PROGRAMMING LOGIC AND TECHNIQUES

Algorithms: Loops

Fixed repetition structures: FOR

- **FOR** loops are **fixed repetition structures.** They are used to execute an instruction (or a set of instructions) a certain number times in a row.

Fixed repetition structures: FOR

- In the pseudocode for this course, the number of repetitions that a **FOR** loop performs must be established before the loop begins.
  - However, note that this is not necessarily the case in programming in general, even though it is common to use a pre-set, constant number of iterations (repetitions) when using **for** loops.
  - In fact, some complex looping behavior depends upon using variable criteria for continuing/exiting the loop.

Fixed repetition structures: FOR

- The use of **loops** (repetition structures) can prevent code duplication.
- Loops also enable complex and useful program behavior that cannot be achieved without them.
- Here is a short example to help demonstrate how useful such a structure can be:

Fixed repetition structures: FOR

- To calculate the square of a number, you need to use the following instructions:

```
READ a
c ← a * a
WRITE c
```

- But what if you had to execute these instructions 50 times? You would have to copy out the 3 lines of code needed for the calculation 50 times.
- Technically, this is possible, but it is very inefficient. It would result in code duplication, and would cause the application to both take longer to develop and be more difficult to update.

Fixed repetition structures: FOR

- Therefore, we will use a repetition structure that allows us to repeatedly execute the instructions without duplicating the code.

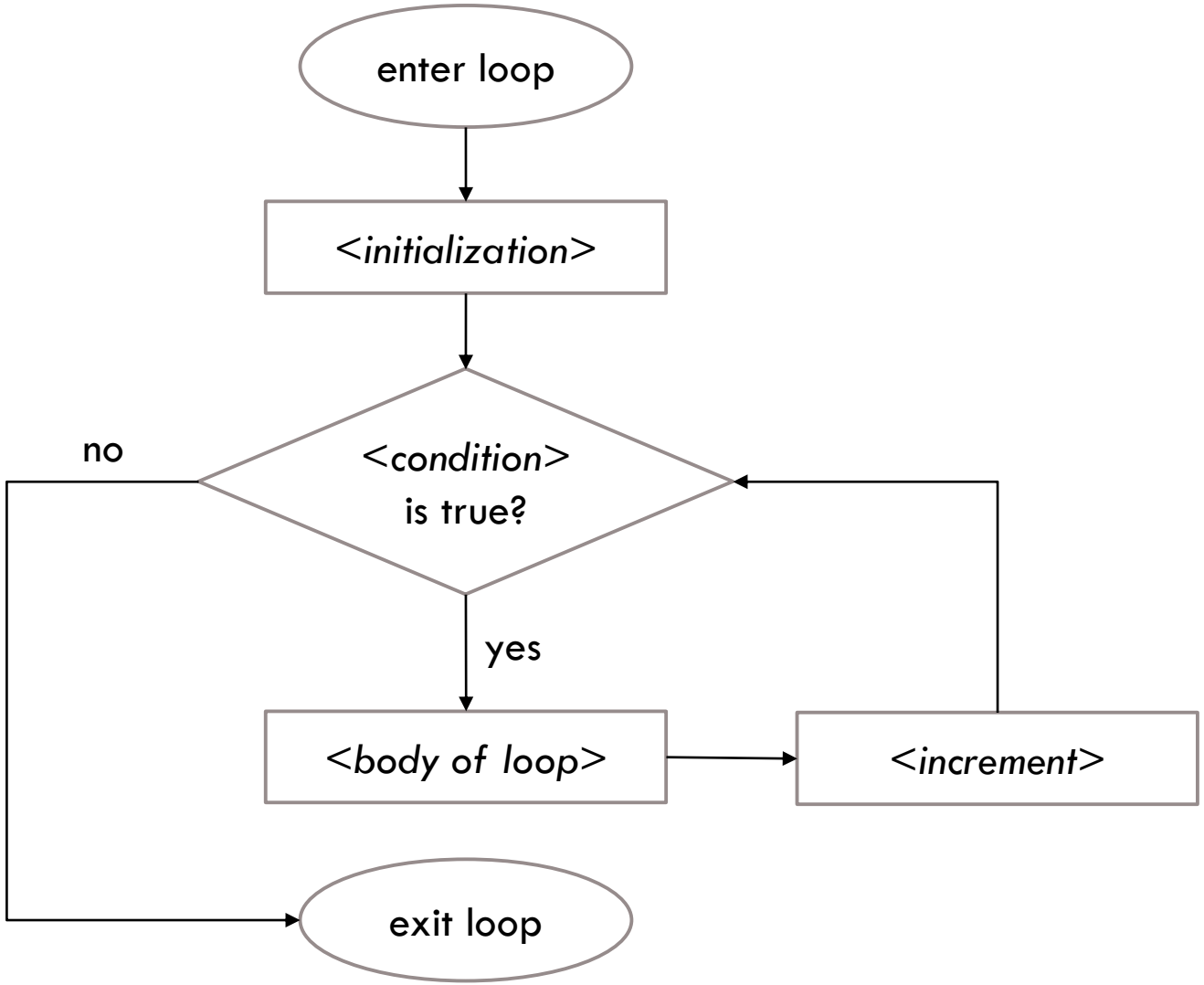
Syntax:

