Lecture 3:
Algorithmic Constructs

Aims:

- To agree on a basic language that we will use for the presentation of algorithms throughout this module:
  - data types and their operations;
  - variables and assignment; and
  - control structures (sequence, conditional and iteration).

3.1. Algorithmic Pseudocode

We need a language in which we can write our algorithms. The language we will develop will resemble some parts of Java. Why not use Java itself? The idea here is communication between us, humans, rather than communication with the machine. Using Java, or any other programming language, would require excessive detail and precision.

Instead, we will use what is often called *pseudocode*. This is a language in which we can express algorithms. It has many of the constructs of modern programming languages. But it makes life easier in a number of ways. Most radically, we allow ourselves to include informal English statements, provided they are reasonably precise and reasonably unambiguous. We also rid ourselves of many of the awkward details of real programming languages. For example, we will allow ourselves to write well-recognised abbreviations such as $3x$ instead of `3 * x`. Similarly, we allow oursleves to write 3^2 instead of the more 'linear' forms that are found in programming languages such as `3 * 3` or `3 ** 3` or `3↑2` or `Math.exp(3, 2)`. (Which of those would be the one supported by Java?) We also won't get too bogged down in details such as consistent use of semi-colons for ending or joining statements.

In other words, you can write pretty much what you like! However, I advise you to stick more or less to what is covered in this lecture.

I am going to give our pseudocode language a name, so that I can refer to it, for example in exam questions. The name I choose is DECAFF.

Here's an example of an algorithm written in DECAFF. We discuss the details in the rest

of the lecture.

Algorithm: BINARYSEARCH$(x, a, lower, upper)$

Parameters: x is an integer; $a[lower \ldots upper]$ is an array of distinct integers stored in non-decreasing order; $0 < lower \leq upper$.
Returns: The position of x in a if found, otherwise **fail**.

```
{   lo := lower;
    hi := higher;
    while lo ≤ hi
    {   mid := (lo + hi) div 2;
        if a[mid] < x
        {   lo := mid + 1;
        }
        else if a[mid] = x
        {   return mid;
        }
        else
        {   hi := mid − 1;
        }
    }
    return fail;
}
```

3.2. Algorithm Name and Problem Specification

- We optionally begin with the name of the algorithm and the formal parameters.

Algorithm: BINARYSEARCH$(x, a, lower, upper)$

- Then we optionally give the *problem specification* for the problem that this algorithm solves. This has the advantage of giving more details about the parameters, e.g. their type (integer, Boolean, etc.). but we'll omit it when the problem that we are solving is clear from the context.

Parameters: x is an integer; $a[lower \ldots upper]$ is an array of distinct integers stored in non-decreasing order; $0 < lower \leq upper$.
Returns: The position of x in a if found, otherwise **fail**.

3.3. Data Types

- A *data type* is
  - a *set of values*; and
  - the *operations* we can use to create, access and modify the values.

3.3.1. Integers

- For integers, we allow two kinds of operation.
- Operations on integers that return other integers:

$$+ \qquad - \qquad \times \qquad \text{div} \qquad \text{mod}$$

Three of these are obvious: $+, -$ and $\times$. We use div to indicate integer division, i.e. the result will be an integer. Integer division truncates twowards zero. So, e.g., $7 \operatorname{div} 2 = 3$ and $-7 \operatorname{div} 2 = -3$. We use mod for modulo, i.e. $x \bmod y$ gives the remainder after integer division of x by y. Let Q be $x \operatorname{div} y$ and R be $x \bmod y$, then $x = Q \times y + R$. So, for example, $7 \bmod 2 = 1$ and $-7 \bmod 2 = -1$.

The symbols you would use for these in Java are: `+`, `-`, `*`, `/` and `%`, respectively. There is no penalty if you use these Java symbols: I will know what you mean.

Equally, abbreviations, such as $3x$ instead of $3 \times x$ (or `3 * x`) are also acceptable.

