Functional Programming for Imperative Programmers

R. Sekar

This document introduces functional programming for those that are used to imperative languages, but are trying to come to terms with recursion and other techniques used in functional programming. We use OCaml as the primary language, and we assume that the reader has been introduced to its basic syntax and features. The goal of this document is to help these programmers get more comfortable with functional programming techniques, while addressing the following topics:

- Understanding and visualizing recursion
- Iteration as a special case of recursion
  - Imperative programming in functional languages by "threading" state through functions
- Higher-order functions and iteration
- Writing efficient functional code
- Functional programming in imperative languages

1 Simple Vs Mutual Recursion

Recursion can be simple recursion, which means that a function f is recursive with itself. In other words, f is defined in terms of f and $h_1, \ldots, h_n$, where $h_1, \ldots, h_n$ have *no* dependence on f. In OCaml, such a function f can be defined using a single `let rec`. The factorial function defined below is one such example:

```
let rec fact n =
       if n=0
          then 1
       else n * (fact (n-1))
```

In this case, `fact` is defined in terms of itself and two other functions, `if` and the subtraction function on integers.

As a second example, consider the Fibonacci function, where each invocation of `fib` makes two recursive invocations of `fib`.

```
let rec fib n =
      if n=0
         then 0
      else if n=1
         then 1
      else (fib (n-1)) + (fib (n-2))
```

In a more complex case of recursion, functions can be mutually recursive, i.e., f is defined in terms of another function g, which is in turn defined in terms of f. A set of mutually recursive functions are defined in OCAML using a single `let rec`, with a connective `and` separating these function definitions.

```
let rec
    odd x =
       if x = 0
          then false
       else (even (x-1))
and
    even x =
       if x = 0
          then true
       else (odd (x-1))
```

A more realistic example is one that involves searching an n-ary tree:

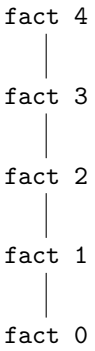

Figure 1: Recursive invocations of `fact`

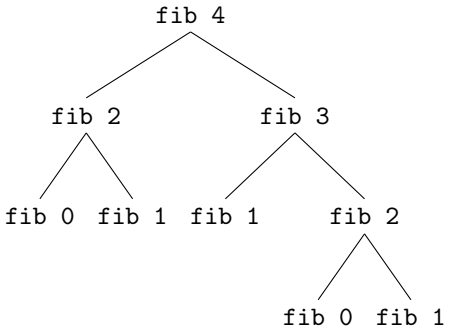

Figure 2: Recursive invocations of `fib`

```
type naryTree =
       Leaf of int
     | Node of naryTree list
```

Thus, an internal node of an n-ary tree has an arbitrary number of subtrees, and these subtrees are stored in a list. With this definition, we can define a search function as follows:

```
let rec
    search t v = match t with
      |  Leaf v1 -> v = v1
      |  Node ls -> searchList ls v
and
    searchList l v = match l with
      |  [] -> false
      |  l1::ls -> (search l1 v) || (searchList ls v)
```

2 Bottom-up vs Top-down Recursive Computation

We can characterize a recursive function as top-down or bottom-up, based on whether the bulk of computation happens before or after the recursive call. For instance, in the case of the factorial example above, the bulk of computation happens after the recursive call: a simple decrement operation on the argument is performed before the recursive call, while a more substantive multiplication operation takes place afterwards.

To understand the term "bottom-up" vs "top-down" recursion, imagine the *activation tree* of recursive calls unfolding. Figures 1 and 2 show these trees for the calls `fact 4` and `fib 4` respectively. When `fact 0` returns, `fact 1` performs a multiplication and then returns to its caller, `fact 2`, which in turn performs another multiplication and returns to its caller, and so on. Thus, not much computation happens in top-down phase when the tree of activations is being constructed from top to bottom. Instead, it happens as the activation tree unwinds from the leaf to root, i.e. in the bottom-up phase. A similar observation can be made about `fib`, i.e., the "interesting" part of the computation happens as the recursion unwinds from leaf-to-root.