FOR (<variableName> ← <min> **TO** <max>)

    <procedure>

ENDFOR

Fixed repetition structures: FOR



Fixed repetition structures: FOR

- In the case of our example, the procedure will consist of the 3 instructions for squaring numbers, and the loop will allow it to repeat 50 times. Here are the necessary instructions:

```
VARIABLES
    Integer : a, c, i
START
    FOR (i ← 1 TO 50)
        READ a
        c ← a * a
        WRITE c
    ENDFOR
END
```

A mistake to avoid

- Note: by default, the counter variable increments by 1
- Therefore, you should avoid changing the value of the counter variable (**i**, in this case) with an instruction that is inside the loop, as is done in the following example:

```
FOR (i ← 1 TO 50)
    READ a
    c ← a * a
    WRITE c
    i ← i + 1
ENDFOR
```

A mistake to avoid

- The mistake here is the modification of the variable **i**. By changing the value of **i**, the total number of times that the loop will execute its procedure will be fewer. This is because the statement **i ← i + 1** ensures that **i** increments faster than normal.

A mistake to avoid

- Accordingly, you should avoid changing the value of the counter variable from inside the loop.
  - In fact, contrary to this, sometimes in programming it is appropriate to modify the counter from within a loop.
  - However, this is an advanced technique for more complex algorithms, and it must be done carefully and with good reason. So for this course, we should avoid doing so.

The increment value

- By default, a loop's **increment value** is 1. However, increment values can be modified.

The increment value

- In cases where it is necessary to increment by values other than 1, we proceed as follows:

```
FOR (i ← 1 TO 30 JUMP 2)
    <procedure>
ENDFOR
```

In this example, the procedure will be executed 15 times.

The increment value

- It is also possible to carry out a **decrementation**. To accomplish this, the increment value must be negative. Here is an example:

```
FOR (i ← 20 TO 1 JUMP –1)
    <procedure>
ENDFOR
```

Here, the value of **i** starts at 20, and the procedure will be executed until the value of **i** is 1.

As with incrementation, it is possible to decrement with **JUMP** values other than –1.

Alternative syntax

- In this course, we can also use the following syntax for writing **FOR** loops. This syntax is ultimately more explicit, clear, and flexible than the easier syntax given previously. This syntax is equivalent to what is actually used in most programming languages:

- Syntax:

- **FOR** (<initialization>**;** <condition>**;** <increment>**)**
  <procedure>
  ENDFOR

Alternative syntax

- For example, the following two **FOR** loops lines are equivalent:

- FOR (i ← 0 TO 9 JUMP 1)
  <procedure>
  ENDFOR

- FOR (i ← 0; i <= 9; i ← i + 1)
  <procedure>
  ENDFOR

Exercise

- For a company with **ten** employees:
- Given each employee's hourly wage and number of hours worked this week, make a program that calculates and displays the net salary of each employee. In addition, for all hours worked of overtime (over 40 hours), increase the hourly wage by 50%.
- Tax rates vary as follows:
  - 20% for less than $500
  - 30% for $500 and over

Flexible repetition structures

- Apart from fixed repetition structures, there is another category of loops known as **flexible repetition structures.**

- The instructions contained in a flexible loop's procedure are repeated **while** a condition is satisfied (as long as it remains true). Thus, we can say that loops of this kind are loops with flexible repetition.

