DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CSL 102 ; Introduction to Computer Science
Semester II 2006-07

Handout : Ocaml

Author **Sandeep Sen**

Contents

1 Introduction to Caml

This write-up gives a brief introduction to the Objective Caml language (referred to as Caml hereafter) through some tutorial examples. There are many advanced and important features of this language that are not covered in this document. For the purpose of the course the students are advised to restrict themselves to the features described in this document when writing Caml programs.

Caml is a very *strongly typed* language that is primarily functional in flavour although it supports full fledged imperetive features (like Pascal, C, etc). A functional programming language views every program as a mathematical function, namely one that maps input (data) to output (results). Although this is true in general for any program, most programming languages do not explicitly present computation as evaluating functions with some given arguments. Like every function has a well-defined domain and well-defined co-domain (also called range), Caml wants the programmer to specify the domain and the co-domain very precisely and refuses to proceed in case of mix-ups or ambiguity.

Suppose you want to write a program to compute the square of a given number. Using mathematical language that you are familiar with, you may define a function square as

$$\text{let square x} = \text{x times x}$$

This function takes in one parameter, namely x, and maps it to the number x^2 using the multiplication of two numbers. Multiplication is itself a function of two parameters that you may not bother to elaborate to anyone above grade 2. We are assuming implicitly that the parameter x is such that multiplication is allowed, i.e., it is a number and not, (say) an alphabet. However Caml will need information if the input numbers will be from the set of integers ($\mathbb{Z}$) or set of reals ($\mathbb{R}$). Although in hand-computation, we multiply the integers and the real numbers in a similar fashion, most computers have specialised circuits to handle multiplication of integers and reals (called floating point numbers) that are separate. So, by specifying the precise nature of the inputs, we are (implicitly) making use of specialised circuit. A deeper reason for enforcing types has its roots in program specification and verification. These are very important issues in writing large software and are outside the scope of this course.

Some of the most common types like integer, floats, boolean, character-strings are pre-defined in Caml. However, it also offers a versatile repertoire for defining new types, complex record types as well as higher-order functions. Similarly, Caml has an in-built repository of large number of common functions like addition, subtraction, division, multiplication, square, square-root, trigonometric functions etc. More importantly, it supports very powerful mechanisms for defining your own functions - this is what programming is about. So do not despair if you don't find a certain *useful* function, say exponentiation or cube-root - we should be able to build our own personalised library of functions. This is actually half the fun(ction).

2 The Caml Calculator - getting started

When you type the command **ocaml**, you are inside the Ocaml interpreter and you see the following on your screen, with the cursor next to the # prompt.

```
        Objective Caml version 1.03
#
```

You can exit ocaml by typing `#quit ;;` or cntrl-D (control-D). You can carry out simple calculations like addition, subtraction etc. The prompt `#` accepts your commands for calculations in a fairly intuitive format. Each command ends with ';;' (two consecitive semi-colons) and as soon as you press $< enter >$ the result appears in the next line beginning with "- :" followed by its type (int is integer).

```
# 3 + 5 ;; <enter>
- : int = 8
```

The $< enter >$ key does not appear on the screen but it moves the cursor to the next line. Hereafter, it will be implicit in the remaining examples. You may type a single calculation command over several lines and the result will be identical. This is because the white spaces are skipped by Caml interpreter and the result is evaluated only after the ";;" characters.

```
# 3
  + 5 ;;
- : int = 8
```

However it is advisable that you try to type it in a neat format that is easy to read.

```
# 3 + 5.0 ;;
This expression has type float but is here used with type int
```

Caml distinguishes between integers and floats (real numbers) by use of a decimal point following the number. In the above example, Caml does not like the mixing up of an integer (3) and float (5.0) in the same mathematical expression. In fact it reserves the use of '+' for integers only (both summands should be integers).

```
# 3.0 +. 5.0 ;; (* a good thumb-rule is that the floating-point operators
                   have a '.' along-side the integer operator*)
- : float = 8

# 4.0 *. 5.0 ;;
- : float = 20

# 6 / 4 ;;   (* quotient function defined for integers only*)
- : int = 1

# 6  mod 4 ;; (* remainder function  defined for integers only*)
- : int = 2
```

We know from mathematics that for two positive integers, m and n, there are unique integers q and r such that $m = n \cdot q + r$, where $0 \leq r < n$. Here q and r are called quotient and remainder respectively. Division as we know, is defined for floats only.

Notice that anything enclosed within `(* ....*)` is ignored by Caml for evaluation purposes. It is a good practice to provide comments that are helpful for future references.

```
# 6.0 /. 4.0 ;; (* real division *)
- : float = 1.5
```

Below, we give a more complete description the available functions for the types int (integers), float (reals), string and bool (boolean).

2.1 INTEGER FUNCTIONS

```
# ~- 1 ;; (*unary negation - this is also a function of one argument*)
- : int = -1

# succ 4 ;; (* successor *)
- : int = 5

# pred 5 ;; (* predecessor*)
- : int = 4

# abs ~-8 ;; (* absolute value *);;
- : int = 8

# max_int ;; (*maximum value of an integer *)
- : int = 1073741823

# min_int ;; (* minimum value of an integer *)
- : int = -1073741824
```

2.2 FLOATING POINT FUNCTIONS

```
# exp 3.0 ;; (* e to the power 3.0*)
- : float = 20.0855369232

# log 10.0 ;; (*  log to the base e *)
- : float = 2.07944154168

# sqrt 4.0 ;; (* square root *)
- : float = 2

# sin 1.5 ;;
- : float = 0.997494986604

# cos 1.5 ;;
- : float = 0.0707372016677

# abs_float ~-.2.3 ;; (* absolute value of a negative *)
```

```
- : float = 2.3

# float 3 ;; (* convert integer to float *)
- : float = 3

# truncate 5.3 (* floor function *) ;;
- : int = 5
```

2.3 STRING FUNCTIONS

```
# "abcde" ;; (* a character string *)
- : string = "abcde"

# "abc" ^ "ef2" ;; (* concatenation of two strings *)
- : string = "abcef2"
```

Note that the string ”0” is not the same as integer 0 in Caml. Although they both look alike on the screen, the internal representations are different. For example, we cannot apply the addition operation to strings ”56” and ”23”. Sometimes, you may find the following type conversion functions useful.

```
# string_of_int 853 ;; (* converts an integer to string *)
- : string = "853"

# string_of_float 23.567 ;; (* converts a float to string *)
- : string = "23.567"
```

2.4 RELATIONAL OPERATORS

These return true or false.

```
# 5 = 5 ;;
- : bool = true

# 5 = 6 ;; (* equality testing *)
- : bool = false

# 8 > 9 ;;
- : bool = false

# 9 < 8 ;;
- : bool = false

# 6 <= 6 ;; (* less than or equal to *)
- : bool = true

# 8 >= 8 ;; (* greater than or equal to *)
- : bool = true
```

```
# 9 <> 10 ;; (* not equal *)
- : bool = true
```

2.5 BOOLEAN OPERATIONS

```
  not true ;; (* negation *)
- : bool = false

# true && false (* boolean AND *);;
- : bool = false

# false || true ;; (* boolean OR *)
- : bool = true
```

Exercise 1 *What do the folowing expressions evaluate to ?*

(i) `2 + 3 * 5 ;;`

(ii) `25 + 6 * ~- 15 mod 8 ;;`

(iii) `2.0 /. 3.0 /. 2.0 ;;`

What did you expect and how can you enforce it ?