Now that we have discussed a couple of examples where most of the action occurs after the return of recursive calls, we can ask what "top-down" computation looks like. With top-down computation, *nontrivial part of the computation occurs before a recursive call,* and, moreover, *the results of this computation are passed into the recursive invocation*[1] In other words, partial results computed during a node's invocation flow down the activation tree to its descendants. The function itself needs to take additional parameters through which these partial results can be passed into a recursive invocation. We will illustrate this using an alternative version of the factorial function. (Note that single quotes are valid within identifiers in OCaml — so `fact'` is a valid function name.)

```
let rec fact2 n p =
       if n=0
```

[1]Without the second requirement, the computation could be performed after the recursive call returns, and hence it would be an instance of bottom-up computation.

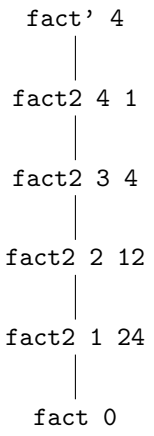

Figure 3: Recursive invocations of `fact2`

```
        then p
      else fact2 (n-1) (p*n)
let fact' n = fact2 n 1
```

Note that `fact2` computes the product operation as the recursion unfolds: unlike `fact n` which postpones the multiplication by n until `fact (n-1)` returns, `fact2` computes the product involving n before making a recursive call. The tree of activations looks as shown in Figure 3: This version of factorial function, which computes and passes partial results into recursive calls, is said to follow an *accumulating style.* (Partial results are accumulated and passed down.) For some problems, accumulating style can yield faster implementations, as we discuss later on.

3 Tail Recursion

In general, "top-down" recursion can involve nontrivial computation before *and* after a recursive invocation, i.e., computation occurs while the activation tree grows from the top to bottom, and again as the recursion unfolds bottom-up[2]. However, there are those *extreme* cases where *no computation is performed in the bottom-up phase.* This is true of the `fact2` function above: the very last operation performed by an invocation of `fact2` is to make a recursive call. When the call returns, the results are immediately passed on up the activation tree.

A recursive function f such that each call of f results in at most one recursive call, and moreover, this call is the very last operation in the body of f, is said to be *tail-recursive.* For instance, `fact2` is tail-recursive. In contrast, `fact` needs to perform a multiplication operation after the recursive call returns, and hence `fact` is not tail-recursive.

Since a tail-recursive function f does not access any of variables (or parameters) after the recursive call returns, there is no need to preserve the environment or the activation record after the recursive call. In other words, the callee can make use of the same activation record as the caller! Consequently, functions such as `fact2 n p` can be computed while using just a single activation record, as opposed to the `n` activation records needed to compute `fact n`.

For tail-recursive functions, since the recursive invocation is to the same function and uses the same environment and store, its effect can be achieved by simply jumping back to the beginning of the function, rather than making a call. In other words, tail recursion can be replaced by iteration. Such an optimization, called *tail-recursion elimination*, plays an important role is functional languages. Compilers for imperative languages have also begun to incorporate this optimization.

4 Iteration

Based on the above discussion, we can say that

- *iteration is nothing but a special case of recursion* where the recursive call appears as the very last step in a function body.

[2]This contrasts with our definition of bottom-up recursion, where all nontrivial computations occur only in the bottom-up phase.

- *iteration corresponds to an extreme case of top-down recursion* where no computation occurs in the bottom-up phase.

Now that we have established that recursion is more general than iteration, let us try to define a function `repeat` that captures the behavior of the familiar repeat–until loop in imperative languages. Recall that the behavior of **repeat** S **until** C is to execute S once. At this point, if the condition C holds, we are done. Otherwise, we reexecute the entire repeat–until statement. Let us try to capture this behavior as follows:

```
let rec repeat s c =
     let s1 = s
     in
       if c
          then s1
       else repeat s1 c
```

Are we done? Can we now go back to using iteration to write our programs, rather than recursion that seems less familiar? If you tried this approach, you will very quickly realize it does not work. In fact, `repeat` either terminates with one iteration, or it never terminates at all! So what happened? The problem is that in a functional language, the condition `c` always evaluates to the same value, regardless of how many recursive invocations it has been passed through. As a result, if `c` initially evaluates to `true`, then the function terminates immediately. Otherwise, it loops for ever since `c` remains `false` for ever. This problem does not occur in an imperative language because C is an expression on variables whose values are stored in memory, and these values can change due to the execution of S. In other words, the reason why iteration works in imperative languages is because of the reliance of these languages on side-effects!

