CSE 120/130

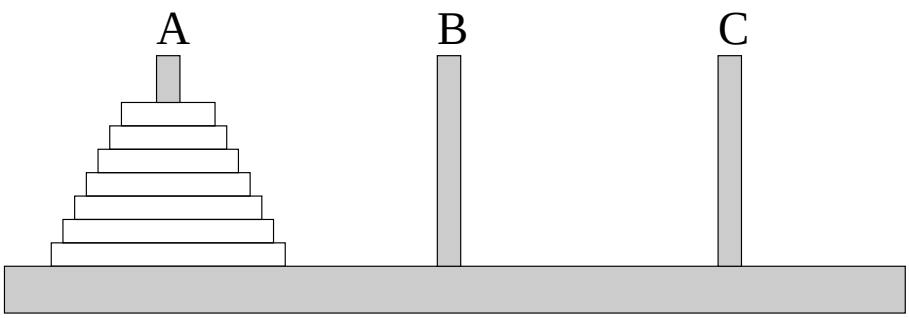
Introduction to

Programming Languages and Techniques

Fall 2000

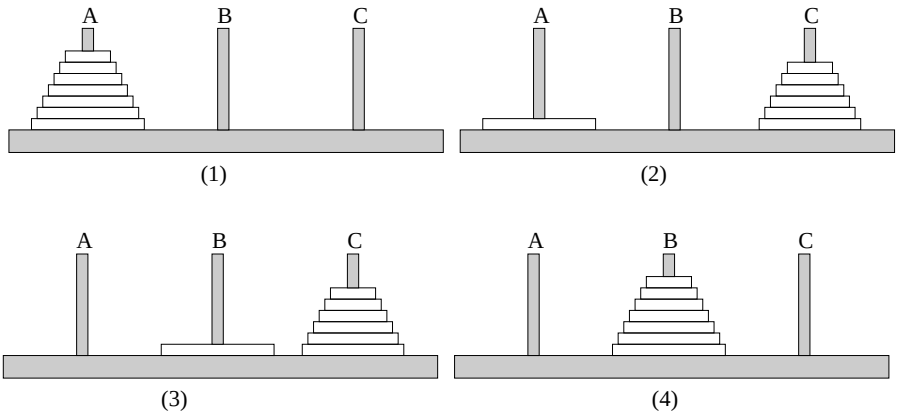
Handout 2

The Towers of Hanoi problem



Goal: Transfer the disks from peg A to peg B, using C for “temporary storage,” moving only one disk at a time and never putting a larger disk on a smaller disk.

The solution



Explanation We have to move n disks (1). Suppose we know how to move $n-1$ disks. Leaving the largest disk at the bottom of A, we move the top $n-1$ to C, using B (2). Next, we move the largest disk to B (3). Finally we move the $n-1$ disks from C, using A, to B where they are placed on top of the largest one (4).

The program

First a small formatting function. (Recall that ^ is string concatenation.)

```
# let  oneMove (s1:string) (s2:string) =
    s1 ^ "->" ^ s2 ^ "; ";;
val oneMove : string -> string -> string = <fun>

# oneMove "A" "B";;
- : string = "A->B; "
```


The recursive function

Returns a string describing the sequence of moves that solves the problem.

```
# let rec toh (numdisks:int)
              (fromP:string) (toP:string) (spareP:string) =
    if numdisks = 0 then ""
    else   (toh (numdisks-1) fromP spareP toP)   (* (2) *)
         ^ (oneMove fromP toP)                   (* (3) *)
         ^ (toh (numdisks-1) spareP toP fromP)   (* (4) *);;

val toh : int -> string -> string -> string -> string = <fun>
```

(2), (3), and (4) refer to the steps in the solution diagram a couple of slides back.

An example

```
# toh 4 "A" "B" "C";;

- : string =
  "A->C; A->B; C->B; A->C; B->A; B->C; A->C; A->B;
   C->B; C->A; B->A; C->B; A->C; A->B; C->B; "
```


A digression on syntax

When we write something like `square 3` in ML we mean "apply `square` to `3`".