Exercise 2 *Find out the truth-table of the boolean AND (`&&`) and boolean OR (`||`) by exhaustively listing all the possible arguments.*
How many distinct boolean functions are possible that take in two arguments ?

3 Writing functions in Caml

What makes a programming language more powerful than a calculator is the ability to define new functions. We will often use symbolic references for values for the ease of writing and generalization. In mathematics and physics, it is a standard accepted practice to use symbols like π or h (Plank's constant) rather than the exact numerical value. The simplest of all is binding a value to a symbolic name. The "let" directive binds the value of the expression on the right of the "=" to the symbolic name on the left of the "=".

```
# let x = 3 ;;
val x : int = 3

#let y = x ;;
val y : int = 3

#let pi = 22.0 /. 7.0 ;;
val pi : float = 3.14285714286
```

```
#let x = 4 ;;
val x : int = 4

#y ;;  (* returns the value of the symbolic name y *)
-: int = 3 ;;
```

Notice that although x was redefined as 4, y retained its previous value, namely 3. So once a value is defined, it remains unchanged till it is explicitly changed again (*within its scope* - a concept that we will discuss in near future).

A recurring theme in functional languages is that no distinction is made between values and functions. Binding a value to a symbolic name is similar to binding a function-description to a symbolic name. This may seem a little unnatural in the beginning but you will begin to appreciate this as we write more complex programs. This is also consistent with modern advanced mathematics. Let us look at some simple example of functions that you are used to.

```
# let square : int -> int =
                       function x -> x*x  (* the actual definition of the
                                            function square *)
;;

val square : int -> int = <fun>
```

The function name is "square" and has domain and co-domain as integers. This is referred to as the *type* of the function. The definition of the function says that it maps an integer x to `x*x`. Here x is the *formal parameter*. Whenever we want to make use of the function square we must invoke it with an *actual parameter*.

```
#square 4 ;; (* invoking the function with argument 4 *)
- : int = 16

# square 4.0 ;;
This expression has type float but is here used with type int
```

of the RIGHT TYPE.

```
#square ;;
- : int -> int =<fun>
```

The last statement is similar to our previous experience with values. Unfortunately it only returns the type of the function and not the actual definition.

Let us delve some more.

```
# let cube x = x*x*x ;;
val cube : int -> int = <fun>

# cube 3 ;;
- : int = 27
```

So Caml is actually somewhat intelligent ! It is able to conclude from the rather sloppy statement that cube is a function mapping integers to integers. Although Caml has forgiven you for an incomplete definition, your instructor may not and so let us stick to the more rigid format. Shortcuts or terse statements will not be appreciated in programming especially when are starting out.

```
# let quad : int -> int = function
                         x -> x*x*x*x
;;
val quad : int -> int = <fun>

# let quad2 : int -> int = function
                         x -> (square x)*(square x)
;;
val quad2 : int -> int = <fun>

# quad 3 ;;
- : int = 81

# quad2 3 ;;
- : int = 81
```

The two definitions quad and quad2 take a given integer parameter and raise it to the fourth power. The first one is the straightforward definition where as the second one actually makes use of a previous definition square. That it saves some typing (imagine writing a function to raise x to its hundredth power) is a secondary issue. The main advantage is that we can build up more and more complex functions starting from simple functions as building blocks. The more complex function can become a building block for further definitions. This hierarchy of function definitions will be a recurring theme for writing any complex program.

One of the most important issues in this exercise is being consistent about the types of the functions we are trying to manipulate. In the previous example, function square has type `int -> int`, so we can use the integer multiplication `*` to multiply the resulting integers. Notice that Caml actually verifies it and concludes that quad2 has the type int -> int consistent with its definition.

Here is another way of computing the fourth power using function composition.

```
# let quad3 : int -> int = function
                           x -> square(square x)
  ;;
val quad3 : int -> int = <fun>

# quad3 3 ;;
- : int = 81
```

We have simply exploited the well known fact that $(x^2)^2 = x^4$. Again Caml could verify the type of the function quad3 from its definition by function composition of square. A critical thing for function composition fog (read as f of g) is that the co-domain of g must be the domain of f. For the function square it is clearly so, namely int.

So far we have been looking at functions of one parameter. There is a natural extension to functions of multiple parameters.

```
# let length : float*float -> float = function
                             (x,y) -> sqrt( x *. x +. y *. y )
  ;;

val length : float * float -> float = <fun>
# length (3.0 , 4.0) ;;
- : float = 5

# let norm3 : float*float*float -> float = function
                         (x,y,z) -> (x *. y) +. z
  ;;

val norm3 : float * float * float -> float = <fun>
# norm3 (1.1 , 2.0 , ~-.2.1) ;;
- : float = 0.1
```

Exercise 3 *Write a function* area *for computing the area of a circle whose diameter is given.*

Exercise 4 *Write a function that returns the intersection point of two (non-parallel) straight lines.*

3.1 Functions defined by cases

Until now, we were looking at functions that were relatively smooth in a mathematical sense. However there are functions that we define in terms of several cases of simple smooth functions. For instance, the absolute value is defined mathematically as

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

All programming languages provide mechanisms for condition checking in some form that are self-explanatory. In Caml, the above function can be written as

```
# let absol : float -> float = function
             x -> (if x >= 0.0) then x else (~-. x)
;;
val absol : float -> float = <fun>

# absol 3.2 ;;
- : float = 3.2

# absol ~-.3.2 ;;
- : float = 3.2
```

The basic syntax (format) of a conditional statement in Caml (as in most other languages) is
If $<$ *condition* $>$ then $<$ *action* $>$ else $<$ *action* $>$
The $<$ *action* $>$ itself may be another conditional statement and several levels of nesting is allowed.

```
# let zerorone : int -> string = function
      x -> if (x <2) then if (x >= 0) then if (x = 0) then "zero"
                                                         else "one"
                                        else  "smaller"
               else "larger"
  ;;
val zerorone : int -> string = <fun>

# zerorone 1 ;;
- : string = "one"

# zerorone 5 ;;
- : string = "larger"

# zerorone ~-2 ;;
- : string = "smaller"
```

Long nested conditional statements are not easy to read or verify. However in some situations where the number of possibilities is *finite*, we can use the `match` instruction in the following manner

```
# let imply : bool * bool -> bool = function (x, y) ->
    match (x,y) with (false , false) -> true |
                     (false , true) -> true |
                      (true, true ) -> true |
                      (true , false ) -> false ;;
val exor : bool * bool -> bool = <fun>

# imply (true , false ) ;;
- : bool = false
```

This is much simpler to read, but if the domain is not finite or if all cases are not covered, then Ocaml issues a *warning* that it is not a fully specified (over the entire domain) function

```
# let lower_upper : char -> char = function x -> match x with
                                          'a' -> 'A' |
                                           'b' -> 'B'|
                                           'c' ->  'C' ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
'd'
val lower_upper : char -> char = <fun>
```

When use of nested if-then-else statements is unavoidable, try to indent it properly as it has been done above - the else statements have been written directly below the condition that it is associated with. The thumb rule is that an $< elseaction >$ gets associated with the nearest $< ifstatement >$.

