Teaching guide: Programming concepts (Selection)

This resource will help with understanding the use of selection in programming concepts. It supports Section 3.2.2 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning outcomes:

- Understand how selection is used to structure flow through a program.
- Be able to use one branch and two branch selection (IF and IF-ELSE respectively).
- Be aware of multiple branch selection (such as CASE) and realise its equivalence to nested IF-ELSE statements.

Making choices

You are about to leave the house to walk to the train station and you look out of the window. You might then make this reasonable choice:

- if it is raining, I'll wear a coat.

This is an example of one branch selection: take an expression that can have a value of either `True` or `False` (a Boolean expression) and based on the value of that expression perform an action. Boolean expressions and selection are used all the time in real life:

- if the road is clear I can safely cross it
- if I have enough money I can buy a computer
- if my music is too loud the neighbours will complain
- if the pan is hot I'll burn myself when I pick it up.

It is impossible to live independently unless we make these reasoned decisions based on whether something is true or not. (In real life, many things are neither exactly true nor false – for example 'it is raining' is not absolute as it could be slightly damp, a bit drizzly or a howling storm – however in computer programming we will deal only with expressions that are either completely true or completely false.)

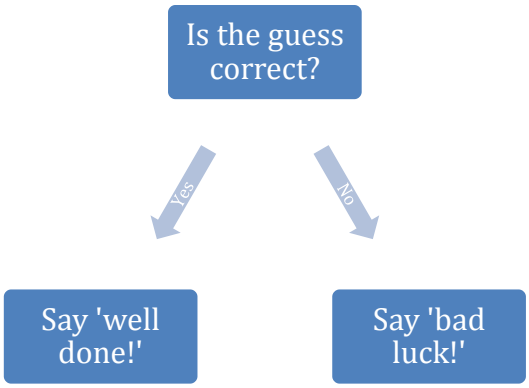
Programming with selection is even simpler than writing sentences about it. Applying this to a guessing game where the guesser only has one go at guessing

the number and if they get it correct then they get a 'well done' message. We could write this program as:

```
setters_number ← USERINPUT
guess ← USERINPUT
IF setters_number = guess THEN
   OUTPUT 'well done!'
ENDIF
```

The message is only outputted if the Boolean condition `setters_number = guess` evaluates to `True`. The program does not loop back in the same way as it would if a `WHILE` instead of an `IF` had been used. Selection is about making the choice of what to execute and, once it is executed, continuing with the remainder of the program.

The guessing game program doesn't tell the guesser if they got the answer wrong, so you can use another technique called two-branch selection which creates a fork in our program. If the value of a Boolean condition is `True` then the program forks one way, but if it is `False` it forks another (the Boolean expression is only evaluated once to achieve this).



As a computer program, we would write this as:

```
setters_number ← USERINPUT
guess ← USERINPUT
IF setters_number = guess THEN
   OUTPUT 'well done!'
ELSE
   OUTPUT 'bad luck!'
ENDIF
```

If you return to a previous version of our game (discussed in the Programming concepts (iteration) resource) that allows for multiple guesses, you can include selection to automate the reply to whether the guess was too high or too low although we do need to be careful here. You could write this program:

```
setters_number ← USERINPUT
num_of_guesses ← 0
   REPEAT
   guess ← USERINPUT
```

```
    num_of_guesses ← num_of_guesses + 1
    IF guess < setters_number THEN
        OUTPUT 'guess was too low'
    ELSE
        OUTPUT 'guess was too high'
    ENDIF
UNTIL setters_number = guess
OUTPUT num_of_guesses
```

This program would work for all guesses that are either too high or too low but when the correct answer is guessed it would still output `'guess was too high'`. In this instance an `IF-ELSE` is the wrong decision and you need to use two `IF`s to check if the guess was too low or too high. Using `ELSE-IF` (introduced later in this section) would be an even better choice.

```
setters_number ← USERINPUT
num_of_guesses ← 0
REPEAT
    guess ← USERINPUT
    num_of_guesses ← num_of_guesses + 1
    IF guess < setters_number THEN
        OUTPUT 'guess was too low'
    ENDIF
    IF guess > setters_number THEN
        OUTPUT 'guess was too high'
    ENDIF
UNTIL setters_number = guess
OUTPUT num_of_guesses
```

The general pattern for one branch selection is:

```
IF Boolean expression THEN
    # do this if the Boolean expression is True
ENDIF
```

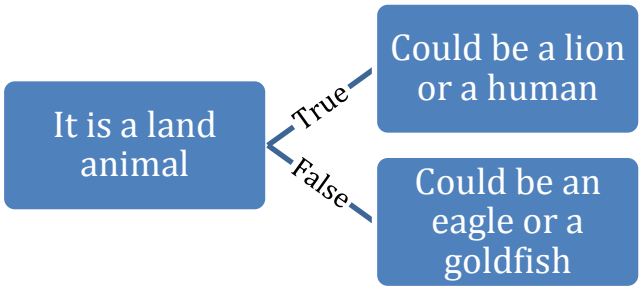
The general pattern for two branch selection is:

```
IF Boolean expression THEN
    # do this if the Boolean expression is True
ELSE
    # do this if the Boolean expression is False
ENDIF
```