Since OCaml is free of side-effects, we need to explicitly pass any "state changes" from one iteration to the next through parameters. In particular, we can (i) make `c` and `s` to be functions that accept an additional parameter that represents this state, and (ii) make `s` produce a new value for this state. Thus, the new version of `repeat` is as follows:

```
let rec repeatt s c st =
     let st' = (s st)
     in
       if (c st')
          then st'
       else repeatt s c st'
```

Note how the state `st` that captures the "effect of executing" `s` gets passed in from `repeatt` to `s`, and the resulting new state routed to `c` and then the recursive invocation of `repeatt`. This manner of passing the state through the functions is called *threading*. (This is why this "threaded" version of repeat function has been named `repeatt`.)

Now, consider the following iterative program, which computes $\sum_{i=1}^{n} i$:

```
i = n; v = 0;
repeat
  v = v + i;
  i = i - 1;
until (i == 0)
```

We can easily express this in OCaml:

```
type state = {i: int; v: int}
let s st =
    let v' = st.v + st.i in
    let i' = st.i - 1 in
    {v = v'; i = i'}
and c st = (st.i = 0)
in
repeatt s c {i=n; v=0}
```

The function `c` captures the "until" condition, while the function `s` captures the body of the repeat/until loop. Other than the fact that we needed to define a record type in order to conveniently represent the state, the code in OCaml looks very close to the imperative version.

```
fac = 1;
fsum = 0;
i = 1;
repeat
  fac = fac * i;
  fsum = fsum + fac;
  i = i + 1;
until (i > n)
```

Figure 4: `facsum` in Pascal

```
type state = {fac:int; fsum:int; i:int}
let c st = (st.i > n)
and s st =
   let fac' = st.fac * st.i in
   let fsum' = st.fsum + fac' in
   let i' = st.i + 1 in
     {fac = fac'; fsum = fsum'; i = i'}
in
 (repeatt s c {fac=1; fsum=0; i=1}).fsum
```

Figure 5: `facsum` in OCaml using iteration

```
let rec facsum1 fac fsum i n =
   if i > n
      then fsum
   else
      let fac' = fac * i in
      let fsum' = fsum + fac' in
      let i' = i + 1 in
        facsum1 fac' fsum' i' n
let fsum n = facsum1 1 0 1 n
```

Figure 6: `facsum` in OCaml using recursion

It seems inconvenient to have to think through exactly what state needs to be threaded through the loop: it seems so convenient if "every thing" is threaded through, which is what happens in imperative languages. In general, though, being explicit about changes, instead of relying on side-effects, is better in many ways. It leads to programs where the effects are clearer because they are explicit. Such programs are easier to understand, formally reason about, and optimize. Moreover, such programs are easier to reuse *correctly,* as opposed to imperative languages where one often encounters hidden assumptions in software. So, while it is more effort to identify and thread through the relevant effects, it is all worth the trouble.

In a manner similar to repeat/until loops, we can define OCaml functions that capture the behavior of for-loops and while-loops. After this is done, we can use iteration without having to define any recursive functions at all: as in the above example, we will only be defining non-recursive functions such as `c` and `s`.

The purpose of the discussion is to show that iterative constructs can be captured in a straight-forward manner in OCaml. But we certainly do not suggest that it is the best way to solve problems in OCaml. Indeed, there are many problems that can be more succinctly solved using general recursion rather than iteration, which, as we observed earlier, is a special case of recursion. However, for problems where the iterative style happens to be natural, the above discussion illustrates how to use iteration in OCaml. For instance, consider the example `facsum` in Figures 4 to 6 for computing $\sum_{i=1}^{n} i!$.