Exercise 5 *Write a Caml function to determine the roots of a quadratic equation. Figure out a way to handle the case when the the roots are not real.*

Exercise 6 *Let d be an integer and m be a string. Write a Caml function that returns true iff d and m form a valid date (assume non-leap year). For example 31 April is not a valid date.*

3.2 Repetitive application of functions

Let us start with a specific example - that of writing a function $fact$ for computing the factorial of a given (positive) integer. Recall that

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ \prod_{i=1}^{n} i & \text{otherwise} \end{cases}$$

The techniques described until now do not let us write a function that requires an unbounded number (not known in advance) of operations. Here, it is the number of times we have to multiply the numbers 1 to n. Since n is itself a parameter of the function, we cannot write it in terms of a fixed number of operations. Notice however that the notation $\prod_{i=1}^{n} i$ itself is a reasonable description of what we have to do. You are perhaps more used to the summation notation $\sum_{i=i}^{n}$. As long as such formulae are regular and nice, we can hope to write them compactly using regular mathematical notations.

However, you may encounter much more complicated situations - say the problem of finding out the sum of the first n prime numbers. Till now, mathematics has not given us a compact closed-form formula to compute the i-th prime number for any i. Lest you think that this is an unreasonable problem, rest assured that it will be one of your assignment problems. A more important fact is that the above problem can be certainly solved by hand for small values of n and given a suitable reward you may even go upto $n = 1000$. Theoretically, there is no reason why you cannot do it for any value of n given enough time and resources. In other words, you have a method that works for all n. So, it is only reasonable that we should be able to write a function for it, given some additional features. This is a central thesis of computer science, i.e., we are able to write functions (programs) for problems that we can solve intuitively.

A very versatile technique for describing functions is *recursive definitions.* This is basically an application of the principle of mathematical induction. Before we deal with the previous problem, let us first look at the relatively simpler problem of computing factorial. We can define the factorial function alternatively as

$$fact(n) = \begin{cases} 0 & \text{if } n = 0 \\ n * fact(n-1) & \text{otherwise} \end{cases}$$

In order to compute factorial of 5, $fact(5)$ needs to know the value of $fact(4)$ which in turn needs $fact(3)$ and so on. Eventually, $fact(0)$ is computed directly as 1. And, then the actual calculations begin in the reverse direction. The entire process can be viewed as having two phases - the unfolding phase and then the calculation phase. These are sometimes referred to as *top-down* and *bottom-up* phases. We will discuss more about these in the near future.

The above recursive definition can be directly written in Caml as

```
# let rec fact : int-> int = function
             n -> if n = 0 then 1
                      else n*(fact (n-1))
```

```
  ;;

val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

The 'let rec' indicates to Caml that the function definition that follows is a recursive definition.

Here is another example involving two variables - power (x,n) that computes x^n for any non-negative integer n.

```
# let rec power : float*int -> float = function
                         (x,n) -> if n = 0 then 1.0
                                     else x*.(power(x, (n-1)))
  ;;

val power : float * int -> float = <fun>

# power (1.0 , 10) ;;
- : float = 1

# power (2.0 , 16) ;;
- : float = 65536
```

Can you write the above function without recursive definition - the way we had written for some fixed powers of x earlier like x^2 and x^4 ?

Until now we have been defining functions in Caml and testing it by applying it on some arbitrary input. We were satisfied if it gave the expected answer for that input. *But how do we know that it will give the correct answer for all possible inputs ?* . A program is useless if it does not produce correct results. There are many critical applications (like in space) where the slightest error could lead to disasters. So how do we check the correctness of a program ? It may not always be possible or practically feasible, especially for large softwares. It also depends on the way the program is written and here lies the advantage of recursive definitions. We can use the principle of Mathematical Induction (MI) directly to prove the correctness of recursively defined functions.

Consider the power function defined above. To use MI, we must define our Induction Assertion $P(n)$ formally and precisely. In this case, it can be stated as:
P(n) : power(x,n) $= x^n$
The goal of MI is to establish that $P(n)$ is true for all $n \geq 0$.
BASE CASE $n = 0$, it is clearly true as $power(x, n) = 1$.
Now we will show that *for all k* $[P(k) \Rightarrow P(k+1)]$. [1] From the Caml function,

$$power(x, (k+1)) = x * power(x, k)$$

$$= x * x^k \text{ from Inductive hypothesis P(k)}$$

$$= x^{k+1}.$$

[1]Recall that there is another form of induction where you must show that [*for all* $k < n$ $P(k)$] $\Rightarrow P(n)$. These forms are logically equivalent and their use depends on the convenience of the proof.

This completes the proof of correctness. Although this proof is very simple, you may come across situations where even the formulation of the Induction Assertion may be very tricky. We shall look at more examples a little later.

Exercise 7 *Use the principle of mathematical induction to prove the following*

- *For all* $n \geq 1$, $x^{2n-1} + y^{2n-1}$ *is divisible by* $x + y$.
- *For all* $n \geq 1$,
$$1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + \frac{1}{\sqrt{4}} \cdots \frac{1}{\sqrt{n}} > 2(\sqrt{n+1} - 1)$$

Exercise 8 *The* greatest-common-divisor *(gcd) of two non-negative integers* m, n *is known to satisfy the identity* $gcd(m, n) = gcd(m, n + m)$. *Prove it.*
Then use it to give a recursive definition of gcd of two (non-negative) integers in Caml.

Exercise 9 *Using the observation that* $(x^k)^2 = x^{2k}$, *give an alternate recursive definition of the function power(x, n) that computes* x^n *and write a corresponding Caml program.*
Hint : If n is odd then n-1 is even !

Exercise 10 *Can you write the following functional definition of power in Caml directly ? Explain what happens and what can you conclude.*

$$power(x, n) = \begin{cases} 1 & if\ n = 0 \\ power(x, (n+1))/x & otherwise \end{cases}$$

4 Some Useful top level directives

The interpreter has very poor editorial features. You may want to use your favourite editor like vi or emacs and load the file into the interpreter using *use* command prefixed with "#". Here is a list of some useful commands (all prefixed with "#").

```
#use <filename> ;; (* the text in the file will be included as if typed
                from within the interpreter *)

#trace <function-name> ;; (* The argument and the result will be displayed
          at each call to the function. It can be a very useful debugging
        tool but it may clutter up your screen if not used discreetly *)

#untrace <function-name> ;; (* stops tracing the function *)

#untrace_all ;; (* stops tracing all functions traced so far *)
```

5 More on recursive definitions

Let us now look at a more complex function, that is primality testing. We want to determine if a given integer is prime (or composite). There is no magic formula for prime but we can exploit the definition of prime. Recall that a number $n > 2$ is prime if and only if it is not divisible by any number other than 1 and itself. So the straightforward algorithm will be to check if there is any number in the set $\{2, 3 \ldots n-1\}$ that divides n. Here division pertains to integer division, that is x divides y if the remainder of y/x is zero. So we can begin by trying to divide n by 2. If it is divisible, then we can pronounce that n is not prime. Otherwise, we try dividing by 3 and so on. If we haven't found an integer that divides n till $n-1$, then we can certify that n is prime.

To write this algorithm as a Caml function, we will first write an intermediate function that checks if an integer x divides n. If so, then it returns false (i.e. n is not prime) else it tries to check if $x+1$ divides n. Let us call this intermediate function trynext. So trynext can be as follows.

