CMPS 10
Introduction to Computer Science
Lecture Notes

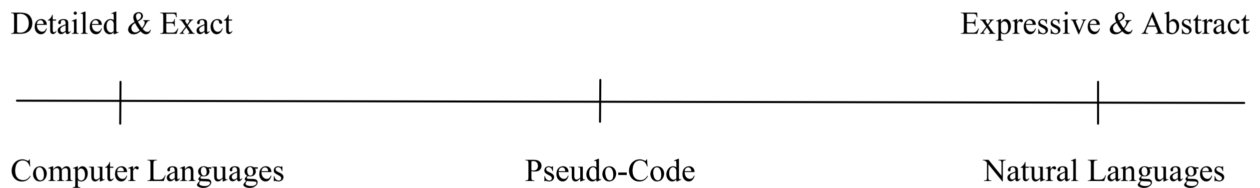
Chapter 2: Algorithm Design

How should we present algorithms?

Natural languages like English, Spanish, or French which are rich in interpretation and meaning are not ideal for this purpose. We need a more precise notation to present algorithms, that is not subject to differing interpretations.

At the other extreme *formal programming languages* (such as C, C++, Java, Perl, ADA, Python, etc.) are very inflexible in their syntax, and require much attention to detail. These details may be irrelevant in the initial design phases of an algorithm.

Instead we use an informal and flexible language called *pseudo-code* which uses English language constructs and conventions modeled to look like commands available in most computer languages.



Pseudo-code is a language in which all necessary operations can be expressed, but which is independent of any computer language, or of any computer for that matter.

Sequential Operations are of three basic kinds: *computation*, *input*, and *output*. The instruction for performing a calculation and storing the result is written as

$$\text{variable} \leftarrow \text{expression}$$

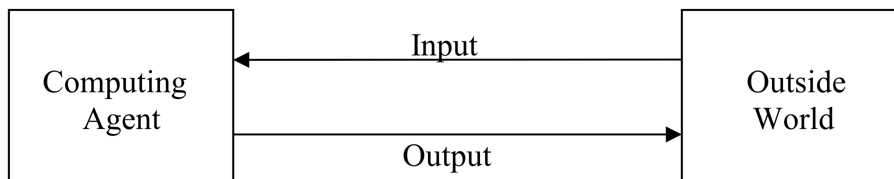
This instruction tells the computing agent to evaluate the expression on the right and assign its value in the variable on the left. The left arrow ← in this context is called the *assignment operator*.

Example $c_i \leftarrow a_i + b_i + \text{carry}$

Example $x \leftarrow \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

Unless otherwise stated, we will assume that the computing agent is at least as capable as any scientific calculator, so that the above expressions are considered primitive operations.

Input and output operations allow the computing agent to receive data values from, and send results to the outside world, i.e. the user.



The operations used for input and ouptut are get and *print* respectively.

get variable
print variable
print expression

Example get x

Example print x

Example print $x + 5$

Text to be printed will be placed in single quotes.

Example print 'error: division by zero'

An algorithm which uses only sequential operations is sometimes called a *straight line* algorithm.

Example

1) get a, b, c, d
2) sum $\leftarrow a + b + c + d$
3) average $\leftarrow$ sum / 4
4) print average

What does this algorithm do?

Conditional operations (also called *branching operations*) are expressed using the *if* statement or the *if-else* statement. The if statement is written as

```
if condition
      do something
      do something
      ..
      do something
do something
..
```

In this structure "condition" stands for a *logical* (or *boolean*) expression, i.e. an expression that can be evaluated as true or false. If condition is true, then the indented lines are executed, followed by the unindented lines. If on the other hand condition is false, then the indented lines are skipped, and execution picks up at the next unindented line. The indented commands are known as the *true branch* of the conditional. An if statement may also come with an accompanying else clause.

```
if   condition
      do something
      ..
      do something
else
      do something
      ..
      do something
do something
..
```

If condition is true, then the first group of indented instructions is executed, and the second group is skipped, followed by the next unindented instruction. If condition is false, then the first group is skipped, and the second group of indented instructions is executed, followed by the next unindented instruction. In this structure the second group of indented instructions is known as the *false branch* of the conditional, while the first group is know as the true branch, as before.

Example What does the following algorithm do?