We will also, on occasion, assume that we can raise to a power (x^y). There may be other things which we will allow. We'll introduce them as we need them.

And, one final thing: what about illegal operations? We disallow division by zero (using div or mod). But we won't worry about overflow. We'll assume that our computer can perform the five operations on numbers no matter how great their magnitude.

- Operations on integers that return Booleans:

$$= \quad \neq \quad < \quad \leq \quad > \quad \geq$$

These are all obvious. Feel free to also use the following if you find them convenient: $\not<, \not\leq, \not>$ and $\not\geq$. Of course, they aren't necessary because each is equivalent to one of the ones above, e.g. $\not<$ is equivalent to $\geq$.

The symbols you would use in Java are `==`, `!=`, `<`, `<=`, `>` and `>=`. The first of these is, of course, a pair of equal signs. Writing a single equals sign in Java when you should have written a pair is one of the commonest Java errors. In DrCaff, I am going to use a single equals sign, as we would in maths. I don't mind which you use.

3.3.2. Booleans

- We also allow Booleans.
- There are just two values: **true** and **false**.
- Operations on Booleans that return other Booleans:

$$= \quad \neg \quad \wedge \quad \vee$$

p	$\neg p$
true	**false**
false	**true**

p	q	$p \wedge q$	$p \vee q$
true	**true**	**true**	**true**
true	**false**	**false**	**true**
false	**true**	**false**	**true**
false	**false**	**false**	**false**

- We will introduce some other types as we proceed through the module (e.g. lists). But integers and Booleans will suffice for now.

3.3.3. Precedence and Associativity

When there are multiple operators in an expression, how should evaluation proceed? Consider

$$2 + 3 \times 4$$

This evaluates either to the number 20 or to the number 14. It depends on which operator you evaluate first. If you apply $+$ to 2 and 3 to give 5 and then apply $\times$ to 5 and to 4, the whole expression has 20 as its value. However, if first you apply $\times$ to 3 and 4 to give 12 and then apply $+$ to 2 and 12, the whole expression has 14 as its value.

Consider also

$$5 - 2 + 3$$

This evaluates either to the number 6 or to the number 0. $5 - 2$ gives 3, then $3 + 3$ gives 6. Alternatively, $2 + 3$ gives 5 and $5 - 5$ gives 0.

Parentheses provide one way to disambiguate. They make the evaluation order explicit:

$(2 + 3) \times 4$	evaluates to the number 20
$2 + (3 \times 4)$	evaluates to the number 14
$(5 - 2) + 3$	evaluates to the number 6
$5 - (2 + 3)$	evaluates to the number 0

We will use parentheses liberally. But there are ways to avoid *some* parentheses, as we will now illustrate.

One way to do away with parentheses is to have a convention about which of two operators is applied first: these conventions are known as *operator precedence rules* and *operator associativity rules*. The following rules are based on those used in Java.

Here are the operator precedences:

$$\begin{array}{cl} \neg & \textit{highest precedence} \\ \times, \text{div}, \text{mod} & \\ +, - & \\ <, \leq, >, \geq & \\ =, \neq & \\ \wedge & \\ \vee & \textit{lowest precedence} \end{array}$$

So

$$2 + 3 \times 4$$

is evaluated as

$$2 + (3 \times 4)$$

because $\times$ has higher precedence than $+$.

The associativity rules tell us what to do in the event of a tie in precedence. Again we use Java's rules. In Java all operators are *left-associative*. So

$$5 - 2 + 3$$

is evaluated as

$$(5 - 2) + 3$$

because $+$ and $-$ associate to the left.

(As a matter of passing interest, Java does in fact have one right-associative operator. It's not one of the operators discussed above. I'll leave you to find out what it is from a Java textbook, if you're interested.)

Remember, you can override the precedence and associativity rules with parentheses. E.g. you would use

$$(2 + 3) * 4$$

$$5 - (2 + 3)$$

to obtain different evaluation orders.