```
let rec trynext : int*int -> bool = function
              (i, n) -> if (n mod i = 0) then false (* n is not prime *)
                          else     trynext ((i+1) , n) (* try next number *)
;;

val trynext : int * int -> bool = <fun>
```

This is the most important operative component in our algorithm. Now, we can try writing the primality testing function by invoking this with parameters $(2, n)$.

```
let prime : int -> bool = function n -> if (n =2) then true
                                            else  trynext (2, n) ;;

val prime : int -> bool = <fun>
```

What is the problem with this function ? What happens if n is prime ? There is no stopping criterion in the function trynext if n is prime - that is, if we have tried out all potential divisors, we must declare n to be prime. In absence of that, this function will go on for ever trying out the entire set of integers (until max_int). Hence it is not a valid algorithm !

Exercise 11 *Modify trynext appropriately so that it always terminates and then use it in the function prime.*

Exercise 12 *Along the lines of the previous function power, work out a formal proof of correctness of the function prime.*
Hint : How about the inductive assertion - trynext (j, n) is invoked iff n is not divisible by $\{2, 3 \ldots j-1\}$. *Notice that the inductive assertion is stronger than the end-result that we need.*

In the previous examples of recursive definitions, the function was being invoked at most once in the definition. There is no such restriction in the in the next example. The Fibonacci sequence is defined as follows:

$$Fib_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ Fib_{i-1} + Fib_{i-2} & \text{otherwise} \end{cases}$$

The equivalent Caml function (any sequence is a function from the domain of integers) can be written readily as

```
# let rec fib : int -> int = function
            i -> if i = 0 then 0
                    else if i = 1 then 1
                            else (fib (i-1)) + (fib (i-2))
  ;;
val fib : int -> int = <fun>

# fib 10 ;;
- : int = 55
```

5.1 Cost of computation

If you execute a function using paper and pencil (and your mental prowess), the cost of computation is related to amount of paper you use (space complexity) and time you spend (time complexity). We have ignored the cost of computation because we are not paying to use the computer ! However, you may have experienced that during certain hours of the day, when there are lots of users, the computer does not respond instantaneously. So the computer does get bogged down if there are too many jobs running - that is, it takes more time to service individual requests. If you could isolate your computer (like your PC if you have one), then you can claim to get rid of this problem. So now you expect to get answers to your Caml functions in a blink. Try fib 50 - the 50th Fibonacci number using the previous definition.

To see why it is so, let us first study the execution of the factorial function fact that we had defined earlier. The execution profile is basically expanding or unfolding the recursive function calls.

```
fact 5 =  5*(fact(4))
       =  5*(4*(fact(3)))
       =  5*(4*(3*(fact(2))))
       =  5*(4*(3*(2*(fact(1)))))
       =  5*(4*(3*(2*(1*(fact(0))))))
       =  5*(4*(3*(2*(1*1))))
       =  5*(4*(3*(2*1)))
       =  5*(4*(3*2))
       =  5*(4*6)
       =  5*(24)
       =  120
```

It has two distinct phases - one where the recursive definition is being unfolded and a second phase where the actual computation is taking place. For each function call and for each computation (in this case integer multiplication), the computer spends some time say u. For simplicity, we are assuming that all operations take the same time. The actual value of u depends on the speed of the underlying hardware-circuit and differs from machine to machine. In today's technology, u can be as small as 10^{-9} (nano) seconds which may appear to be negligible at first sight. In the above computation, if we were to compute the factorial of 10^9, then it would take about 1 second. This is quite encouraging as we will rarely be computing factorial of numbers larger than 100. You may want to ponder over the following question - does it take you longer to multiply two 20 digit numbers than it takes you to multiply two 4 digit numbers by hand ? Does it have any bearing in computing factorial ?

Exercise 13 *How big (number of digits) is factorial of 100 ?*
You may find the following approximation called Stirling's approximation quite useful, namely,

$$n! \approx \sqrt{2\pi n}\Big(\frac{n}{e}\Big)^n.$$

In the above computation of factorial, we end up multiplying the sequence $1 \cdot 2 \cdot 3 \ldots n$ just the way it is defined but the unfolding phase is an artifact of the recursive definition of factorial. If we were to do it on a piece of paper, we may want to do the multiplications immediately rather than waiting for the second phase without affecting the answer. Namely

```
fact(5) = 5*(fact(4))
        = 20*(fact(3))
        = 60*(fact(2))
        = 120*(fact(1))
        = 120*(fact(0))
        = 120
```

How do we force the computation to proceed like above ?

```
# let rec facti   : int*int -> int = function (x, y) ->
                           if ( y =0 ) then x
                               else   facti( x*y , (y-1))
  ;;
val facti : int * int -> int = <fun>

# let factnew : int -> int = function
               n -> facti (1 , n)
  ;;
val factnew : int -> int = <fun>

# factnew 5 ;;
- : int = 120
```

Convince yourself by tracing the computation profile of the new definition of factorial - that uses the recursive function facti.

Exercise 14 *Write an alternate (equivalent) definition of the function facti that multiplies the numbers in the order $1 \ldots n$ instead of $n \ldots 1$. Prove the correctness of factorial using the new definition. Note that in this case we are computing the factorial of i from the value of factorial of $i-1$.*

The advantage in writing the new definition of factorial is not as drastic as it will be in our next example. Using the previous definition of Fibonacci sequence calculation, try to compute fib 50. Don't wait for ever as you can terminate the computation by typing `^C`. Let us try to analyse what is happening by unfolding the recursion. It will look like a tree branching upside-down. How many operations have to occur for fib 10 ? How about a general formula for fib n ?

Exercise 15 *Verify using induction that*

$$Fib_n = \frac{1}{\sqrt{5}}(\phi^n - \phi'^n)$$

where $\phi' = 1 - \phi = \frac{1}{2}(1 - \sqrt{5})$.
Also prove that

$$Fib_{i+1} = 1 + \sum_{j=1}^{i} Fib_j$$

We claim that the number of function calls for fib n equals fib n. So what is the time taken to compute fib 100 assuming 1 nano-second per operation ? A rough calculation yields about 10,000 years !

But there is some hope. Perhaps we can use the same idea as the previous exercise. That is, given the values of Fib_{i-1} and Fib_{i-2}, we can compute Fib_i by just one addition operation. In other words, to compute Fib_n, we can start writing down $Fib_0, Fib_1, Fib_2 \ldots Fib_{i-2}, Fib_{i-1}, Fib_i \ldots Fib_n$ where

$$Fib_i = Fib_{i-1} + Fib_{i-2}$$

from the inductive definition. In this case, we need $n - 1$ additions which is very reasonable and much smaller than Fib_n itself. A little thought brings forth the realization that we only need to retain the previous two numbers of the Fibonacci sequence in order to compute the next one instead of writing down the entire sequence.