Figure 4 shows an implementation in an imperative language. A straight-forward translation to OCaml using the `repeat` function is shown in Figure 5. Alternatively, we can express the same algorithm in a form that looks more native to OCaml by taking the following steps:

- Replace the use of `repeat` with an explicitly recursive function `fsum1`
- Pass the components of `st` above as explicit parameters to this function

The new program is shown in Figure 6.

5 Iteration and Higher-Order Functions

The iterators, such as the `repeatt` function defined above, are in fact higher order functions: they take functions such as `c` and `s` as arguments. Much like the repeat-loop construct, which can perform different computations depending on the body of the statement, `repeatt`'s behavior is dictated entirely by the behavior of its function parameters.

Iterative constructs such as `repeatt` operate by emulating an imperative style of programming, where the "state" is threaded through all of the functions. As noted earlier, this is neither natural nor a recommended style of programming in OCaml. However, there are other specialized iterative constructs that do not resort to such an imperative style, and are hence much more natural for OCaml. The most important among these are the following functions for iterating through lists. All these functions take a function f and a list l as parameters:

- `map`, which iterates through every element in l, and applies f to each element,
- `filter`, which iterates through each element x in l, outputting those elements for which fx holds, and
- `reduce`, which iterates through successive elements of l and uses f to combine them.

Their behaviors can be precisely understood from their definitions below. (Most functional languages include built-in implementations of these functions, so it is usually not necessary to define them — in OCaml, they can be accessed as `List.map`, `List.filter`, etc.)

```
let rec map f l =
        match l with
        | [] -> []
        | x::xs -> (f x)::(map f xs);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

As an example, suppose you want to multiply each element of a list `l` by a factor. This can be achieved using `map (fun x -> 2 * x) l`. Note the use of an unnamed function that is constructed on the fly and passed to `map`. This feature, called an *anonymous function, lambda abstraction,* or *lambda expression* used to be available only in functional language, but is beginning to appear in most recently designed or updated languages, including C++, Java, JavaScript, Python, Ruby and so on. Nevertheless, its use is much less common on conventional imperative languages such as Java and C++, but more common in functional languages (e.g., OCaml and Haskell) and languages strongly influenced by the functional style (e.g., Python and JavaScript).

The same example can be written without the use of lambda expressions as `let f x = 2*x in (map f l)`, but the result is a bit less concise. However, if the function definition gets to be complicated, then a named function is preferred, because a meaningful name can make the function much more readable as compared to its code.

The type information shows that `map` is polymorphic: its first argument is any unary function f, and its second argument is a list whose elements have the same type as the input type of f. The output type of `map` is also a list, but the elements have the output type of f.

The function `filter` is very similar to `map` in terms of its structure:

```
let rec filter f l =
        match l with
        | [] -> []
        | x::xs -> if (f x)
            then x::(filter f xs)
          else (filter f xs)
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

As an example, `(filter (fun x -> x mod 2 = 1) l)` will return all of the odd numbers from `l`.

Finally, the higher order function `reduce` can be used to iterate through a list to compute a single value from all the elements:

```
let rec reduce f l i =
        match l with
        | [] -> i
        | x::xs -> reduce f xs (f x i)
val reduce : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Since `reduce` does not return a list, it needs an additional argument `i` that should be returned when it is called on an empty list. Now, `reduce` can be used to iterate through a list `l` and add up the elements as follows:

```
(reduce (fun x y -> x+y) l 0)
```

These functions can operate on a single list, but often, we want to iterate through two lists in parallel, combining them into one. For instance, we may want to add corresponding elements of a list. An additional function called `zip` serves as a useful intermediate in coding such functions:

```
let rec zip l1 l2 =
        match (l1, l2) with
        | (x::xs, y::ys) -> (x, y)::(zip xs ys)
        | ([], []) -> []
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

Note that we have chosen to leave the behavior of `zip` to be undefined when it is given lists of unequal length. An alternative would have been to simply truncate the longer list, but the approach we have taken is better, since it will raise a runtime match failure exception — an exception that can be detected and handled, while the alternative approach will silently suppress what is potentially an erroneous use of zip.