3.4. Assignment

- Assignment is the first of the commands that we allow in DECAFF.
- A *variable* is a named storage location that can hold a value of a particular type.
- The *assignment* command is used to store the value of an expression into a variable:

$mid := (lo + hi) \operatorname{div} 2;$

- In programming languages, you have to follow numerous rules concerning variables. These rules concern declaration, initialisation, scope, lifetime and visibility. For example, in Java, you would have needed to declare the variable `mid`, indicating its *type* and giving some initial value:

```
int mid = 0;
```

  In DECAFF, we take a low-ceremony approach: use whatever variables you need, when you need them!

- Note that in DECAFF, I prefer to use the symbol := for assignment, whereas in Java the equals sign, `=`, is used. I don't mind which you use; there's no penalty either way.
- Remember that DECAFF is pseudocode. So if we want, we can mix in statements of English (provided they are reasonably precise and unambiguous), e.g.:

$z :=$ the larger of x and y;

- We allow arrays in DECAFF.
- In DECAFF, arrays are indexed fixed-length collections of values of the same type.

  Mostly, we'll use arrays of integers. But arrays of other data types are also allowed, e.g. arrays of Booleans, arrays of arrays, etc.

  In Java, the cells of an array are indexed starting from 0. So an array of length n has cells that are indexed by the integers from 0 to $n-1$. Perhaps confusingly then, the first cell is cell 0, the second is cell 1, the third is cell 2, and the last is cell $n-1$ (not n).

  In DECAFF, we choose how we want to refer to the cells.
- $a[1 \ldots n]$ is an array of length n, indexed from 1 to n. (N.B. $1 \ldots n$ are the *indexes* of the cells, **not** the *contents*.)
- $a[0 \ldots n-1]$ is an array of length n, indexed from 0 to $n-1$. (This is like Java.)
- $a[lower \ldots upper]$ is an array of length $upper - lower + 1$, indexed from $lower$ to $upper$. This is a nice general way to specify an array.
- Array assignment changes the value of one of the components:

$$a[i+1] := a[j] \times 2;$$

3.5. Sequence

- Control structures enable us to determine the order of execution of the commands within an algorithm.
- The most basic control structure is *sequence*.
  - You have several commands and you want them executed one after the other.
  - In DECAFF, simply write them one after the other!

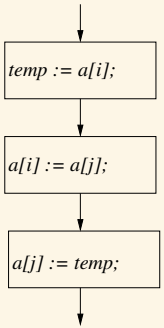
$$C_1; C_2; \ldots; C_n$$

- A sequence of one or more commands can be grouped into a single command called a *block* or *compound command* using curly braces.

$$\{C_1; C_2; \ldots; C_n\}$$

- Here's an example:

$$temp := a[i];$$
$$a[i] := a[j];$$
$$a[j] := temp;$$



- Here it is as a block:

$$
\begin{array}{l}
\{ \quad temp := a[i]; \\
\quad\; a[i] := a[j]; \\
\quad\; a[j] := temp; \\
\}
\end{array}
$$

3.6. Conditional

One-armed conditional: There are occasions when we want a statement to be executed only if a certain condition is satified.

if B
  C;

If Boolean expression B evaluates to **true**, then command C is executed. If B is **false**, C is not executed. Remember that C can be a block of commands (which acts like a single command). I like to write the braces around C, even when C is a single command.

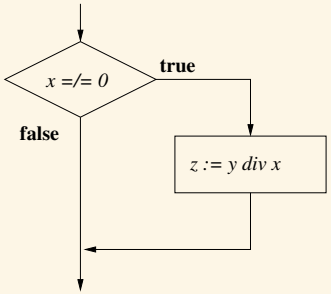
Two-armed conditional: There are other occasions when we want to select one of two commands according to the condition.

if B
  C_1;
else
  C_2;

If Boolean expression B evaluates to **true**, then command C_1 is executed. If B is **false**, then C_2 is executed. Again C_1 and/or C_2 can be blocks of commands. Using curly braces, even when not strictly needed, will avoid certain errors – see the next lecture.