```
 let rec ifib : int*int*int*int -> int = function
      (prev, curr, i , n) -> if i= n then curr (* prev = Fib(i-1), curr = Fib(i) *)
                                  else ifib(curr, prev+curr, i+1, n)
  ;;
val ifib : int * int * int * int -> int = <fun>

#  let fibnew : int -> int = function
    n -> if n = 0 then 0
            else if (n =1) then 1
                    else ifib(0, 1, 1, n)
  ;;
val fibnew : int -> int = <fun>

# fibnew 35 ;;
- : int = 9227465
```

Exercise 16 *Establish the correctness of the function fibnew using MI.*

5.2 A measure of cost

It must be clear that the cost of computation depends critically on the way we write the function. Of course we would like to write it in a way so as to reduce the cost. What is a sound way to measure the cost ? If we manage to actually time it (in seconds or some appropriately chosen unit) then it will vary from computer to computer - a Pentium will be much faster than a 286 and so on even if we use the same function. This does not appear to be fair as we would like to focus on the way the function is written rather than the machine that it executes on. Moreover, the time for executing fib 10 will be less than fib 50 as $50 > 10$. Ideally, we would like to know how much time fib n would take for any n.

This leads us to the notion of growth rate or a functional measure of the running time in terms of some basic primitive operations. The basic predefined operations supported by the hardware (which may not be the same as Caml) forms the set of primitive functions. Formally, the *time complexity* of a program is defined as a function (not to be confused with the program) that gives a rough estimate of the number of basic operations required to execute the program in terms of the *size* of the input.

The *size* of input is related to the length of the representation of the input. It is a common mistake to confuse the size of the input with the *magnitude* of the input. For instance, the input to the factorial function is a number n whose magnitude is n but its length is the number of digits. If $n = 50000$, the length of n is only 5. If we are dealing with decimal representation then the size of n is roughly $\log_{10} n$. Therefore the factorial function takes roughly $10^{(|n|)}$ operations. We will use $|x|$ to denote the size of x. Is it a reasonable measure considering that factorial seemingly takes exponential number of operations ? Do most basic operations take as many ? Addition of two d digit numbers takes about d single digit additions and carry computation using the straightforward method.

Exercise 17 *Show that the conventional method of multiplying of two numbers m and n takes roughly $\log_{10} m \cdot \log_{10} n$ operations where an operation is adding or multiplying two single digits.*

It is strongly recommended that you spend some time analysing the time-complexity (as a function of the input-size) for every program that you write. The field of computer science dealing with designing algorithms places tremendous importance on time complexity of a given problem. There are several techniques that aid in designing faster algorithms. The notion of *faster* is in the functional domain as we are dealing with time-complexity. How can we compare two functions ? Consider $f_1(x) = x^2 + 20$ and $f_2(x) = 20x$. If you plot the graphs, you will find that initially (for small values of $x \geq 0$ as input sizes are always non-negative), $f_1(x) < f_2(x)$ but subsequently, after say $x \geq x_o$ $f_1(x) \geq f_2(x)$. In essence, we are comparing the functions in an asymptotic manner, that is, when $x \to \infty$.

A common notation used for time-complexity is the $O()$ (pronounced *Big-Oh*) that is keeping with the asymptotic growth rate. We say that an algorithm $\mathcal{A}$ has time complexity $O(g(x))$ where $g(x)$ is a function of the input-size x if the actual number of steps $f(x)$ satisfies the following as $x \to \infty$

$$\frac{c \cdot g(x)}{f(x)} \geq 1 \text{ for some constant c.}$$

Intuitively $f(x)$ **never exceeds** $g(x)$ for large values of x modulo a constant factor. This is also referred to as the **worst-case** time-complexity. For an input-size x, we are trying to bound the number of steps in the worst-case scenario, i.e., whatever be the actual input. Using this notation, we can say that the conventional method for multiplying two d digit numbers has time complexity $O(d^2)$.
Remark If $f(x) = x$, then $f(x)$ is $O(x)$ as well as $O(x^2)$ from the definition. In particular, the O notation should not be used in the sense of equality but is closer in spirit to inequality ($\leq$) that gives an *upper-bound.*

What is a desirable time complexity when we are designing algorithms ?. The conventional wisdom over the past few decades broadly classifies an exponential time-complexity (something like 2^x) to be impractical and a low degree polynomial time-complexity (say x^3) as acceptable. The reason for this distinction is self-explanatory as the exponential growth rate is too high for the algorithm to terminate in one's lifetime even for modest input sizes as 50. The first definition of the Fibonacci sequence has an exponential time-complexity whereas the subsequent definition is only a linear (degree 1) polynomial.

Obviously, the faster your algorithm is, the better. But you cannot make an algorithm as fast as we want even if you are the cleverest person on earth. There are fundamental limitations to how fast an algorithm can be that is dependent on the problem. These are called *lower-bounds* and this topic is quite deep and outside the scope of this course.

5.3 An example - primality testing

Let us analyse the Caml function for primality testing at the beginning of the section. The Caml function prime (n) calls trynext (2, n). So, we must analyse trynext (2, n) or in a more general situation - trynext (x, y). The function trynext is supposed to determine if there is an integer t, $x \leq t < y$ that divides y. The function trynext uses the in-built Caml function "mod" to check if some integer t divides y. Let us assume that "mod" (more generally all arithmetic function) takes one operation. We will also assume that the condition checking takes one operation. That makes a total of 2 operations for every recursive call.

What is the worst-case (maximising the number of operations) behaviour of trynext (x, y) ? Clearly, it is the maximum number of t for which the recursive calls are made. This in turn depends on the stopping criterion that we ommitted, but it is pointless to try any t greater than y. This yields at most $2(y-x)$ operations. Therefore, prime(2, n) takes at most $2(n-2)$ operations. The input size of n, that is $|n| = \lceil \log_{10} n \rceil$. Therefore, the number of operations as a function of $|n|$ is less than $2(10^{|n|} - 2)$. In the Big-Oh notation we can write $O(\exp(|n|))$.

This is not good news because it is an exponential algorithm. We can do somewhat better by changing the stopping criterion (How ?). However, it remains an open problem to design a polynomial time-complexity algorithm (without using probabilistic analysis).

Exercise 18 *We made an assumption that the mod function takes one operation. Recall that integer division (including remainder) that we do by hand is proportional to the number of the digits in the divisor and the dividend. Reanalyse the primality algorithm in light of this.*
Hint: Show that the long division takes $O(|divisor|) \cdot |dividend|)$ operations.

5.4 A special kind of recursive definitions

Most of the recursive functions that we developed in this section have the following form.

$$f(n) = \begin{cases} \text{direct evaluation} & \text{if base case} \\ \text{f(k)} & \text{otherwise, where } k < n \end{cases}$$

More specifically, the recursive call $f(k)$ does not involve any operations involving $f(k)$ like $g(k) \odot f(k)$ where $\odot$ is some operation and $g(k)$ is a function of k. Recall that for factorial $g(k) = n$ and $\odot = \times$. Consequently, there is no need for the *computation phase* as the answer is available immediately after the terminating condition. In other words, if you were to work it out using a pencil and paper, you only need to *remember* the last recursive call - there is no need to work backwards. Additionally, if you are armed with an eraser, you may even cut down the cost of paper drastically. You will use the paper to remember only the last recursive call, calculate the *new* parameters and overwrite the *old* parameters. Here over-writing implies erasing and writing.

If recursive function definition has the above simple form, then it is referred to as *tail recursion.* As we have seen in this section, it is possible to rewrite a more general recursive function as tail-recursion like the functions for factorial and Fibonacci sequence. Whether it is possible in a more general setting is something that we won't be able to discuss formally as it requires some knowledge of general recurrences. But from now, we will try to write *tail recursive* definitions for functions for the obvious benifits.

The work involved in calculating the *new* parameters from the *old* parameters is often referred to as an *iteration.* Using a pencil and paper we perform the iterations until a certain terminating condition is satisfied. At this point one of the parameters gives us the final answer. So, an alternative scheme for doing the same computation is to think in terms of iterations directly instead of tail-recursive definitions. Most computer languages support constructs for writing iterations (also called *loops*). We shall soon see that iterative style often leads to more efficient excution in computers.

In the beginning, it is advisable to work out the iterative scheme starting from a tail-recursive definition. It is easier to verify correctness and analyse time-complexity of a function expressed in terms of tail-recursion because we can use mathematical induction more directly and naturally.

We have ignored the issue of *space-complexity* so far which is related to the memory in the computer. Memory is used for storage which is critical for the computation to proceed - recall how recursive functions are unfolded and then evaluated. Even for tail-recursive definitions, we must *remember* the previous parameters. The memory in the computer is reusable, otherwise we will be running out of memory very quickly. This is analogous to doing computation on paper without an eraser. We have already seen that tail-recursion can be quite parsimonious in its use of storage although it is not clear how it is done inside Caml functions. This will be our next topic where we will discuss a style of programming where we will be explicitly managing storage elements inside the program. This will enable us to exercise for control in our use of memory.

Exercise 19 *Write tail recursive Caml functions for the following problems*

(i) Summing n terms of a series where the i-th term is given by $(-1)^i \cdot \frac{2^i}{i!}$.

(ii) Summing the first n prime numbers.

(iii) Number of digits in a given integer. For example 84653 has 5 digits.

Exercise 20 *Write a tail recursive definition for the method outlined in Exercise 9 and analyse the time-complexity.*

6 Functions as parameters

Functions do not have any special status in ocaml (or for that matter any functional languages) and can be freely passed as parameters to other functions. The description of the function is the function as opposed to the evaluation of the function for a given parameter value. For example the description of the function that squares its input is