The following example illustrates `zip`:

```
(zip [1;2;3] ['a';'b';'c']) = [(1, 'a');(2, 'b');(3, 'c')]
```

Using `zip`, we can define the following functions very easily:

```
let listsum l1 l2 = map (fun (x, y) -> x+y) (zip l1 l2)
let listprod l1 l2 = map (fun (x, y) -> x*y) (zip l1 l2)
let dotprod l1 l2 = reduce (fun x y -> x+y) (listprod l1 l2) 0
```

Note that `listsum` adds corresponding elements of two lists, while `listprod` multiplies them. If `l1` and `l2` are viewed as two vectors, then `dotprod` computes their vector dot product, also known as the inner product or scalar product. In other words, corresponding elements of two lists are multiplied, and the results added. It is illustrated by the following example:

```
(dotprod [3;7;4] [100; 10; 1]) = 374
```

5.1 List Comprehension

List comprehension provides a way to combine `map` and `filter` functions using the familiar notation used in mathematics for defining sets. For instance, the set of squares of odd natural numbers is given by

$$Y = \{x^2 | x \in \mathbb{N} \wedge odd(x)\}$$

After defining `square` and `odd` as follows:

```
let square x -> x * x
let odd x -> x mod 2 = 1
```

we can define the set Y in OCaml using a notation very similar to the set construction notation above:

```
let y = map square (filter odd x)
```

More generally, the set construction operation in mathematics takes the form

$$Y = \{f(x) | x \in X \wedge p(x)\}$$

where f is a function and p a predicate. This general form is captured in OCaml as:

```
let y = map f (filter p x)
```

Since this way of constructing new lists is concise enough, OCaml does not introduce a special syntax for this purpose. Other languages, such as Haskell and Python do introduce a special notation. The benefit of the special notation is that it can be more readable than the above form that uses `map` and `filter`. For instance, in Python, the above set will be defined using the notation

```
y = [ f(z) for z in x if odd(z)]
```

5.2 Illustrative Examples

To illustrate the concepts discussed so far, consider the implementation of Sieve of Eratosthenes algorithm for computing prime numbers shown in Figure 7 and 8. The first version is a purely functional OCaml version, while the second uses a functional style but is written in Python.

```
let delMult x l = filter (fun y-> y mod x!=0) l
let rec collectPrimes l p = match l with
  | []-> p
  | l1::ls-> collectPrimes (delMult l1 ls) (l1::p)
let rec range l h =
  if h=l then [l] else l::(range (l+1) h)
let sieve n = collectPrimes (range 2 n) []
```

Figure 7: Sieve of Eratosthenes in OCaml

```
def delMult(x, l):
  return [y for y in l if y % x != 0]
def collectPrimes(l, p):
  if l==[]: return p;
  else:
    p.append(l[0]);
    return collectPrimes(delMult(l[0], l), p)
def sieve(n): return collectPrimes(range(2,n),[])
```

Figure 8: Sieve of Eratosthenes in Python

A more complex example, involving matrix multiplication, is shown below. Assume that a matrix is represented in row-major format, i.e., it is a list of rows, where each row is a list of integers. Matrix multiplication requires computing the inner product of the rows of a first matrix with the columns of the second matrix. We have already written the function `dotprod` for computing this inner product, but to use this function, we need to first convert the second matrix into column-major form. The following code performs this function. (Note that computing the column-major form of a matrix is equivalent to computing its transpose in row-major form, and hence the name `trans`.)

```
let rec trans l =
   let getFirst l = map List.hd l and
       getRest  l = map List.tl l
   in match l with
      | [] -> []
      | []::ls -> []
      | ls -> (getFirst ls)::(trans (getRest ls))
```

Note that `getFirst` extracts and returns the first element of each row in a matrix, while `getRest` returns the remaining elements. On input `[[1;2;3];[4;5;6]]`, `getFirst` yields `[1;4]`, while `getRest` yields `[[2;3][5;6]]`.