- Two types of loop fall under this category:
  - **WHILE** loops
  - **REPEAT…WHILE** loops

Flexible repetition structures

Syntax:

WHILE (<condition>) **DO**

<procedure>

ENDWHILE

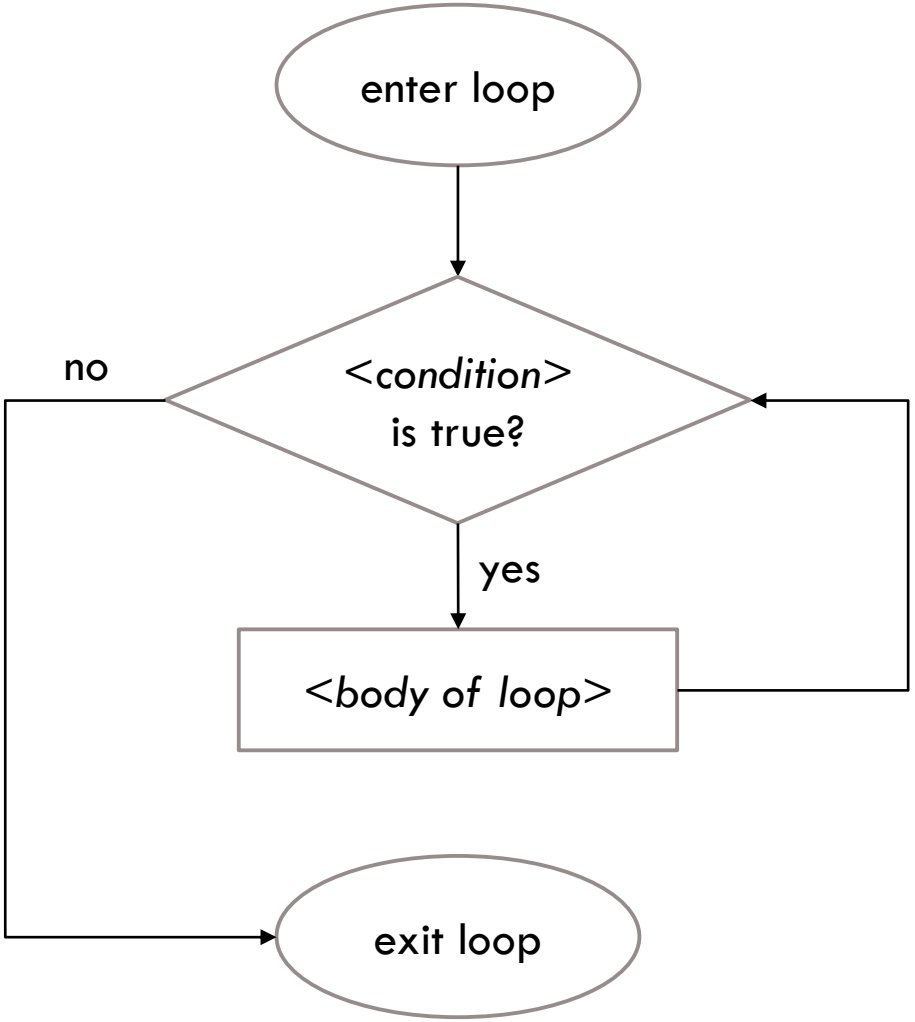
or

REPEAT

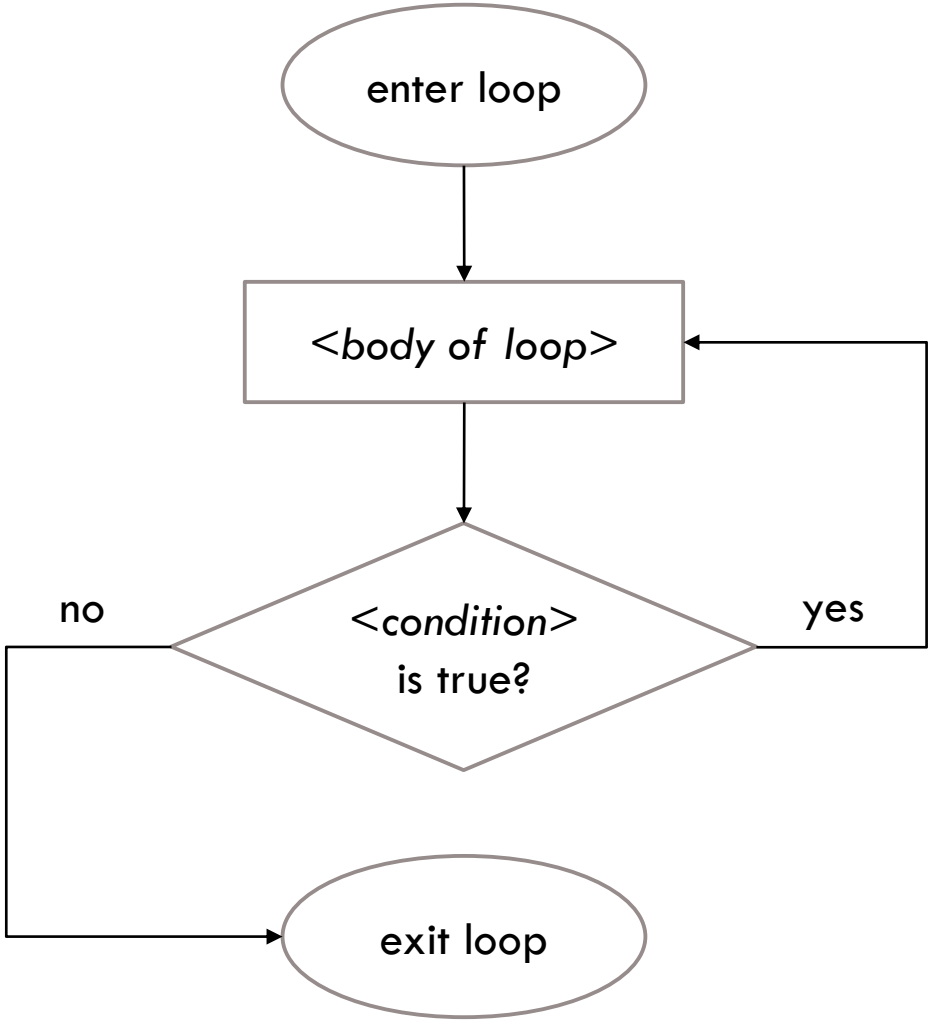
<procedure>

WHILE (<condition>)

Flexible repetition structures: WHILE



Flexible repetition structures: REPEAT…WHILE



Flexible repetition structures

- The two structures shown in the syntax section above appear similar. However, on closer examination, you'll notice the following difference:
  - In the first structure (**WHILE** loop), the procedure will only be executed if the condition is satisfied.
  - In the second structure (**REPEAT…WHILE** loop), the procedure will be executed at least once, since the condition is only evaluated after the procedure has been run.

Flexible repetition structures

- The conditions that appear in flexible loops are analogous to the conditions used in conditional statements, and they are constructed and evaluated in the same way.

- As such, they can be built up using **relational operators** and **logical operators**, and they are evaluated as being either **true** or **false**.

Flexible repetition structures

- Here is an example of a **REPEAT…WHILE** loop:

```
VARIABLE Integer : number
REPEAT
        WRITE "Enter a number (0 to quit): "
        READ number
        WRITE number * 2
WHILE (number <> 0)
```

This structure multiplies the number entered by the user by 2. The procedure is executed as long as the user does not enter the number 0. Note that the procedure will execute at least once.

Flexible repetition structures

- Here is an example of a **WHILE** loop:

```
VARIABLE Integer : number
WRITE "Enter a number: "
READ number
WHILE (number <= 0) DO
        WRITE "Enter a number: "
        READ number
ENDWHILE
```

In the structure above, the procedure will be executed as long as the user enters a value less than or equal to 0. Using a **WHILE** loop in this way enables input validation.