```
function x -> x*x ;;
```

The following function is the implementation of the bisection method for computing a (real) root of a given function. This takes as parameters an initial interval that contains exactly one root, the function for which the root is being computed and the precision to which it needs to be computed.

```
let rec bisection  = function (* passing function f as a parameter *)
       (low, high, f , eps) -> if f ((low +. high)/. 2.0) > eps then
                                  bisection (low, (low +. high)/. 2.0, f, eps)
```

```
                           else if (f ((low +. high)/. 2.0) +. eps < 0.0)
                              then bisection((low +. high)/. 2.0, high, f, eps)
                                 else (low +. high)/. 2.0  ;;

# bisection (0.0 , 8.0 , (function x -> x*.x -. 8.0 ) , 0.001) ;;
- : float = 2.82836914062
```

Exercise 21 *Write an OCAML function for computing the* numerical *derivative of a given function. The* numerical *derivative of a function f at a point x is defined as* $\frac{f(x+dx)-f(x)}{dx}$ *for a very small (pre-defined) value dx.*

Exercise 22 *Write an OCAML function* compose *that takes two functions f and g as input and returns the composition $g(f)$.*

7 Scope

When you type ocaml, we enter the *top level* of the interpreter. Any assignment of values or functional description to identifiers are valid in this environment. Only the latest assignment is relevant. As each function is executed, the parameters of the function is relevant only within the function and are distinct from the top-level.

let x = 5; binds x to 5 but is distinct from the parameter x in let sqr = function x -> x*x ;; This is important especially in the context of recursive function definitions where each level of function call creates a new set of identifiers valid only within that level and distinct from the other levels. The interpreter keeps track of these and hence the the programmer is freed from the burden of finding new names for every level of function calls. It is a good practice to define the identifiers within the function where it is required rather than *globally* (valid within the entire top level). For instance we can define pi only within the area function using the construct **let** < binding>**in**

```
# let area : float -> float
       = function r -> let pi = 22.0 /. 7.0  in pi *. r*. r ;;
val area : float -> float = <fun>
```

In the top-level pi is undefined

```
# pi ;;
Unbound value pi
```

We can have several bindings in an environment using **let** binding1 **and** binding2 **and** .. **in**.

```
# let area : float -> float
       = function r -> let pi = 22.0 /. 7.0  and
                               sqr : float  -> float = function x -> x*. x
                           in pi *. sqr r ;;
val area : float -> float = <fun>
# area 4.0 ;;
- : float = 50.2857142857
```

8 Processing Lists: variable length input

Till now we have defined functions with a fixed number of input - in many occasions that is not known. For instance what if we try to find out the average marks in a class of students and we don't want to write a separate program for each class size. OCAML has a data type called lists to handle such a scenario.

A list is an ordered set of elements of the same type. A list is empty if it has no elements which is denoted by []. Otherwise a list can be partitioned into the *head* which is the first element and the remaining called the *tail*. Note that a tail may be empty for a one element list. The *head* is attached to the tail using the *cons* operator denoted by "::". A list in OCAML is represented by its ordered set of elements separated by ";".

```
# let l = [ "is" ; "was" ; "will" ; "may"] ;;
val l : string list = ["is"; "was"; "will"; "may"]
# let l2 = [ 3 ; 45 ; 97 ; ~-4 ; 100 ] ;;
val l2 : int list = [3; 45; 97; -4; 100]
# 11:: l2 ;;
- : int list = [11; 3; 45; 97; -4; 100]
# 23:: [] ;;
- : int list = [23]
```

The base type of the elements in a list defines the type of lists - integer list, string list , float list etc. One can define a list of lists. One can apply the same function to all elements of the list by using the (higher order) function List.map. For instance

```
# List.map (function x -> x*x) [ 0; 2 ; 4; 9 ] ;;
- : int list = [0; 4; 16; 81]
```

Exercise 23 *What is the domain and range of List.map ?*

The following program generates all the sub-lists of a given list by a clever use of the List.map function.

```
let rec powerset l = match l with
           []  ->   [[]] |
         hd::tail -> let cons a b = a::b in
               (powerset tail) @ (List.map (cons hd) (powerset tail)) ;;
```

The most common way of processing the lists is to process it recursively beginning from the head. For example if we are interested in reversing a given list, then we can *concatenate* the head of the list at the end of the (recursively) reversed tail. The concatenation operation joins two given lists in the order that it is given.

```
# [  3; 4 ] @[ 4] ;; (* @ is the concatenation operator *)
- : int list = [3; 4; 4]
```

The following function reverses any given list.

```
# let revlist = function l -> if l = [] then [] else
                  let rec revlist1 = function head::tail -> if tail = [] then
                          [head] else (revlist1 tail) @ [head] in
                revlist1 l ;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val revlist : 'a list -> 'a list = <fun>
```

The first condition was necessary to take care of empty list but the function revlist1 cannot handle empty-list for which OCAML has issued the warning. To handle such cases we can use the "match" statement that is able to handle all possible structures of the lists.

```
# let rec revlist l = match l with [] -> []
                          | head::tail -> (revlist tail) @ [head] ;;
val revlist : 'a list -> 'a list = <fun>
# revlist [2 ; 3; 4; 5] ;;
- : int list = [5; 4; 3; 2]
```

Exercise 24 *Write an OCAML function to find the minimum element in a list*

Exercise 25 *Write an OCAML function to arrange a list in ascending order.*

Exercise 26 *Write an OCAML function to implement List.map function defined earlier using matching.*

9 Declaring Types

Quite frequently, you may have felt the limitation of predefined types in OCAML. For instance there is no predefined type rational (which can be viewed as a tuple of numerator and denominator) or a polynomial. OCAML provides very powerful mechanisms for defining new types. We will look at some very simple examples.