Note that the ith row of the product matrix is obtained by computing the inner product of the ith row with each of the columns of the second matrix, so we can use the following code to perform this function:

```
let rowmult row cols =
  (map (fun col -> dotprod row col) cols)
```

Now, we simply need to iterate through the rows of the first matrix in order to compute the entire product matrix:

```
let matmult rows rows2 =
  let cols = (trans rows2) in
    (map (fun row -> rowmult row cols) rows)
```

6 Efficiency of Functional Programs

In the class, we discussed the idea that some times, the most obvious implementations in functional style can be inefficient. Most often, efficiency problems can be traced back to the same fundamental problem in all languages: the use of bad algorithms. For instance, the function `fib n` makes of the order of 2^n recursive calls. Another example of inefficiency discussed in class was the straight-forward implementation of the sum of factorials:

```
let rec facsum n =
    if n == 0 then 0
    else (fact n) + (facsum (n-1))
```

Since a call `fact i` performs `i` recursive calls, and since `facsum n` calls `fact` a total of `n` times, the number of recursive calls is of the order of $\mathtt{n}^2$.

Algorithmic inefficiencies can only be fixed by changing the algorithm. For instance, `facsum` is inefficient because it is not sharing the $\mathtt{i}-1$ multiplications that are common between the computations of `fact i` and `fact (i-1)`. We can address this by computing the factorial of `i` by multiplying factorial of $\mathtt{i}-1$ by `i`. This is what is done in the implementation shown in Figure 6. In this version, `fsum n` makes of the order of `n` recursive calls, and performs just one multiplication and a couple of addition/subtraction operations per invocation. Thus, its runtime is linear in `n`, which is a major improvement over the quadratic runtime of `facsum` above.

A second important source of inefficiency is due to the fact that some operations that are very efficient in imperative languages are inefficient in purely functional languages. For instance, lists in imperative languages can be implemented to provide constant time access to the beginning as well as the end of the list[3]. In contrast, only the first element can be accessed in constant time in OCaml lists. Accessing the last element requires time proportional to the size of the list. Similarly, an operation to append to a list l takes time linear in the size of l in OCaml, but can be performed in constant time in imperative languages. As a result, some seemingly simple functions can become expensive. For instance, consider the following (typical) implementation of `reverse`:

[3]Python lists rely on the imperative nature of the language — they use an array-based implementation that provides constant-time access to any element. However, modification operations are expensive, taking time that is linear in the size of the list.

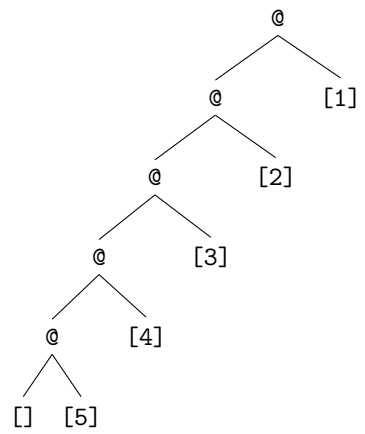

Figure 9: Append operations in `reverse [1;2;3;4;5]`

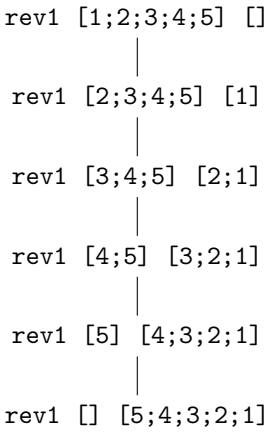

Figure 10: Recursive invocations of `rev [1;2;3;4;5]`

```
let rec reverse l = match l with
    | [] -> []
    | x::xs -> (reverse xs) @ [x]
```

Figure 9 shows the structure of an expression tree resulting from a call to `reverse [1;2;3;4;5]`. It is obvious from this picture that reversing a list with n elements will result in n invocations of append. It is also apparent that the ith append operation (counting from left to right) is appending to a list with i elements. Adding up the times for each append, we arrive at a total runtime of $n(n-1)/2$ for `reverse`. This is surprising: our intuition tells us that we can reverse a list in linear time, but `reverse` is quadratic!