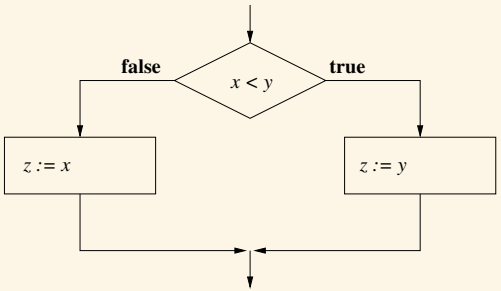
Here's an example of a one-armed conditional.

```
if x ≠ 0
{    z := y div x
}
```



And here's an example of a two-armed conditional.

```
if x < y
{    z := y
}
else
{    z := x
}
```



3.7. Iteration

- If our only control structures are sequences and conditionals, then no part of an algorithm would ever be executed more than once. We introduce iteration (loop) control structures to allow parts of the algorithm to be executed multiple times.
- *Iteration* control structures typically comprise:
  - a *body* — the command that may be executed multiple times;
  - a *test* — a Boolean expression that determines whether to execute the body another time or whether to exit the loop
- In *unbounded iteration*, when the processor encounters the loop, it does not know how many times the body of the loop is to be executed.
- In *bounded iteration*, when the processor encounters the loop, it does know how many times the body of the loop is to be executed.

3.7.1. Unbounded Iteration

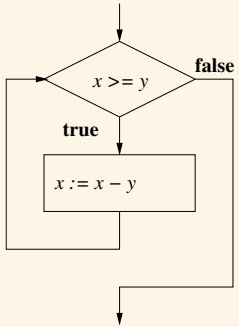
- **while** loops are perhaps the most common forms of unbounded iteration.

while B
  C;

- If Boolean expression B evaluates to **true**, then command C is executed, *and then the **while** command is repeated.* If B is **false**, C is not executed.

- Thus, C is executed repeatedly for as long as B evaluates to **true**.
- Again C can be a block of commands.
- Here's a concrete example.

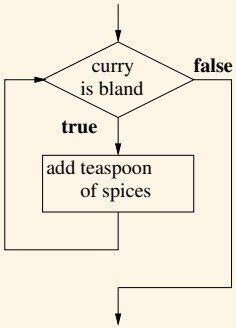
```
while x ≥ y
{    x := x − y;
}
```



- There are at least four types of unbounded iteration depending on
  - placement of the test: *pre-test* or *post-test* i.e. whether the test is made before or after executng the body.
  - nature of the test: *exit-when*-**true** or *exit-when*-**false** i.e. whether the loop is exited when the test evaluates to **true** or when it evaluates to **false**.
- So let's look at the four possibilities. The first is just the **while** loop that we have already looked at.

3.7.1.1. Pre-test, exit-when-false

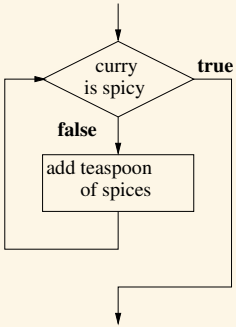
```
while curry is bland
{    add teaspoon of spices
}
```



Java has this.

3.7.1.2. Pre-test, exit-when-true

```
until curry is spicy
{    add teaspoon of spices
}
```

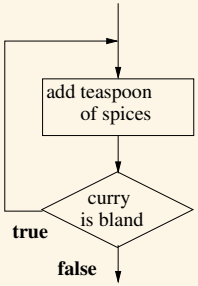


Java does not have this.

The key observation about pre-test loops is that the body may not be executed even once.

3.7.1.3. Post-test, exit-when-false

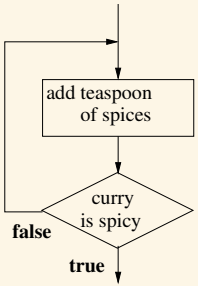
```
do
{    add teaspoon of spices
}
while curry is bland
```



Java has this.