The general rule is that if `x` and `y` are ordinary names or numbers (they are not operators like `+`, `*`, etc.) then `x y` means "apply the function `x` to the argument `y`".

We can write `x(y)` if we want, but these parentheses are unnecessary.

Parentheses are needed in cases like `square (5 - 1)`. What would happen if we omitted them?

If we write `x y z`, ML's interpretation of this is that a two-argument function named `x` is being applied to the arguments `y` and `z`.

Thus `square square 3` will produce an error. We need to write `square(square 3)`.

Real Numbers

Real (or "floating point") numbers are always written in OCaml using a decimal point.

```
# 2.1 *. 3.4;;
- : float = 7.14

# 1.3 +. 4.7;;
- : float = 6

# 1.3 -. 4.7;;
- : float = -3.4
```

Note that the names of the addition, subtraction, and multiplication operators are different when their arguments are `float`s rather than `int`s (they add a "`.`").

Inside the computer, real numbers are represented in a completely different way than integers. The code for adding and multiplying real numbers is also completely different.

Real operations do not work on integers:

```
# 1 +. 2;;
Characters 0-1:
This expression has type int but is here used with type float
```

And vice versa:

```
# 1.2 * 3.4;;
Characters 0-3:
This expression has type float but is here used with type int
```

There is no built-in operation for adding an integer to a real. To do this we must first "float" the integer.

```
# float 3 +. 4.5;;
- : float = 7.5
```

(Note that `float` is the name of both a function and a type!)

Note that *division* behaves differently on reals and integers.

```
# 5.0 /. 3.0;;
- : float = 1.66666666667

# 5 / 3;;
- : int = 1
```

The integer division operation always returns an integer.

Some real operations have no corresponding operation on integers:

```
# sqrt 10.0;;
- : float = 3.16227766017
```

And vice versa:

```
# 5 mod 3;;
- : int = 2
```

`x mod y` operation returns the *remainder* when `x` is divided by `y`. In other words:

```
x  =  (x / y) * y + x mod y
```


Overloading

Many languages (C, Java, Pascal, Standard ML) provide some form of *overloading*. The behavior of a function is determined by the type of its arguments. In C, for example, we can write `1 + 2`, `1 + 2.5`, and `1.3 + 2.5`. In each case a different operation is being carried out, as we can see if we translate into OCaml:

C/Java, etc	OCaml
`1 + 2`	`1 + 2`
`1 + 2.5`	`float(1) +. 2.5`
`1.3 + 2.5`	`1.3 +. 2.5`

In C/Java the symbol `+` is said to be *overloaded*. In Java we can even overload symbols that we define ourselves.

Overloading is a convenience: it does not fundamentally increase the power of the language. Its main benefit is that it frees programmers from having to remember different names for the same operation on different types. Its main cost is that it increases the complexity of the language.

Integer vs. Real Arithmetic

Integers in OCaml are represented using a fixed number of bits. This can lead to wrong results when we calculate with very large numbers.

```
# 600000000 + 600000000;;
- : int = -947483648
```

For reasons of efficiency, OCaml does not tell us about the error here, but just returns "junk." (Some languages will detect and report the error, but returning junk is more common.)

In both OCaml and most other languages we can get round the problem by using real numbers.

```
# 600000000.0 +. 600000000.0;;
- : float = 1200000000
```


The Dangers of Real Arithmetic

However, real numbers have their own pitfalls.

Look at this example. (`1e15` is "scientific notation" for a `1` followed by `15` zeroes).

```
# 1e15 +. 1.0 > 1e15;;
- : bool = true

# 1e16 +. 1.0 > 1e16;;
- : bool = false
```

Computers can only represent *approximations* to real numbers.

When the numbers get too big (or too small) these approximations can lead to errors in real arithmetic. Almost all programming languages can be made to "go wrong" this way. Indeed, programming numerical calculations with a high degree of precision is a major field of applied mathematics and computer science (called "numerical analysis").