To speed up reversal operation, we need to avoid the inefficient operation of accessing the last element of a list. In fact, this is easy, if we realize that the list is more akin to a stack that provides efficient access to the top but not the bottom. And we can reverse a stack by using a second (empty) stack: pop off elements one by one from the first stack, and then push them onto the second. The function `rev1` shown below uses this approach. It relies on `rev` to call it with the stack to be reversed as the first argument, and an empty stack as the second argument.

```
let rec rev1 l l2 = match l with
    | [] -> l2
    | x::xs -> rev1 xs (x::l2)
let rev l = rev1 l []
```

Figure 10 illustrates the recursive calls made by `rev`. Note that the innermost `rev1` invocation returns the answer, which is its second argument. From the figure, it is obvious that only n recursive invocations are involved, and each of them perform only a constant number of operations. Thus, this version of list reversal takes only linear time!

(A careful reader would have noticed that the OCaml version of Sieve of Eratosthenes ends up generating the prime numbers in reverse order. This is because we wanted to avoid appending new prime numbers to the end of the list, since append is expensive. If prime numbers are needed in increasing order, we can reverse the list at the end — this will be faster than maintaining the list in increasing order at every step of the computation.)

We can speed up `fib` in a similar way using a faster algorithm. The standard way to speed up Fibonacci series computation is to recognize that for any given i, `fib` calls `fib i` far too many times. If we avoided these duplicate calls and instead stored all results in an array, then `fib n` can be computed using a single addition from the results of `fib (n-1)` and `fib (n-2)` that were previously computed and stored in the array. We can apply the same idea to speed up our OCaml version of `fib`, with two minor modifications: (a) we will use lists instead of arrays, and (b) to avoid expensive operations to access the last element of this list, the list will be in the reverse order as compared to the array:

```
let fib n =
   let rec fib2 n l = match l with
       x1::x2::xs ->
         if n == 0 then x1
         else (fib2 (n-1) ((x1+x2)::l))
```

```
in
  if n == 0 then 0
  else if n == 1 then 1
  else fib2 (n-1) [1;0]
```

Note that `fib` just handles the base cases and then sets up the parameters for `fib2` that performs the real work. It stores successive Fibonacci numbers in its list argument, with the most recent ones at the front. Here again, it is useful to think of the list as a stack. The way `fib2` is called, this stack always has two or more elements. Each step of `fib2` picks out the top two elements of the stack, which correspond to the two most recently computed Fibonacci numbers, adds them to compute the next Fibonacci number, which is then pushed to the top of the stack. The first argument is used to keep track of how many such numbers to compute. When it reaches zero, we are done, and we simply return the number at the top of the stack. It is quite easy to see that `fib2 n l` takes time linear in `n`.

7 Functional Programming in Imperative Languages

There are several key elements/features associated with purely functional languages:

- *Absence of side-effects.* Functions in these languages behave like functions in mathematics. They are *referentially transparent,* which ensures that an invocation of a function with the same argument values will always return the same value.
- *Recursive programming style.*
- *Rich type system* that supports:
  - *parametric polymorphism*,
  - versatile built-in types such as lists and tuples,
  - *algebraic data types* and *pattern-matching*, and
  - *type inference.*
- *First-class functions,* i.e., the language permits functions to appear in any context as any other data. In particular, this means support for:
  - *higher-order functions* that can take other functions as parameters
  - *lambda abstractions* that enable new functions to be created on the fly, and possibly returned as return values of functions.
  - *currying* that enables partial application of functions.

Limiting side-effects. Avoiding updates and side-effects leads to programs that are more reliable. More importantly, functions and modules in functional programs are much easier to reuse *correctly* because they do not permit side-effects, which prevents one caller of a function from modifying its future behavior. Moreover, the absence of side-effects permits efficient and correct sharing of data structures without requiring copying. Some of these benefits can be realized through the `const` keyword in C and C++. Both these languages allow immutable objects as well as immutable references to mutable objects, although support for immutable objects in C is limited due to its lack of encapsulation. Dynamic languages such as Python and JavaScript do not support "const" because the concept is philosophically more closely aligned with static typing. Java too has only limited support for const, requiring additional programmer effort to create immutable references.

