Currying

ML chooses the most general (least-restrictive) type possible for user-defined functions.

Functions are first-class objects, as in Scheme.

The function definition

```
fun f x y = expression;
```

defines a function **`f`** (of **`x`**) that returns a function (of **`y`**).

Reducing multiple argument functions to a sequence of one argument functions is called *currying* (after Haskell Curry, a mathematician who popularized the approach).

Thus

```
fun f x y = x :: [y];
val f = fn : 'a -> 'a -> 'a list
```

says that **f** takes a parameter **x**, of type **'a**, and returns a function (of **y**, whose type is **'a**) that returns a list of **'a**.

Contrast this with the more conventional

```
fun g(x,y) = x :: [y];
val g = fn : 'a * 'a -> 'a list
```

Here **g** takes a pair of arguments (each of type **'a**) and returns a value of type **'a** list.

The advantage of currying is that we can bind one argument and leave the remaining argument(s) free.

For example

```
f(1);
```

is a legal call. It returns a function of type

```
fn : int -> int list
```

The function returned is equivalent to

```
fun h b = 1 :: [b];
val h = fn : int -> int list
```

Map Revisited

ML supports the **map** function, which can be defined as

```
fun map(f,[]) = []
  | map(f,x::y) =
       (f x) :: map(f,y);
```

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

This type says that **map** takes a pair of arguments. One is a function from type **'a** to type **'b**. The second argument is a list of type **'a**. The result is a list of type **'b**.

In curried form **map** is defined as

```
fun map f [] = []
  | map f (x::y) =
       (f x) :: map f y;
```

```
val map =
 fn : ('a -> 'b) ->
       'a list -> 'b list
```

This type says that **map** takes one argument that is a function from type **'a** to type **'b**. It returns a function that takes an argument that is a list of type **'a** and returns a list of type **'b**.

The advantage of the curried form of **map** is that we can now use **map** to create "specialized" functions in which the function that is mapped is fixed.

For example,

```
val neg = map not;
```

```
val neg =
  fn : bool list -> bool list
```

```
neg [true,false,true];
```

```
val it = [false,true,false] :
 bool list
```

Power Sets Revisited

Let’s compute power sets in ML.

We want a function **pow** that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

```
pow [1,2] =  [[1,2],[1],[2],[]]
```

We first define a version of **cons** in curried form:

```
fun cons h t = h::t;
val cons = fn :
  'a -> 'a list -> 'a list
```

Now we define `pow`. We define the powerset of the empty list, `[]`, to be `[[]]`. That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of `h::t`, we compute the power set of `t`, which we call `pset`. Then the power set for `h::t` is just `h` distributed through `pset` appended to `pset`.

We distribute `h` through `pset` very elegantly: we just map the function `(cons h)` to `pset`. `(cons h)` adds `h` to the head of any list it is given. Thus mapping `(cons h)` to `pset` adds `h` to *all* lists in `pset`.

The complete definition is simply

```
fun pow [] = [[]]
 |   pow (h::t) =
     let
      val pset = pow t
     in
     (map (cons h) pset) @ pset
     end;
```

```
val pow =
  fn : 'a list -> 'a list list
```

Let's trace the computation of **pow [1,2]**.

Here **h** = **1** and **t** = **[2]**. We need to compute **pow [2]**.

Now **h** = **2** and **t** = **[]**.

We know **pow [] = [[]]**,
so **pow [2] =**
(map (cons 2) [[]])@[[]] =
([[2]])@[[]] = [[2],[]]

Therefore **`pow [1,2] =`**

```
(map (cons 1) [[2],[]])
 @[[2],[]] =
[[1,2],[1]]@[[2],[]] =
[[1,2],[1],[2],[]]
```

Composing Functions

We can define a composition function that composes two functions into one:

```
fun comp (f,g)(x) = f(g(x));
val comp = fn :
('a -> 'b) * ('c -> 'a) ->
 'c -> 'b
```

In curried form we have

```
fun comp f g x = f(g(x));
val comp = fn :
('a -> 'b) ->
('c -> 'a) -> 'c -> 'b
```

For example,

```
fun sqr x:int = x*x;
val sqr = fn : int -> int
comp sqr sqr;
val it = fn : int -> int
comp sqr sqr 3;
val it = 81 : int
```

In SML **o** (lower-case O) is the infix composition operator.

Hence

sqr o sqr $\equiv$ **comp sqr sqr**

Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses.

That notation is

```
fn arg => body;
```

For example,

```
val sqr = fn x:int => x*x;
val sqr = fn : int -> int
```

In fact the notation used to define functions,

```
fun name arg = body;
```

is actually just an abbreviation for the more verbose

```
val name = fn arg => body;
```

An anonymous function can be used wherever a function value is needed.

For example,

```
map (fn x => [x]) [1,2,3];
val it =
[[1],[2],[3]] : int list list
```

We can use patterns too:

```
(fn [] => []
  |(h::t) => h::h::t);
val it = fn : 'a list -> 'a list
```

(What does this function do?)

Polymorphism vs. Overloading

ML supports polymorphism.