The Newton-Raphson algorithm for square roots

We can write a program to calculate square roots by using the observation, from calculus, that if x is an approximation to $\sqrt{a}$, then $(x + a/x)/2$ is a better approximation.

```
# (* If x is an approximation to sqrt(a), then
     the result of newApprox is a better approximation *)
  let newApprox (x:float) (a:float) =
    (x +. a/.x) /. 2.0;;
val newApprox : float -> float -> float = <fun>

# (* Test whether x is "close enough" to the square root of a *)
  let closeEnough (x:float) (a:float) =
    (x*.x -. a) < 0.0000001 && -0.0000001 < (x*.x -. a);;
val closeEnough : float -> float -> bool = <fun>
```


This is actually not a great way of testing closeness: it may lead to non-termination in the case that `a` is very large. An industrial strength implementation of the Newton-Raphson algorithm would use a more sophisticated method.

The Newton-Raphson algorithm – continued

```
# (* Starting from approximation x, get a "close enough"
     approximation *)
  let rec sqrtAux (x:float) (a:float) =
    if closeEnough x a then x
    else sqrtAux (newApprox x a) a ;;
val sqrtAux : float -> float -> float = <fun>

# (* Calculate square roots, starting with an approximation of 1.0 *)
  let sqrt (a:float) = sqrtAux 1.0 a ;;
val sqrt : float -> float = <fun>
```


Some results

It's interesting to see how the sequence of approximations converges. This is the sequence of approximations generated by `sqrt 1024.0`

```
512.5
257.24902439
130.61480157
69.2273240545
42.009585631
33.1924874169
32.021420905
32.0000071648
32.0
```

Note how quickly the series converges once it gets "close". Can you think of a faster way of getting close initially?

Some more syntax

It is remarkable how much computation we have been able to do using *only* the idea of functional/procedural abstraction.

Some of our computations were less readable than we might have liked, because they repeated the same subexpression more than once. For example, we just saw the function

```
let closeEnough (x:float) (a:float) =
  (x*.x -. a) < 0.0000001 && -0.0000001 < (x*.x -. a)
```

which calculates `(x*.x -. a)` twice.

OCaml has a construct `let...in...` that allows us to give temporary values to names.

```
let closeEnough (x:float) (a:float) =
  let d = (x*.x -. a) in
  d < 0.0000001 && -0.0000001 < d
```


To see that this is a temporary definition (sometimes called a temporary *binding*), look at the following behaviors:

```
# let x = 4 in x*x;;
- : int = 16

# x*x;;
Characters 0-1:
Unbound value x
```

```
# let y = 7;;
val y : int = 7

# let y = "cow" in y^y;;
- : string = "cowcow"

# y;;
- : int = 7
```

Hiding names

When defining `sqrt`, we used a number of other functions that are probably of no general interest. We can use `let...in...` to bundle these functions into the definition of `sqrt`:

```
# let sqrt(a:float) =
    let newApprox (x:float) (a:float) = (x +. a/.x) /. 2.0 in

    let closeEnough (x:float) (a:float) =
      let  d = (x*.x -. a) in
         d < 0.0000001 && -0.0000001 < d in

    let rec sqrtAux (x:float) (a:float) =
      if closeEnough x a then x
      else sqrtAux (newApprox x a) a in

    sqrtAux 1.0 a;;
```

(We've removed comments to make the structure easier to see.)

Local scope

Now that we have moved all the auxiliary functions into a region of program text where the parameter `a` is defined, we can simplify the functions:

```
# let sqrt (a:float) =
    let newApprox (x:float)  = (x +. a/.x) /. 2.0 in

    let closeEnough (x:float)  =
      let  d = (x*.x -. a) in
         d < 0.0000001 && -0.0000001 < d in

    let rec sqrtAux (x:float)  =
      if closeEnough x  then x
      else sqrtAux (newApprox x)  in

    sqrtAux 1.0;;
```

Lists – a useful data type

Lists provide a flexible and general mechanism for representing sequences of data. They are provided as a "built in" type in ML and a number of other languages such as Lisp, Scheme, and Prolog.