Richer types. While recursion was included in almost all programming languages designed in the last several decades, it has taken a much longer time for other features (first-class functions and richer type systems) to find their way into today's imperative languages. C++ support for parametric polymorphism, through its template types, became available in the 1990's. Java's support for parametric polymorphism came with the introduce of *generics* in 2000's. Parameterized list types that provide features similar to OCaml lists became available right away, as did tuples of fixed arity. But tuples in their full generality (arbitrary number of components) are still not available in Java, and have been available in C++ only since its 2011 standard. Full type inference has not arrived yet, but C++11's `auto` keyword enables types to be inferred in the most common cases, which involve initialized variables.

In C++14, these features have been further enhanced to support inference of function return types. The addition of type inference has been a great relief for the frustrations surrounding the declaration and use of nested template classes in C++. For instance, the complex declaration

```
std::vector<int>::const_iterator itr = myvec.cbegin();
```

can be replaced by the much more concise

```
auto itr = myvec.cbegin();
```

Unfortunately, this feature has not yet found its way into Java[4]. The type systems of dynamically typed languages such as Python and JavaScript are already more flexible than statically typed languages, so there is no need to add features such as templates to such languages. However, the downside of dynamic typing is that almost all errors can be detected only at runtime, and most working programs will likely contain some type errors that may manifest when unexpected inputs are provided.

Other type-related features, such as algebraic data types and pattern-matching, are still not available in most imperative languages. This is one of the main reasons why it is easier to develop (and teach) interpreters in functional languages such as OCaml[5].

First-class functions. Most higher order features have also been slow to arrive in imperative languages, but by now, mainstream languages have incorporated many of them. Functions as parameters, realized using function pointers, was the first to arrive. It has been included in virtually every imperative language designed since the 1970's. Although Java does not directly support function pointers, the effect can be realized by defining the function within a class, and passing references to objects of this class. Direct support for method references was added in Java 8.

Anonymous functions, also called lambda abstractions, were introduced in C++11, with further improvements (to support type inference) in C++14. They are defined using the square bracket notation as follows:

```
[](int x, int y) { return x + y; }
```

They were also added to Java in a limited form in Java 8.

While currying may not be supported directly in many imperative languages, the closely related concept of *closures* is critical for supporting lambda abstractions. Closures allow a newly created function to contain references to variables in its surrounding context. These references need to be preserved until such time this function is invoked. C++ requires such variables to be explicitly specified within the square brackets denoting the lambda abstraction. Closures enable programmers to implement curried functions:

```
int add(int y) {
  return [y](int x) { return x + y; };
}
```

Note that `y` will be resolved to its binding in the scope surrounding the definition of the anonymous function. (If `y` is not included within the square brackets, a compile time error will result.) When `add` is supplied a single argument, it returns a function that can be passed around to other functions, which can then provide its second argument and cause its evaluation. Alternatively, both arguments could be provided right away, as in `add(2)(3)`. Support for closures have been added, in a somewhat limited form, to Java 8.

Higher order features are fully supported in Javascript, including the ability to pass functions as arguments and return values, lambda abstractions, closures and so on. Lambda abstractions are defined using the keyword **function**:

```
function add(x) {
   return function(y) { return x+y; };
}
```

As in the C++ example above, this JavaScript example creates a curried function `add`. When supplied with its first parameter, it returns an anonymous function that can take another argument and return the sum of the two arguments. Note that `y` is a parameter to the anonymous function, but `x` is a variable that is coming from the context surrounding this function definition.

Python's anonymous functions rely on the keyword **lambda**. Here is an example of a lambda abstraction being used to implement curried functions:

[4] Java allows omission of template parameters, thus reducing clutter a bit, but the gains in the C++ example will be minimal — replacement of `std::vector<int>::const_iterator` with `std::vector<>::const_iterator`.

[5] Of course, the other important reason is that the interpreter is written using the language of mathematical functions, which makes it much easier to reason (or simply convince ourselves) about its correctness.

```
def add(x): return lambda y: x+y
```

Now, `add` can take two arguments, one at a time. A call `add(1)` will return a function object that can accept a second argument. Thus, `add(1)(2)` yields `3`.

In summary, JavaScript and Python provide full support for first-class functions.(There is a minor technical limitation in Python that the body of a lambda abstraction should be an expression, but this restriction would be there is any purely functional language as well.) C++ also provides full support for first-class functions.