Flexible repetition structures

- You should choose which loop structure to use based upon the kind of procedure you want to implement. To make this choice, you only need to ask yourself one question: "Should the procedure be executed **at least once**?" If the answer to this question is yes, you should use the second structure:
  - **REPEAT**

    <procedure>

    WHILE (<condition>)

Exercise

- For a company in which the number of employees varies from week to week:
- Given each employee's hourly wage and number of hours worked this week, make a program that calculates and displays the net salary of each employee. In addition, for all hours worked of overtime (over 40 hours), increase the hourly wage by 50%.
- Tax rates vary as follows:
  - 20% for less than $500
  - 30% for $500 and over

Another use of flexible loops

- Flexible repetition structures are often used to validate input entered by the user.
- For example, let's readdress the net salary problem:

Another use of flexible loops

- Given an employee's hourly wage and number of hours worked this week, make a program that calculates and displays the net salary of the employee. In addition, for all hours worked of overtime (over 40 hours), increase the hourly wage by 50%.
- Tax rates vary as follows:
  - 20% for less than $500
  - 30% for $500 and over

Another use of flexible loops

- Now lets add some constraints.
- The program must ensure that the user enters an hourly wage between $10.25 and $30 (inclusive), given that this range represents the minimum and maximum wages within this company.
- In addition, the number of hours worked must be no less than 0, and no greater than 75.

Another use of flexible loops

- To this end, we should add two flexible loops to our program that will allow us to verify that the input entered by the user is valid.
- If the input is **not** valid, the program will continue to ask the user for new values until the user's input is valid.

Another use of flexible loops

- Example 1:

```
REPEAT
    WRITE "Enter the hourly wage of the employee (the
        wage must be between $10.25 and $30.00): "
    READ hourlyWage
WHILE ((hourlyWage < 10.25) OR (hourlyWage > 30))
```

We will remain inside the loop as long as the exit condition is not true; that is, until a user enters a value that falls within the imposed limits.

Another use of flexible loops

- Example 2:

```
REPEAT
    WRITE "Enter the number of hours worked by the
        employee (the number of hours must be between
        0 and 75): "
    READ numberOfHours
WHILE ((numberOfHours < 0) OR (numberOfHours > 75))
```

This is the same type of situation as in the previous example.

Accumulators

- Definition: An **accumulator** is a variable used to progressively add up values.

  Example: A program that reads the sales prices of multiple items, each time adding the price to a progressively increasing sum (the accumulator variable), and at the end displays the final total.

Accumulators

Analysis

Input:
The prices of the items
Output:
The final amount
Constants:
None
Procedure:
Sum up the prices of all the items.

Accumulators

- Pseudo code

```
VARIABLES
  Real : itemPrice          // (Input) The price of an item
  Real : finalAmount:       // (Output) The total cost of the items
  String : anotherPrice     // (Input) For continuing the loop
START
  finalAmount = 0           // Initialization of the accumulator
  REPEAT
      WRITE "Enter the item's price"
      READ itemPrice
      finalAmount = finalAmount + itemPrice          //Accumulator
      WRITE "Would you like to enter another price (Enter Y or N )"
      READ anotherPrice
  WHILE (anotherPrice == "Y")
  WRITE "The total price of your purchase before tax is: $"
  WRITE finalAmount
END
```

Accumulators

- Rules for using accumulators:
  - An accumulator must always be initialised before the loop.
  - An accumulator must always be incremented inside the loop.

Counters

- Definition: A **counter** is a variable whose purpose is to count the number of times a loop has repeated.

  Example: A program that adds up 10 integers – using a counter variable to track the number of integers added so far – and that displays the resulting sum at the end.

Counters

Analysis

Input:
The 10 integers
Output:
The sum of the 10 integers
Internal data:
The counter
Procedure:
Add up the 10 integers that were entered by the user

Counters

- Pseudo code

```
VARIABLES
  Integer : number          // (Input) The 10 integers
  Integer : sum             // (Output) Accumulator for the sum
  Integer : numberCounter // (Internal) Counter of the numbers
START
  sum = 0                          // Initialization of the accumulator
  numberCounter = 1                // Initialization of the counter
  REPEAT
        WRITE "Enter an integer: "
        READ number
        sum = sum + number                                  // Accumulator
        numberCounter = numberCounter + 1                   // Counter
  WHILE (numberCounter <= 10)
  WRITE "The sum of the 10 integers is: ", sum
END
```

Exercises

- Write an algorithm that takes as input a decimal number and outputs the binary equivalent.

- Write an algorithm that takes as input a binary number and outputs the decimal equivalent.