An easy way to specify a list in OCaml is as follows:

```
# [1; 3; 2; 5];;
- : int list = [1; 3; 2; 5]
```

Note the type that ML gives for this list: `int list` for "integer list" or "list of integers".

The type of lists

```
# ["cat"; "dog"; "gnu"];;
- : string list = ["cat"; "dog"; "gnu"]

# [true; true; false];;
- : bool list = [true; true; false]

# [[1; 2]; [2; 3; 4]; [5]];;
- : int list list = [[1; 2]; [2; 3; 4]; [5]]
```

If we have a type `T`, we always have a type `T list`.

ML lists are homogeneous

OCaml will not allow you to mix types within a list:

```
# [1; 2; "dog"];;
Characters 7-13:
This expression has type string list but is here used with type int list
```


Examples of using lists

We use some built-in OCaml operations such as `List.rev` and `@` that respectively reverse and append lists. These are not the most fundamental operations on lists, though (we'll see those shortly).

```
# [1; 2; 3] @ List.rev [1; 2; 3];;
- : int list = [1; 2; 3; 3; 2; 1]
```

The functions `explode` and `implode` convert between strings and lists of characters. (They are not built in, but are easy to write.)

```
# explode "bomb";;
- : char list = ['b'; 'o'; 'm'; 'b']

# implode ['c'; 'o'; 'w'];;
- : string = "cow"
```

Note that we are seeing a new type `char` for the first time.

OCaml distinguishes between strings of size 1 and characters:

```
# 'a' = "a";;
Characters 5-8:
This expression has type string but is here used with type char
```


Simple examples with lists – continued

```
#  let stringRev (s:string) = implode(List.rev(explode s));;
val stringRev : string -> string = <fun>

# stringRev "eldoon";;
- : string = "noodle"

# let palindrome (s:string) = (s = stringRev s);;
val palindrome : string -> bool = <fun>

# palindrome "able was i ere i saw elba";;
- : bool = true
```

map: “apply-to-each”

OCaml has a predefined function `List.map f l`, where `f` is a function and `l` is a list, that produces another list by applying `f` to each element of `l`. We'll soon see how to define `List.map`, but for the time being let's see how it works:

```
# List.map square [1; 3; 5; 9; 2; 21];;
- : int list = [1; 9; 25; 81; 4; 441]

#  List.map stringRev ["I"; "cannot"; "tell"];;
- : string list = ["I"; "tonnac"; "llet"]
```

Note the *polymorphic* (generic) nature of `List.map`: it works for lists of integers as well as for lists of strings. With this example we are getting closer to ML's real power.

More on map

An interesting feature of `List.map` is that one of its arguments is itself a function. For that reason we say that `List.map` is *higher-order*.

Here's another use of `List.map`. The function `Char.uppercase` does what its name suggests.

```
# let  upCase (s:string) =
     implode(List.map Char.uppercase (explode s));;
val upCase : string -> string = <fun>

# upCase("This is an important message");;
- : string = "THIS IS AN IMPORTANT MESSAGE"
```

The need for higher-order functions arises naturally in programming. They are easy to define in ML. Some languages do not allow higher-order functions. In Java, as we shall see, they can only be defined in a round-about way.

Another useful higher-order function

We will also be able to define the built-in higher-order function `List.filter p l` that extracts from `l` the list those elements that satisfy the predicate `p` which is a *predicate* – a function returning a boolean.

```
# let even (n:int) = (n mod 2) = 0;;
val even : int -> bool = <fun>

# List.filter even [1; 2; 3; 4; 5; 6; 7; 8; 9];;
- : int list = [2; 4; 6; 8]

# List.filter palindrome  ["glengarry"; "glenross"; "glenelg"];;
- : string list = ["glenelg"]
```

`List.filter` is also polymorphic.

Lists – constructing them

We now look at the *fundamental operations* on lists. As with many data types, these operations are of two kinds: those that *construct* lists and those that *take them apart*.