```
# type rational = {num: int ; den : int} ;;
type rational = { num : int; den : int; }
```

By this we have defined rational to be a new type that has two components num and den both of type integer. To declare a new rational number we must define the two components.

```
#let r = {num = 5 ; den = 10 } ;;
val r : rational = {num = 5; den = 10}
# r.num ;;
- : int = 5
```

Once you have a new type you will like to define functions on these. One of the common requirements of a rational number is that the numerator and denominator should be co-primes (i.e. no common factors). For that we can define a function reduce that will remove common factors.

```
#let reduce : rational -> rational = function r ->
    let rec gcd : int*int -> int = function (a,b) ->
         if a > b then gcd ( b, a)
            else if (b mod a = 0) then a
                 else gcd ( a, b-a )    in
      {num = r.num / gcd(r.num , r.den) ; den = r.den/gcd(r.num , r.den) } ;;
# reduce { num = 5 ; den = 10 } ;;
- : rational = {num = 1; den = 2}
```

Exercise 27 *Prove the correctness of the gcd (greatest common divisor) function - popularly known as Euclid's algorithm.*
Hint: You may have to use recursion on two variables, i.e. let it be true for all numerator $\leq$ denominator $\leq n$. Then prove for all numerator $\leq$ denominator $\leq n + 1$.

Exercise 28 *Define functions that adds, multiplies and divides two rational numbers.*

Exercise 29 *Can you make the gcd algorithm more efficient ?*

You can now build more complex data types for instance, you may keep a list of all students with various attributes like name, address, ranks etc.

```
#type stdrecord = { name : string ; address : string ; marks : float } ;;

type stdrecord = { name : string; address : string; marks : float; }

# [ {name = "x1" ; address = "y1" ; marks = 50.0 } ; {  name = "x2" ; address =
         "y2" ; marks = 65.3 } ] ;;
- : stdrecord list =
[{name = "x1"; address = "y1"; marks = 50.};
 {name = "x2"; address = "y2"; marks = 65.3}]
```

Suppose we want to define a new type that has finite (small) number of values, say gender which takes two values.

```
# type gender = Male | Female ;; (*must begin with upper-case unlike identifier
                                   names*)
type gender = Male | Female
```

9.1 Variant types

If you want to mix up types or define a superset of previously defined types, that is done as follows

```
# type realnumber = Int of int | Float of float | Rational of rational ;;
type realnumber = Int of int | Float of float | Rational of rational
```

The following example shows how to generalise the multiply function to realnumber (it is somewhat incomplete - try to complete all cases).

```
#let ratmult x y = { num = x.num * y.num ; den = x.den * y.den } ;;

#type number = Int of int | Float of float | Rational of rational ;;

#let mulnumber x y = match ( x, y) with
        (Int a , Int b) -> Int ( a * b ) |
        ( Float a , Float b) -> Float ( a *. b ) |
        (Rational a , Rational b) -> Rational ( ratmult a b ) |
        (Int a , Float b) -> Float ( (float a ) *. b );;
```

9.2 Defining new types using recursion

The predefined type list can be thought of like the first element followed by the remaining list. `type 'a lst = Empty | Element of 'a* 'a lst ;; (* our own list type *)` Example of such a list is `Element(2 , Element ( 4 , Element (1, Empty))) ;;` We can define functions on this type exactly the same way as we did for the predefined lists.

```
let rec listlen lst1 = match lst1 with (* computes length of list*)
             Empty -> 0 |
            Element(a, b) -> 1+ listlen(b) ;;
```

More complex functions like Insertion sort can also written in the same way.

```
let rec insort lst = match lst with
    Empty -> Empty |
   Element(head, tail) -> let rec insert x ylst =
                 match ylst with
              Empty -> Element(x, Empty) |
            Element (head1, tail1) -> if x <= head1 then Element(x, ylst)
                    else Element(head1 , (insert x tail1))
           in insert head (insort tail) ;;
```

10 Exception

Most modern programming languages provide useful clues for handling unexpected execution - a classic example being divide by zero. There are essentially run-time errors and we can define exceptions that are tailor made for our applications.

```
exception Empty_list ;; (* defining exception *)

let rec maxlist1 = function lst ->
match lst with
          [] -> raise Empty_list |
          head :: [] -> head |
          head :: tail -> let r = maxlist1 tail in
                   if head >= r then head else r ;;
```

One very common use of exception can be for giving precise error messages by using a *parameterized* exception.

```
 exception Input_type of string ;
let rec fact n = match (n <0) with
     true -> raise (Input_type "factorial not defined for negative") |
      false -> if n = 0 then 1 else n*fact(n-1) ;;
```

The construct `try ..with` can be used to exit a nested sequence of function calls in a controlled manner.

```
let pastestring = function a -> ( maxlist1 a), "valid" ;;

(* if pastestring throws an exception, it is handled in the main function*)

let maxlist lst = let pastestring = function a -> ( maxlist1 a), "valid" in
                      try pastestring lst with
     Empty_list ->  (0 , "not valid") ;;
```

11 Arrays and mutable structures

Arrays in Ocaml are defined as

```
# let a = [| 21 ; 3 ; 45 ; 1 |] ;;
val a : int array = [|21; 3; 45; 1|]
```

To access the element corresponding to i-th index, we use `a.(i)`. Note that the index begins from 0 (and not 1). Each array location behaves like a memory location, i.e., it can be modified by writing.

```
# a.(1) <- 5 ;; (* assignment operator *)
- : unit = ()
# a.(1) ;;
- : int = 5
```

Unlike the "let" statement, no new location is created but the contents of the existing location is modified. Also note that assignment doesn't return any value - it is unit.

11.1 References

We can declare references (pointers) to variables of a certain type

```
# let x = ref 5 ;;
val x : int ref = {contents = 5}
```

The *contents* of x can be referred as

```
# !x ;;
- : int = 5
```

The contents of x can be modified with an assignment instruction.

```
# x := 10 ;;
- : unit = ()
# ! x ;;
- : int = 10
# x := !x + 2 ;;
- : unit = ()
# !x ;;
- : int = 12
```

Note that the modification doesn't return any value (it is unit). Although the reference behaves somewhat like a pointer but you are not allowed to do any arithmetic on the references - i.e., `x +1` is undefined (beware C programmers). You can however assign references as follows.

```
# let y = x ;;
val y : int ref = {contents = 12}
# !y ;; (* x and y refer to the same data *)
- : int = 12
```

Here is an example that will distinguish a variable from the binding defined by `let`

```
# let x = 2 ;;
val x : int = 2
# let y = ref x;;
val y : int ref = {contents = 2}
# let x = x + 1;; (* which x does it change ? *)
val x : int = 3
# !y ;;
```

What is the value returned - is it 2 or 3 ?

12 Objects and classes

Object-oriented Programming is now one of the most widely accepted style of programming that views the task of programming as interactions between different entities called *objects*. An object has various attributes and associated *methods* with which it can interact with other objects and participate in some computational tasks. The general characterization of an object is described as a *class*. For example, the class rational can be described as

```
# class rational (n : int)  ( d :int) =

 object

  val mutable num = n
  val mutable den = d
  method numerator =num
  method denominator = den
  method frac = float (num mod den) /. (float den) (* fractional part *)
  method reduce = let d = gcd1 num den in  (* gcd1 comoputes greatest common
       divisor  and the reduce makes num and den coprime *)
                       num <- num/d ; den <- den/d ;
  end;;
class rational :
  int ->
  int ->
  object
    val mutable den : int
    val mutable num : int
    method denominator :int
    method frac : float
    method numerator : int
    method reduce : unit
  end
```

To create an instance of an object we use `new` which creates a *reference* to the object.