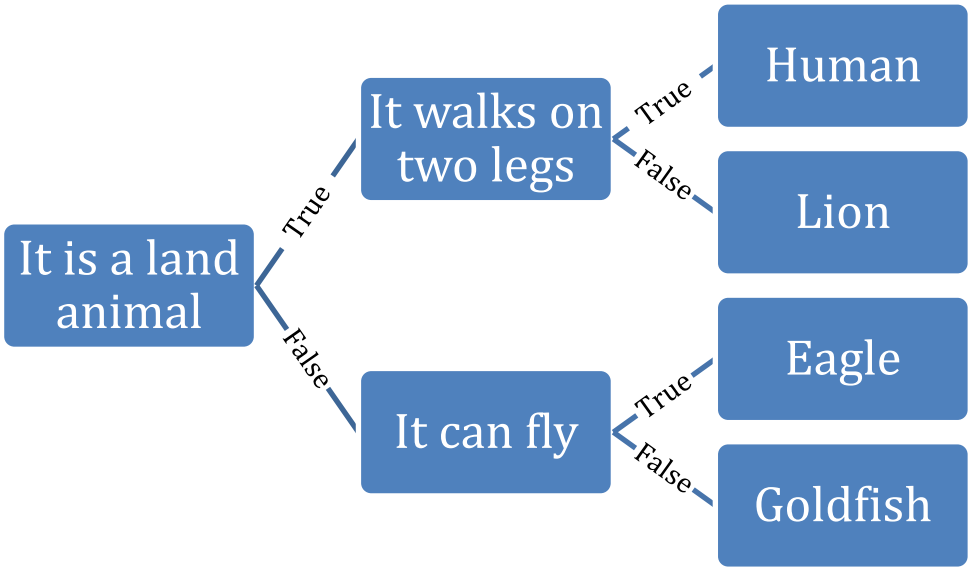
Nested selection

Just as loops can be nested, so can selection statements. Using the example of animal classification where you can make Boolean expressions about an animal and if it is `True` it takes one branch or if it is `False` another and then evaluate

other Boolean expressions until you are able to identify the animal. Instead of covering all of the animal kingdom (which would require a significant number of selection structures), we will consider a world in which the only animals are humans, lions, eagles and goldfish. We want a Boolean expression that divides these four different animals into two separate groups: humans and lions are the only two land animals so answering the expression 'is it a land animal' divides the animals like so:



You can now use different Boolean expressions to divide lions and humans from each other and also eagles and goldfish.



Writing this directly as a program is difficult because you don't yet have the syntax to be able to ask questions like whether it can walk on two legs (a good solution to this would be to create animal objects – this style of programming called Object-Oriented programming is covered at A-Level). You can implement it using user input though:

```
OUTPUT 'is it a land animal?'
answer ← USERINPUT
IF answer = 'yes' THEN
   OUTPUT 'does it walk on two legs?'
   answer ← USERINPUT
   IF answer = 'yes' THEN
      OUTPUT 'human'
   ELSE
```

```
      OUTPUT 'lion'
   ENDIF
ELSE
   OUTPUT 'does it fly?'
   answer ← USERINPUT
   IF answer = 'yes' THEN
      OUTPUT 'eagle'
   ELSE
      OUTPUT 'goldfish'
   ENDIF
ENDIF
```

The outer `IF-ELSE` divides the animals into two groups, the inner `IF-ELSE`s further divide them at which point it is clear what the animal is.

Sometimes you want programs to branch based on a choice of many values, not just two. For example, this program outputs the name of the first four months based on their numerical position in the year:

```
month ← 3
IF month = 1 THEN
   OUTPUT 'January'
ELSE
   IF month = 2 THEN
      OUTPUT 'February'
   ELSE
      IF month = 3 THEN
         OUTPUT 'March'
      ELSE
         IF month = 4 THEN
            OUTPUT 'April'
         ELSE
            OUTPUT 'Not one of the first four months'
         ENDIF
      ENDIF
   ENDIF
ENDIF
```

This, although correct, is beginning to look a little messy as every nested `IF` indents our code further to the right (if you continued to December then you would have run out of space on the page). Some programming languages adapt the way they use `IF-ELSE` so the above program can be rewritten using `ELSE-IF` statements: Python uses the keyword `elif` to achieve this.

```
month ← 3
IF month = 1 THEN
   OUTPUT 'January'
ELSE IF month = 2 THEN
   OUTPUT 'February'
```

```
ELSE IF month = 3 THEN
   OUTPUT 'March'
ELSE IF month = 4 THEN
   OUTPUT 'April'
ELSE
   OUTPUT 'Not one of the first four months'
ENDIF # end all IFs
```

Many languages also have structures that allow programmers to branch based on one of many particular values or cases – quite often they use the keyword `CASE` or `SWITCH`. Logically this is not normally different from using nested `IF-ELSE`s or the second program using `ELSE-IF` although sometimes the meaning of these `CASE` style structures are different and allow more than one of the branches to execute if the value matches more than one particular case.

The previous program could be rewritten as:

```
month ← 3
CASE month OF
   1: OUTPUT 'January'
   2: OUTPUT 'February'
   3: OUTPUT 'March'
   4: OUTPUT 'April'
   DEFAULT: OUTPUT 'Not one of the first four months'
ENDCASE
```

With the exception of subroutines, this is the last of the fundamental building blocks of a structured program. If you look at any program written in Java, Pascal, Python, VB.net and so on there may well be many lines of code that still look very confusing, but you will see that a large part of the code is comprised of variables, iteration and selection.