A common pattern: map

Pattern: take a list and produce a new list,
where each element of the output is calculated from the corresponding element of the input

`map` captures this pattern

```
map: ('a -> 'b) * 'a list -> 'b list
```

Example:

- have a list of fahrenheit temperatures for Seattle days
- want to give a list of temps to friend in England
- specification: convert each temp (F) to temp (C)

```
- fun f2c(f_temp) = (f_temp - 32.0) * 5.0/9.0;
val f2c = fn : real -> real
- val f_temps = [56.4, 72.2, 68.4, 78.4, 45.0];
val f_temps = [56.4,72.2,68.4,78.4,45.0]
            : real list
- val c_temps = map(f2c, f_temps);
val c_temps = [13.5555555556,
               22.3333333333,
               20.2222222222,
               25.7777777778,
               7.22222222222] : real list
```


Another common pattern: filter

Pattern: take a list and produce a new list
of all the elements of the first list that pass some test (a **predicate**)

`filter` captures this pattern

```
filter: ('a -> bool) * 'a list -> 'a list
```

Example:

- have a list of day * temp
- want a list of nice days

```
- fun nice_day(temp) = temp >= 70.0;
val nice_day = fn : real -> bool

- val nice_days = filter(nice_day, f_temps);
val nice_days = [72.2,78.4] : real list
```


Another common pattern: find

Pattern:
take a list and return the first element that passes some test,
raising NotFound if no element passes the test

`find` captures this pattern

```
find: ('a -> bool) * 'a list -> 'a
exception NotFound
```

Example: find first nice day

```
- val a_nice_day = find(nice_day, f_temps);
a_nice_day = 72.2 : real
```


Anonymous functions

Map functions and predicate functions often pretty simple,
only used as argument to map, etc.,
don't merit their own name

Can directly write anonymous function **expressions**:
fn pattern$_{formal}$ **=>** expr$_{body}$

```
- fn(x)=> x + 1;
val it = fn : int -> int
- (fn(x)=> x + 1)(8);
9 : int

- map(fn(f)=> (f - 32.0) * 5.0/9.0, f_temps);
val it = [13.5555555556,...] : real list

- filter(fn(t)=> t < 60.0, f_temps);
val it = [56.4,45.0] : real list
```

Fun vs. fn

`fn` expressions are a primitive notion

`val` declarations are a primitive notion

`fun` declarations are just a convenient syntax for `val` + `fn`

```
fun f(args) = expr
```
is sugar for
```
val f = (fn(args)=> expr)
```

```
fun succ(x) = x + 1
```
is sugar for
```
val succ = (fn(x) => x + 1)
```

Explains why the type of a `fun` declaration prints like a `val` declaration with a `fn` value

```
val succ = fn : int -> int
```

Symptoms of good design

- orthogonality of primitives
- syntactic sugar for common combinations

Nested functions

An example

```
- fun good_days(good_temp:real,
=               temps:real list):real list =
=   filter(fn(temp)=> (temp >= good_temp),
=          temps);
val good_days = fn : real*real list -> real list

(* good days in Seattle: *)
- good_days(70.0, f_temps)
val it = [72.2,78.4] : real list

(* good days in Fairbanks: *)
- good_days(32.0, f_temps)
val it = [56.4,72.2,68.4,78.4,45.0] : real list
```

What's interesting about the anonymous function expression `fn(temp)=> (temp >= good_temp)` ?

Nested functions and scoping

If functions can be written nested within other functions (whether named in a `let` expression, or anonymous) then can reference local variables in enclosing function scope

Makes nested functions a lot more useful in practice

Beyond what can be done with function pointers in C/C++

A general pattern: reduce

The most general pattern over lists simply abstracts the standard pattern of recursion

Recursion pattern:

```
fun f(..., nil, ...) = ... (* base case *)
  | f(..., x::xs, ...) =
      (* inductive case *)
      ... x ... f(..., xs, ...) ...
```

Parameters of this pattern, for a list argument of type `'a list`:

- what to return as the base case result (`'b`)
- how to compute the inductive result from the head and the recursive call (`'a * 'b -> 'b`)

`reduce` captures this pattern

```
reduce: ('a*'b -> 'b) * 'b * 'a list -> 'b
```

ML's form of a loop over a list

Examples using reduce

```
reduce: ('a*'b -> 'b) * 'b * 'a list -> 'b
```

Summing all the elements of a list

```
- val rainfall = [0.0, 1.2, 0.0, 0.4, 1.3, 1.1];
val rainfall = [0.0,1.2,0.0,0.4,1.3,1.1]
             : real list

- val total_rainfall =
=   reduce(fn(rain,subtotal)=>rain+subtotal,
=          0.0, rainfall);
val total_rainfall = 4.0 : real
```