Lists are constructed using

- `[]`, the empty list, pronounced “nil”.
- An operation `::` . `x :: l` puts the element `x` onto the beginning of the list `l`. `::` is pronounced “cons”.

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]

# "cat" :: [] ;;
- : string list = ["cat"]
```

List constructors

Any list can be built by "consing" its elements together:

```
-# 1 :: 2 :: 3  :: 2 :: 1 :: [] ;;;
- : int list = [1; 2; 3; 2; 1]
```

We now see that

$$[x_1; x_2; \ldots; x_n]$$

is simply a "shorthand" for

$$x_1 :: x_2 :: \ldots :: x_n :: []$$

Note that by convention `::` associates to the right: `1 :: 2 :: []` is the same as `1 :: (2 :: [])`.

Some functions that generate lists

```
# let rec repeat (k:int) (n:int) =  (* A list of n copies of k *)
    if n = 0 then []
    else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# let rec fromTo (m:int) (n:int) = (* The numbers from m to n *)
    if n < m then []
    else m :: fromTo (m+1) n;;

# fromTo 9 18;;
- : int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]
```

Modules – a brief digression

OCaml has things called *modules*, which have several uses. The property that interests us here is that they act as *namespaces*, which are groups of names for various functions, values etc.

So, for example, the module `List` contains a number of useful functions like `List.map`, `List.filter`, `List.reverse`, `List.length`, etc.

If we are working intensively with lists, typing in these long names can be annoying. We can make life easier by "opening" a module, e.g.,

```
#open List;;
```

After doing this, we can use `reverse` instead of `List.reverse`, `length` instead of `List.length` etc.

So why don't we simply open *all* the modules? The problem is that we there will be "conflicts", e.g. between `String.length` and `List.length`. The last one opened wins. So if we do

```
open String;;
open List;;
```

At this point `length` means `List.length`. The `length` function for strings has disappeared, though it is still available as `String.length`. More importantly there are probably a few hundred identifiers in OCaml modules. If we open all the modules, we might well "clobber" some code that we have already written.

Lists – taking them apart

From now on we'll assume that the list module has been opened with the `open List;;` directive. Lists are taken apart with two operations

- `hd` (pronounced "head") gives the first element of a list.

```
# hd [1; 2; 3];;
- : int = 1
```

- `tl` (pronounced "tail") gives everything but the first element.

```
# tl [1; 2; 3];;
- : int list = [2; 3]
```

(Lisp programmers say "car" and "cdr" instead of "head" and "tail")

An emptiness test

Finally we need to test whether a list is empty. We'll see various ways of doing this, but in OCaml the obvious way is to ask whether it is equal to the empty list: empty.

```
# [4; 3; 2] = [];;
- : bool = false
```

The emptiness test is so useful that many languages (lisp and other dialects of ML) have a built in `null` function, which we can easily define in OCaml id we want.

That's it. These five fundamental operations on lists: `[]`, `::`, `hd`, `tl`, and an equality test suffice to allow programmers to define "all" the others.

What "all" means in the last sentence is a foundational question for Computer Science. Interestingly, this also has a deep connection with the question whether mathematical reasoning can lead to paradoxes. Mathematical *logicians* have developed the theory of *computable* functions as part of their analysis of the notion of mathematical proof. If Electronics is the technological progenitor of Computer Science then Mathematical Logic is its scientific progenitor.

Some examples

- `hd (tl (tl [1; 4; 5; 6]))`
- `hd (hd [[5; 4]; [3; 2]])`
- `null (1 :: [])`
- `tl (tl (8 :: 7 :: 6 :: 5 :: 4 :: []))`
- Give some values of `x, y` for which `(x :: y) :: z` makes sense.

Recursion on lists

Most useful functions on lists are written using recursion. For example to sum the numbers in a sequence a mathematican might say

$$\begin{aligned} \text{seq-sum}(s) &= 0 && \text{if } s \text{ is empty} \\ \text{seq-sum}(s) &= \text{hd}(s) + \text{seq-sum}(\text{tl}(s)) && \text{otherwise} \end{aligned}$$

We would write

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else hd l + listSum (tl l);;

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Notice how similar this function is to the recursive functions we wrote on integers.

Recursion on lists – continued

Here are some more examples:

```
# let rec length (l: int list) =
    if l = [] then 0
    else 1 + length (tl l);;

# let rec listProd (l: int list) =
    if l = []  then 1
    else hd l * listProd (tl l);;

# let fact (n:int) = listProd (fromTo 1 n);;

# fact 6;;
- : int = 720
```

The last element of a list

```
# let rec last l =
    if tl l = [] then hd l
    else last (tl l);;

# last [1; 4; 5; 2; 7];;
- : int = 7
```

That's odd! We forgot to give a type for the argument of last, and yet it worked!

The last element of a list – continued

Moreover last works for lists of other types:

```
# last ["The"; "fat"; "cat"];;
- : string = "cat"

# last (last [[1; 4; 4]; [5; 3]]);;
- : int = 3
```

This is precisely *because* we "forgot" to give a type to its argument.

Note that `last` works equally well on integer lists and string lists. What type should we give to the argument? If we were using C, or even Java, we would be required to specify the kind of list, e.g. integer list for the argument – in which case the function wouldn't work on a string list.

In fact, without a "hack" in C or Java, we have to write a different function for each kind of list, *even though the code is identical*.

Polymorphism

Let's look at what OCaml prints out when we input the function `last`:

```
# let rec last l =
    if tl l = [] then hd l
    else last (tl l);;
val last : 'a list -> 'a = <fun>
```

OCaml reports that it has found something of type `'a list -> 'a`. Here `'a` – pronounced "alpha" – is a *type variable*. It can be instantiated with any other type. So some possible types for the function are

```
int list -> int
string list -> string
int list list -> int list
```

However the function cannot, for example, be used at type `string list -> int`.

The ability to use type variables is called (parametric) *polymorphism*. It is an extremely powerful feature of languages in the ML family, but not (yet) available in languages such as Java.

append

From now on, it will help in your understanding of functions if you look at the types OCaml infers for them.

```
# let rec append l1 l2 =  (* this is a prefix version of @ *)
    if l1 = [] then l2
    else hd l1  :: append (tl l1) l2;;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [4; 3; 2]  [6; 6; 7];;
- : int list = [4; 3; 2; 6; 6; 7]
```

snoc

```
# let rec snoc l x =  (* put x on the end of l *)
    if l = [] then x::[]
    else hd l  :: snoc(tl l) x;;
val snoc : 'a list -> 'a -> 'a list = <fun>

# snoc [5; 4; 3; 2] 1;;
- : int list = [5; 4; 3; 2; 1]
```

Look closely at the types of `append` and `snoc`.

Reversing a list

We can use `snoc` to reverse a list:

```
let rec  rev l = (* Reverses l -- inefficiently*)
      if l = []  then []
      else snoc (rev (tl l)) (hd l)
val rev : 'a list -> 'a list = <fun>

# rev [1; 2; 3; 3; 4];;
- : int list = [4; 3; 3; 2; 1]
```

Why is this inefficient? How can we do better?

Defining map

Recall that `map` is a function that applies another function to each member of a list – to produce a list of results.

```
let rec  map f l =
      if l = [] then []
      else f (hd l) :: map f (tl l)
 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

The type of `map` is probably even more polymorphic than you expected! The result can even be a list of elements of a *different* type:

```
# map String.length ["The"; "quick"; "brown"; "fox"];;
- : int list = [3; 5; 5; 3]
```


The first argument of `map` is a function – itself polymorphic – that takes an `'a` (alpha) and produces a `'b` (beta). The second is an `'a list`. The result of `map` is a `'b list`.

Note the use of `String.length`. What would have happened if we had used `length` instead?