3.7.1.4. Post-test, exit-when-true

```
do
{    add teaspoon of spices
}
until curry is spicy
```



Java does not have this.

The key observation about post-test loops is that the body will definitely be executed at least once.

Class Exercise

Choosing the wrong loop type is a major cause of error. This can happen even when you only have a choice of two loop types, as you do in Java!

What's wrong with this and how would you fix it?

Parameters: A list, L, of integers.

Returns: The sum of the integers in the list.

```
{   sum := 0;
    do
    {   x := the next number from L
        sum := sum + x;
    }
    until the list is empty
    return sum
}
```

3.7.1.5. Infinite Loops

It is easy to deliberately or inadvertently write infinite loops using the unbounded iteration constructs. Here are some examples.

```
while True
    C;
```

```
while 1 = 1
    C;
```


```
x := 11;
while x ≠ 0
    x := x − 2;;
```

```
q := 0;
while x ≠ 0
{    x := x − y;
     q := q + 1;
}
return q;
```

Look at the last example to the right. Assume that x and y are the formal parameters and that they are both positive integers. This algorithm will sometimes terminate (for some values of x and y) and sometimes will not.

Make a guess at what problem the algorithm is supposed to solve. On what values does it terminate, and on what values does it fail to terminate? How would you fix the algorithm so that it always terminates and does correctly solve the problem?

3.7.2. Bounded Iteration

- In *bounded iteration*, when the processor encounters the loop, it does know how many times the body of the loop is to be executed.
- Typically, some kind of **for** loop is used to give bounded iteration. Here's an example of the simplest kind of **for** loop:

```
for i := 1 upto n
{    sum := sum + i;
}
```

Let V be a variable; let E_1 and E_2 be arithmetic expressions that evaluate to integers; let L be a finite list; and let S be a finite set. Then we allow each of these:

```
for V := E1 upto E2
    C;
```

e.g.

```
for i := lower upto upper
{    a[i] := 0;
}
```

You might be surprised to learn that the exact way that **for** loops work differs from programming language to programming language. We will discuss this further in a lecture on programming language semantics in a few lectures time.

For now, I'll just briefly say how I intend our very simple **for** loops to work. In the above example, E_1 and E_2 are evaluated once, when the loop is encountered, to get values. They are not re-evaluated every time round the loop.

Suppose the values of E_1 and E_2 when the loop is encountered are e_1 and e_2 respectively. Variable V is assigned e_1. If $V \leq e_2$, the loop body, C, is executed. Then V is incremented by 1. Then the **for** loop is repeated from the test.

It is illegal in DECAFF for you to include commands in the loop body, C, that in any way alter the value in V.

We will also allow the following in DECAFF:

```
for V := E1 downto E2
    C;
```

e.g.

```
for i := 10 downto 3
{    x := y
}
```

The difference this time is that the loop body is executed if $V \geq e_2$ and, after executing C, V is decremented by 1.

And we will allow these also (where L is a finite-length list and S is a finite-sized set):

for each V **in** L
    C;

e.g.

for each x **in** L
{ $sum := sum + x$;
}

for each V **in** S
    C;

e.g.

for each x **in** S
{ display x on the screen;
}

These two are much like the previous two. L and S are evaluated once, when the loop is encountered. (Or, equivalently, you are not allowed to include commands in C that in any way alter the contents of L and S.)

Variable V is assigned the first element in L or S, if there are any elements. If V has been successfully assigned an element, the loop body is executed.

After C has been executed, V is assigned the next element in L or S, if there is one. Then the **for each** loop is repeated from the test.

With DECAFF's highly simplified **for** and **for each** loops, it is impossible to write an infinite loop. This is not the case with, for example, Java's much more powerful (even dangerously powerful) **for** loop.

Acknowledgements

The binary search algorithm comes from [Raw91].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

References

[Raw91] G. J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. W. H. Freeman, 1991.