Modules for name-space management

A file full of types and functions can be cumbersome to manage

Would like some hierarchical organization to names

Modules allow grouping declarations to achieve a hierarchical name-space

`structure` declarations in ML create modules

```
- structure Assoc_List = struct
=   type (''k,'v) assoc_list = (''k*'v) list
=   val empty = nil
=   fun store(alist, key, value) = ...
=   fun fetch(alist, key) = ...
= end;
structure Assoc_List : sig
  type ('a,'b) assoc_list = ('a*'b) list
  val empty : 'a list
  val store : (''a*'b) list * ''a * 'b ->
              (''a*'b) list
  val fetch : (''a*'b) list * ''a -> 'b
end
```


Using structures

To access declarations in a structure, use dot notation

```
- val league = Assoc_List.empty;
val l = [] : 'a list

- val league =
=   Assoc_List.store(league, "Mariners", {..});
val league = [("Mariners", {..})]
           : (string*{..}) list

- ...

- Assoc_List.fetch("Mariners");
val it = {wins=78,losses=4} : {..}
```

Other definitions of `empty`, `store`, `fetch`, etc. don't clash

Common names can be reused by different structures

The open declaration

To avoid typing a lot of structure names, can use the **open** *`struct_name`* declaration to introduce local synonyms for all the declarations in a structure (usually in a `let` or within some other struct)

```
fun create_league(names) =
  let
    open Assoc_List
    val init = {wins=0,losses=0}
  in
    reduce(fn(name,league)=>
              store(league,name,init),
           empty, names)
  end
```

Modules for encapsulation

Want to hide details of data structure implementations
from clients, i.e., **data abstraction**

- simplify interface to clients
- allow implementation to change without affecting clients

In C++ and Java, use `public`/`private` annotations

In ML:

- define a `signature` that specifies the desired interface
- specify the signature with the `structure` declaration

E.g. a signature that hides the implementation of `assoc_list`:

```
- signature ASSOC_LIST = sig
=   type (''a,'b) T
=   val empty : (''a,'b) T
=   val store : (''a,'b) T * ''a * 'b ->
=               (''a,'b) T
=   val fetch : (''a,'b) T * ''a -> 'b
= end;
signature ASSOC_LIST = sig ... end
```


Specifying the signatures of structures

Specify desired signature of structure when declaring it:

```
- structure Assoc_List :> ASSOC_LIST = struct
=   type (''k,'v) T = (''k*'v) list
=   val empty = nil
=   fun store(alist, key, value) = ...
=   fun fetch(alist, key) = ...
=   fun helper(...) = ...
= end;
structure Assoc_List : ASSOC_LIST
```

The structure's interface is the given one,
not the default interface that exposes everything

Hidden implementation

Now clients can't see implementation, nor guess it

```
- val teams = Assoc_List.empty;
val teams = - : (''a,'b) Assoc_List.T

- val teams' = "Mariners"::"Yankees"::teams;
Error: operator and operand don't agree
  operator: string * string list
  operand:  string * (''Z,'Y) Assoc_List.T

- Assoc_List.helper(...);
Error: unbound variable helper in path
  Assoc_List.helper

- type Records = (string,...) Assoc_List.T;
type Records = (string,...) Assoc_List.T
- fun sortStandings(nil:Records):Records = nil
=   | sortStandings(pivot::rest) = ...;
Error: pattern and constraint don't agree
  pattern:    'Z list
  constraint: Records
in pattern: nil : Records
```

How to write `sortStandings`, if implementation is hidden?

Including reduce etc. in external interfaces

To provide a complete interface if representation is hidden,
often need to include ways of traversing the data structure

Reduce or its equivalent is often needed,
as the most general pattern of iteration or recursion

E.g.:

```
- signature ASSOC_LIST = sig
=   ...
=   val reduce: ((''a * 'b) * 'c) * 'c *
=               (''a,'b) T -> 'c
= end
= structure Assoc_List :> ASSOC_LIST = struct
=   ...
=   fun reduce(f, base, alist) = ...
= end;
...
- fun sortStandings(records) =
=   ... Assoc_List.reduce(..., records) ...
...
```

Modules vs. classes

Classes (abstract data types) implicitly define a **single** type, with associated constructors, observers, and mutators

Modules can define 0, 1, or many types in same module, with associated operations over several types

- no new types if adding operations to existing type(s)
  - hard to do in C++
- multiple types can share private data & operations
  - requires `friend` declarations in C++
- one new type requires a name for the type (e.g. `T`)
  - class name is also type name in C++, conveniently

C++'s public/private is simpler than ML's separate signatures, but C++ doesn't have a simple way of describing just an interface