Defining filter

```
let rec filter p l =
      if l = [] then []
      else if p (hd l) then hd l :: filter p (tl l)
           else filter p (tl l)

val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Again, make sure you understand its type.

Note that using just the fundamental operations and recursion we were able to define also `rev` and `append` (a prefix version of the infix `@`), built-in operations that we have seen before.

Some exercises:

1. Define `implode` by recursion on lists. You will need a function `charToString : char -> string` that turns a character into a string of length 1. (By the way, `explode` has nothing to do with recursion on lists.)
2. Define by recursion on lists a function that concatenates the strings in a list, `concat : string list -> string`.
3. Define `implode` without recursion on lists, instead using `map`, `Char.toString` and `concat`.

Approaches to Typing

- A **strongly typed** language prevents programs from accessing private data, corrupting memory, crashing the machine, etc.
- A **weakly typed** language does not.
- A **statically typed** language performs type-consistency checks at when programs are first entered.
- A **dynamically typed** language delays these checks until programs are executed.

	Weak	Strong
Dynamic	PERL	Lisp, Scheme
Static	C, C++	ML, ADA, Java*

*Strictly speaking, Java should be called “mostly static”

Practice with Types

What are the types of the following functions?

- `let f (x:int) = x + 1`
- `let f x = x + 1`
- `let f (x:int) = [x]`
- `let f x = [x]`
- `let f x = x`
- `let f x = hd(tl x) :: [1.0]`
- `let f x = hd(tl x) :: []`
- `let f x = 1 :: x`
- `let f x y = x :: y`

- `let f x y = x :: []`
- `let f x = x @ x`
- `let f x = x :: x`
- `let f x y z = if x>3 then y else z`
- `let f x y z  = if x>3 then y else [z]`

And one more:

```
let rec f x  =
    if (tl x) = [] then x
    else f (tl x)
```


Aside: Polymorphism

The polymorphism in ML that arises from type parameters is an example of *generic programming*. (`mapl`, `filter`, etc.) Are good examples of generic functions.

Different languages support generic programming in different ways...

- **parametric polymorphism** allows functions to work *uniformly* over arguments of different types.E.g., `last : 'a list -> 'a`
- **ad hoc polymporphism** (or **overloading**) allows an operation to behave in *different* ways when applied to arguments of different types. There is no such polymorphism in OCaml, but most languages allow some overloading (e.g.`2+3` and `2.4 +3.6`). Java and C++ allow one to extend the overloading of a symbol (e.g. `"dog" + "house"`). This form of overloading is a *syntactic* convenience, but little more.
- **subtype polymporphism** allows operations to be defined for collections of types sharing some common structure

  e.g., a `feed` operation might make sense for values of `animal` and all its "refinements"—`cow`, `tiger`, `moose`, etc.

OCaml supports parametric polymorphism in a very general way, and also supports subtyping (Though we shall not get to see this aspect of OCaml, its support for subtyping is what distinguishes it from other dialects of ML.) It does not allow overloading.

Java provides a subtyping as well as moderately powerful overloading, but no parametric polymorphism. (Various Java extensions with parametric polymorphism are under discussion.)

Confusingly, the bare term “polymorphism” is used to refer to parametric polymorphism in the ML community and for subtype polymorphism in the Java community!

Environments

Earlier, we described how to evaluate programs “by hand” by a process of substitution. We are now going to introduce a different (but equivalent) way of thinking about evaluation, called the *environment* model of evaluation.

The environment model is closer to the mechanism by which programs are actually evaluated in a computer.

When we start OCaml, a variety of symbols already have meanings attached to them. They are mostly functions – like `+`, `*`, `::`, `mod`, `not`, `nil`, `float`, etc.

The association between a symbol and its meaning is often called a *binding*. It is like an entry in a dictionary. When we start up OCaml, a set of bindings is already defined. This set of bindings (a kind of dictionary) is called an *environment*.

The textbook draws environments as diagrams something like this

`float`	$\rightarrow$	*code for float*
...		...
`::`	$\rightarrow$	*code for cons*
`[]`	$\rightarrow$	*representation of nil*

In this diagram, we have used `teletype` font for the actual symbols and *italic* font to describe the associated meaning.

Extending an environment

When we define new symbols (i.e. create new bindings) we *extend* the enviroment.

For example, if we write `let pi = 3.14159` the environment becomes.

`pi`	$\rightarrow$	*3.14159*
`float`	$\rightarrow$	*code for float*
...		...
`::`	$\rightarrow$	*code for cons*
`[]`	$\rightarrow$	*representation of nil*

We may re-define a symbol that was already defined. For example, if we defined