A function may accept a polytype (a set of types) rather than a single fixed type.

In all cases, the same function definition is used. Details of the supplied type are irrelevant and may be ignored.

For example,

```
fun id x = x;
val id = fn : 'a -> 'a
fun toList x = [x];
val toList = fn : 'a -> 'a list
```

Overloading, as in C++ and Java, allows alternative definitions of the same method or operator, with selection based on type.

Thus in Java **+** may represent integer addition, floating point addition or string concatenation, even though these are really rather different operations.

In ML **+**, **-**, ***** and **=** are overloaded.

When **=** is used (to test equality), ML deduces that an *equality type* is required. (Most, but not all, types can be compared for equality).

When ML decides an equality type is needed, it uses a type variable that begins with two tics rather than one.

```
fun eq(x,y) = (x=y);
val eq = fn : ''a * ''a -> bool
```

Defining New Types in ML

We can create new names for existing types (type abbreviations) using

```
type id = def;
```

For example,

```
type triple = int*real*string;
type triple = int * real * string
type rec1=
 {a:int,b:real,c:string};
type rec1 =
  {a:int, b:real, c:string}
type 'a triple3  = 'a*'a*'a;
type 'a triple3 = 'a * 'a * 'a
type intTriple = int triple3;
type intTriple = int triple3
```

These type definitions are essentiality macro-like name substitutions.

The Datatype Mechanism

The **datatype** mechanism specifies new data types using *value constructors*.

For example,

```
datatype color = red|blue|green;
datatype color = blue | green |
red
```

Pattern matching works too using the type's constructors:

```
fun translate red = "rot"
  | translate blue = "blau"
  | translate green = "gruen";
val translate =
 fn : color -> string
fun jumble red = blue
  | jumble blue = green
  | jumble green = red;
val jumble = fn : color -> color
translate (jumble green);
val it = "rot" : string
```

SML Examples

Source code for most of the SML examples presented here may be found in

`~cs538-1/public/sml/class.sml`

Parameterized Constructors

The constructors used to define data types may be parameterized:

```
datatype  money =
    none
  | coin of int
  | bill of int
  | iou of real * string;
```

```
datatype money =
   bill of int | coin of int
 | iou of real * string | none
```

Now expressions like **coin(25)** or **bill(5)** or **iou(10.25,"Lisa")** represent valid values of type **money**.

We can also define values and functions of type **money**:

```
val dime = coin(10);
val dime = coin 10 : money
val deadbeat =
iou(25.00,"Homer Simpson");
val deadbeat =
 iou (25.0,"Homer Simpson") :
 money
fun amount(none) = 0.0
  |  amount(coin(cents)) =
      real(cents)/100.0
  |  amount(bill(dollars)) =
      real(dollars)
  |  amount(iou(amt,_)) =
      0.5*amt;
 val amount = fn : money -> real
```

Polymorphic Datatypes

A user-defined data type may be polymorphic. An excellent example is

```
datatype  'a option  =
 none | some of 'a;
```

```
datatype 'a option =
  none | some of 'a
```

```
val zilch = none;
```

```
val zilch = none : 'a option
```

```
val mucho =some(10e10);
```

```
val mucho =
some 100000000000.0 : real option
```

```
type studentInfo =
 {name:string,
  ssNumber:int option};
```

```
type studentInfo = {name:string,
ssNumber:int option}
```

```
val newStudent =
{name="Mystery Man",
 ssNumber=none}:studentInfo;
```

```
val newStudent =
{name="Mystery Man",
 ssNumber=none} : studentInfo
```

Datatypes may be Recursive

Recursive datatypes allow linked structures *without* explicit pointers.

```
datatype  binTree  =
   null
 | leaf
 | node of binTree * binTree;
```

```
datatype binTree =
 leaf | node of binTree * binTree
  | null
```

```
fun size(null) = 0
  |  size(leaf) = 1
  |  size(node(t1,t2)) =
       size(t1)+size(t2) + 1
```

```
val size = fn : binTree -> int
```

Recursive Datatypes may be Polymorphic

```
datatype 'a binTree  =
  null
| leaf of 'a
| node of 'a binTree * 'a binTree
```

```
datatype 'a binTree =
 leaf of 'a |
 node of 'a binTree * 'a binTree
 | null
```

```
fun frontier(null) = []
 |  frontier(leaf(v)) = [v]
 |  frontier(node(t1,t2)) =
     frontier(t1) @ frontier(t2)
```

```
val frontier =
  fn : 'a binTree -> 'a list
```

We can model n-ary trees by using lists of subtrees:

```
datatype 'a  Tree   =
   null
|  leaf of 'a
|  node of 'a Tree list;
```

```
datatype 'a Tree = leaf of 'a |
node of 'a Tree list | null
```

```
fun frontier(null) = []
  |   frontier(leaf(v)) = [v]
  |   frontier(node(h::t)) =
        frontier(h) @
        frontier(node(t))
  |   frontier(node([])) = []
```

```
val frontier = fn :
 'a Tree -> 'a list
```