1) get a, b
2) if $b = 0$
3)   print 'Error: division by zero'
4) else
5)   quotient $\leftarrow a / b$
6)   print quotient
7) stop

Iterative operations (also called *loops*) will be expressed in pseudo-code through three commands: *while*, *do-while*, and *for*.

```
while  condition
      do something
      ..
      do something
do something
..
```

First condition is tested, and if found to be true, the indented lines (called the *loop body*) is executed. The condition is tested again, and if found to be true, the loop body is excuted again. This process continues until the condition is found to be false, in which case execution picks up at the next unindented line. The while loop condition is sometimes called a *loop repetition condtion* since the loop body repeats only so long as this logical expression is true.

Example What does the following algorihm do?

```
1)  response ← 'yes'
2)  while  response = 'yes'
3)        get  a, b
4)        if  b = 0
5)              print  'Error: division by zero'
6)        else
7)              print  a / b
8)        print  'Do you wish to continue?'
9)        get  response
10) stop
```

The do-while loop structure puts the loop repetition condition after the loop body.

```
do
      do something
      ..
      do something
while  condition
do something
..
```

First the loop body, which consists of just those lines that are indented, is executed. Then the condition is tested, and if found to be true, the loop body is repeated. We continue executing the loop body until condition becomes false, then execution picks up at the next unindented line after while. The loop repetition condition is called a *post test* condition in this case since it is tested *after* the first execution of the loop body. The condition in the while loop on the other hand is called a *pre test* condition, since it is tested *before* the first execution of the loop body.

Example Notice that the loop body in this example is identical to the one in the preceeding example.

```
1)  do
2)        get   a, b
3)        if   b = 0
4)              print   "Error: division by zero'
5)        else
6)              print   a / b
7)        print   'Do you wish to continue?'
8)        get   response
9)  while   response = 'yes'
10) stop
```

We often encounter while loops with the following general structure

$i \leftarrow 1$ **Initialize the loop control variable *i***
while $i \le n$ **Test the loop repetition condition**
    do something with i
    $i \leftarrow i+1$ **Increment the loop control variable**

We can do these three steps, initialize loop control variable, test loop repetition condition, and increment loop control variable, more succinctly with the *for* loop structure below, which is equivalent to the preceding while loop.

for $i \leftarrow 1$ to n
    do something with i

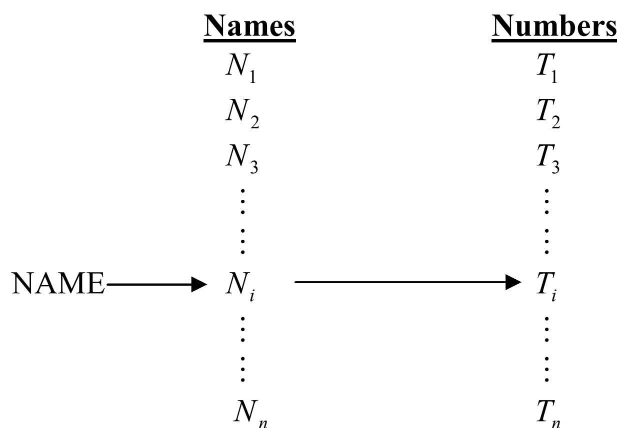
Example Here are two equivalent algorithms, one uses a while loop, and the other uses an eqivalent for loop. A trace of both algorithms would be identical.

1) $i \leftarrow 1$
2) while $i \le 5$
3) print i^2
4) $i \leftarrow i+1$
5) stop

1) for $i \leftarrow 1$ to 5
2) print i^2
3) stop

It has been proved that one can represent *any* valid algorithm with the operations so far described. We now consider some well known algorithms which perform certain basic tasks in computer science.

Example Consider the problem of looking up a name and corresponding telephone number in a directory



We assume that the names are in no particular order (i.e. not necessarily in alphabetical order) so that we must look at each name in succession. If we find a name N_i which matches our target NAME, we return the corresponding number T_i. If no N_i is a match, we print a message to that effect. We call this algorithm **Sequential Search**.

Input: $n \ge 1$ (the number of names in the list), $N_1, \cdots, N_n$ and $T_1, \cdots, T_n$ (the list of names and corresponding telephone numbers), and NAME (the target name we are searching for.)
Output: the telephone number T_i for which $\text{NAME} = N_i$, or if no such name exists, print a message to the user.

Sequential Search

1.) Get $n, N_1, \ldots, N_n, T_1, \ldots, T_n$, NAME from the user
2.) $i \leftarrow 1$
3.) found ← false
4.) while $i \le n$ and not found
5.) if N_i = NAME
6.) found ← true
7.) print T_i
8.) else
9.) $i \leftarrow i + 1$
10.) if not found
11.) print 'sorry' NAME 'not in directory'
12.) stop

The variable found is called a Boolean variable. It stores one of the two logical values true or false.