```
let pi = 3.14159
let float (x) = pi *. x *. x
```

the environment would look like

`float`	$\rightarrow$	*new code for float*
`pi`	$\rightarrow$	*3.14159*
`float`	$\rightarrow$	*predefined code for float*
`...`		`...`
`::`	$\rightarrow$	*predefined code for cons*
`[]`	$\rightarrow$	*predefined representation of nil*

The old code for `float` has not gone away, but we search for meanings starting from the top. So any subsequent use of `float` in our program refers to the definition on top.

Understanding `let...in...`

Look at the following interaction:

```
# let x = 5;;
val x : int = 5

# let x = 3*3 in
  let y = 4*4 in
  x*x + x*y + y*y;;
- : int = 481

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

Just after the first `x` was defined, the environment looked like

x	$\rightarrow$	*5*
...		...

The effect of the `let x = ... in let y = ...` was to extend the environment to

y	$\rightarrow$	*16*
x	$\rightarrow$	*9*
x	$\rightarrow$	*5*
...		...

The expression `x*x + x*y + y*y` after the second `in` is evaluated in this environmment.

After this expression the environment reverts to what it was before the `let`, that is

x	$\rightarrow$	*5*
...		...

Recursive functions – a brief review

Try to figure out what the following functions do. Some of them compute things we've already seen.

```
let rec what lo hi =
  if lo = hi then hi
  else let mid = (lo + hi)/2 in
         what lo mid * what (mid+1) hi


let rec whatnext n =
    if n = 0 then 1
    else let d = n / 2 in
         let m = n mod 2 in
         let w = whatnext d in
           if m = 1 then 2*w*w else w*w
```

```
let rec whatsit n =  (* There are more efficient ways of doing this *)
    let rec aux i =
        if i = 1 then true
        else n mod i <> 0 && aux (i-1)
    in aux (n-1)

let rec take n l =
    if n = 0 then []
    else hd l  :: take (n-1) (tl l)

let rec drop n l  =
    if n=0 then l
    else drop (n-1) (tl l)
```


Counting Iterations

Consider the function defined by

$$f(n) = n/2 \quad \text{if } n \text{ is even}$$
$$f(n) = 3n+1 \quad \text{otherwise}$$

Suppose we repeatedly apply f to a number, how long does it take to get to 1? If we start at 6, for example, we get the sequence

$$6, 3, 10, 5, 16, 8, 4, 2, 1$$

with a total of 8 iterations.

It is conjectured that, from an arbitrary starting point, one always gets to 1. Let us assume that it does, and let us write a function to compute the number of iterations to get to one.

If the conjecture is false, we'll be writing a function that doesn't terminate.

It sometimes simplifies the coding of such problems by generalizing them. Here is the generalization. Suppose f is a function. How many times do we have to apply f to a starting value v in order to achieve a value which satisfies a predicate p?

Here is some OCaml "pseudo-code" for this problem.

```
let rec count v =
        if p v then 0
        else 1 + count(f v)
```

This pseudo-code is easily turned into working OCaml code by making the functions p and f parameters of the function count

```
let rec count p f v =
    if p v  then 0
    else 1 + count p  f  (f v)
```


Now all we have to do is to define the p and f for this specific problem.

```
let next n  = if n mod 2 = 0 then n/2 else 3*n + 1

let isOne n  = n = 1

let countA n = count isOne next  n
# countA 6;;
- : int = 8

# countA 10;;
- : int = 6

# countA 111;;
- : int = 69
```