```
# let a = new rational 8 6 ;;
val a : rational = <obj>
# a#numerator ;;
- : int = 8
# a#denominator ;;
- : int = 6

# a#frac ;;
- : float = 0.333333333333333315
# a#reduce ;; (* numerator and denominator are now co-prime *)
- : unit = ()
# a#numerator ;;
- : int = 4
# a#denominator ;;
- : int = 3
```

13 Imperative constructs

Ocaml supports the standard imperative features like **for** statements.

```
# let int_array  n =  (* creates an array with contents 1, 2, ...n *)
  let res = Array.create n 0 in
  for i = 0 to n-1 do
      res.(i) <- i + 1
    done ;
  res ;;
          val int_array : int -> int array = <fun>
# int_array 10 ;;
- : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

The familiar iterative version of computing sum of n numbers can be written using the while construct

```
let itersum n = let sum = ref 0 and i = ref 0 in
             while !i <= n do
                sum := !i + !sum;
                 i := !i +1 ;
             done ;
           !sum ;;
```

To initialize an m by n two dimensional array in Ocaml, you can use the following program that uses `Array.init` instead of `Array.create` that seems to be inherently 1 dimensional. There is a natural generalization to any arbitrary dimensional array.

```
# let b = Array.init 4 (function i -> i) ;;
val b : int array = [|0; 1; 2; 3|]

#let a = Array.init 3 (function i -> Array.create 2 0) ;;
val a : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]

# a.(0).(1) <- 1 ;;
- : unit = ()
# a ;;
- : int array array = [|[|0; 1|]; [|0; 0|]; [|0; 0|]|]
```

In general, `Array.init n f` returns a fresh array of length n, with element number i initialized to the result of f i. In other words, `Array.init n f` tabulates the results of f applied to the integers 0 to n-1.

The following function swaps two elements of an array (and returns the modified array).

```
let swap i j arr2 =  let y = arr2.(j) in ( arr2.(j) <- arr2.(i) ;
 arr2.(i) <- y ) ; arr2 ;;  (* exchanging elements from index i and j *)
```

The following is an implementation of insertion sort on arrays.

```
let rec insortarr arr i j = (* insertion sort from index i to j *)
 let rec insert k j arr1 =
   if k = j then arr1
    else if arr1.(k) <= arr1.(k+1) then arr1
           else insert (k+1) j ( swap k (k+1) arr1) in
    if ( i = j) then arr else
          insert i  j (insortarr arr (i+1) j ) ;;
```

A more complex example is that of enumerating permutations of numbers $12, \ldots n$.

```
let swap1 i j arr1 = let y = arr1.(j) in (arr1.(j) <- arr1.(i) ; arr1.(i)
 <- y );;

 let print arr = for k =0 to (Array.length arr -1) do print_int arr.(k) done;;

 let rec perm arr i lst = if ( Array.length arr ) = i then
          (* print arr ; print_char '\n' ;*)
          lst := [Array.copy arr]@(!lst)

                  (* What happens if we simply append arr to lst without
                         creating a copy ? *)

       else
         for j = i to ( (Array.length arr) - 1) do
             swap1 i j arr ;
            perm  arr (i+1) lst;
            swap1 i j arr ;
             done ;;

 let permute arr  =  let lst = ref [] in perm arr 0 lst ;
                         !lst ;;
```

The function permute returns a list (lst) of all permutations. To save space, we can instead print out the permutation as soon as it is generated (currently commented).

```
# permute (int_array 3 ) ;;
- : int array list =
[[|3; 1; 2|]; [|3; 2; 1|]; [|2; 3; 1|]; [|2; 1; 3|]; [|1; 3; 2|];
 [|1; 2; 3|]]
```

Exercise 30 *(i) Write a program to generate permutations of n objects when all of them are not distinct.*
(ii) Enumerate all the permutations using the following strategy -
After generating all possible permutations of n-1 objects, insert the n-th object in each of the n gaps induced by the n-1 objects.
(iii) Write a program to generate all choices of k objects from $n > k$ distinct objects. Like the permutation program, use an extra array of size k to keep track of the current choices.

14 Input/Output and Files

One of the most important imperative features of any programming language is its ability to support input and output instructions from a file. A program that deals with large amount of data mst have the ability to obtain its input froma file and print its answer to a file. The default input and output, referred to as `std_in` and `std_out` is set to the keyboard input and monitor output. Reading from `std_in` and Writing to `std_out` ar performed using the following self-explanatory functions -

```
read_int read_float read_line (*for reading integers, floats and reading
an entire line of characters *)
print_char print_int print_float print_string (* for printing character,
integer, floats and strings *)
```

Here is an example of reading an array and printing the elements.

```
 let readwrite len = let arr = Array.create len 0 in
    for k =0 to (Array.length arr -1) do arr.(k) <- read_int() done;
    for k =0 to (Array.length arr -1) do print_int arr.(k); print_char ',' done ;;
```

One must be careful that the input is provided as one integer per line - separating them by any other character leads to an exception that the input is not of proper type. Therefore for many applications it is advisable that we deal only with character inputs and perform our own conversions (to integer or floats). When we view the file as a file of characters, it is called a character file. We must understand the basic methodology of handling files and in particular how Ocaml views files.

To read from a file, a file must be first *opened* - this is required for some bookkeeping operation by the operating system since all files cannot simultaneously reside on the main memory and preparation is required to access the contents of a file. After completion of input it is customary to close it. The files are read sequentially, but we can have several access points simultaneously called *channels*[2] in Ocaml.

Likewise a file must be opened for writing (the contents are cleaned up, so one must be careful about opening for writing) and once writing is over us should be closed for protection. The open and close functions are as follows.

```
#let stream1 = open_in "readfile" in (* open_in returns a stream *)
           input_char stream1 ; (*input_char returns the present character*)
  close_in stream1 ;; (* in a stream and then adjusts the position of the stream
                    to the next character in the file*)



- : char = 't'
```

The function `close_in` closes a **channel** (not the filename as there may be more than one channel).

In the following example we write two strings in the file `outfile` (including two special end-of-line characters)

```
# let stream2 = open_out "outfile" in
     output_string stream2 "fdtfdfukwgdwhdg\nqgfdgqfdyfqdf\n";
     close_out stream2;;
- : unit = ()
```

Other useful functions are `input_line` (for reading a line) and `output_char` (for writing a single character). To detect end of file (while reading), there is a predefined exception `End-of-file`.

[2]A channel is considered to be a special type with distinctions between in-channel and out-channel

```
let read_print f = let chan1 = open_in f in  (* reading till end of file*)
    try
       while (1=1) do
       print_char ( input_char chan1 )
       done
    with End_of_file -> close_in chan1 ;;
```

Exercise 31 *Given a character file F1, write a function that copies the contents to another file F2, such that consecutive blank spaces are compressed into a single space.*