower upto upper − i
        {   if a[j] > a[j + 1]
            {   SWAP(a[i], a[j])
            }
        }
    }
}
procedure SWAP(x, y)

 {   temp := x;
     x := y;
     y := temp
 }
```

- That concludes our overview of DECAFF.

  You may realise that there are lots of features that real programming languages have that DECAFF doesn't. In particular, if you are comparing with Java, DECAFF doesn't have any of the features that make Java object-oriented (e.g. class definitions, constructors, instance variables, inheritance, etc.).

  If DECAFF were a real programming language, these omissions might be very serious. The missing features are ones that enable us to structure large programs, to make them more manageable. On large software engineering projects, such features are essential.

  But we don't need them for our exploration of the theory of computation. We're only dealing with short algorithms and programs. Besides, while these extra features may bring convenience, they bring no additional computational power.

Acknowledgements

The tax example is based on one in [Hor98]. If you want to find out more about line segment intersection (e.g. to find out what is missing from my algorithm), look at Section 15.1.3 of [GT98], which is where I took the idea from.

References

[GT98] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, 1998.

[Hor98] C. Horstmann. *Computing Concepts with Java Essentials*. Wiley, 1998.