Type synonyms

How can ML programmers define their own types?

One way: can give a new name to an existing type
- name and type are equivalent, interchangeable

```
- type person = {name:string, age:int};
type person = {age:int, name:string}

- val p:person = {name="Bob", age=18};
val p = {age=18,name="Bob"} : person

- val p2 = p;
val p2 = {age=18,name="Bob"} : person

- val p3:{name:string, age:int} = p;
val p3 = {age=18,name="Bob"}
       : {age:int, name:string}
```


Polymorphic type synonyms

Can define polymorphic synonyms

```
- type 'a stack = 'a list;
type 'a stack = 'a list

- val emptyStack:'a stack = nil;
val emptyStack = [] : 'a stack
```

Synonyms can have multiple type parameters:

```
- type (''key, 'value) assoc_list =
=      (''key * 'value) list;
type ('a,'b) assoc_list = ('a * 'b) list

- val grades:(string,int) assoc_list =
=   [("Joe", 84), ("Sue", 98), ("Dude", 44)];
val grades =
    [("Joe",84),("Sue",98),("Dude",44)]
  : (string,int) assoc_list
```


Datatypes

(How can ML programmers define their own types?)
Another way: can declare a new "algebraic data type"
- a new type, unlike a type synonym

Simple example: ML's version of enumerated types

```
- datatype sign = Positive | Zero | Negative;
datatype sign = Negative | Positive | Zero
```

Declares both a **type** (`sign`) and a set of alternative **constructor values** of that type (`Positive` etc.)
- order doesn't matter

```
- fun signum(x) =
=   if x > 0 then Positive
=   else if x = 0 then Zero
=   else Negative;
val signum = fn : int -> sign
```

Another example, from ML standard library: bool

```
- datatype bool = true | false;
datatype bool = false | true
```


Constructors with data

Each constructor can have data of particular type stored with it
- constructors are functions that
  allocate & initialize new values with that "tag"

Example:

```
- datatype LiteralExpr =
=   Nil |
=   Integer of int |
=   String of string;
datatype LiteralExpr =
  Integer of int | Nil | String of string

- Nil;
val it = Nil : LiteralExpr
- Integer(3);
val it = Integer 3 : LiteralExpr
- String("xyz");
val it = String "xyz" : LiteralExpr
```

Pattern-matching on datatypes

Constructor names can be used in patterns to test for values with that tag

- can use argument pattern to access data stored with that constructor tag

```
- fun signum_value(Positive) = 1
=   | signum_value(Zero)     = 0
=   | signum_value(Negative) = ~1;
val signum_value = fn : sign -> int

- fun toString(Nil) = "nil"
=   | toString(Integer(i)) = Int.toString(i)
=   | toString(String(s)) = "\"" ^ s ^ "\"";
val toString = fn : LiteralExpr -> string
```

Watch out for formal and constructor name ambiguity, e.g.:

```
- datatype T = x | y;
- fun f(x) = 0;
val f = fn : ... (* what's the type of f? *)
```


Recursive datatypes

Many datatypes are recursive:
one or more constructors are defined in terms of the datatype itself

```
- datatype Expr =
=   Nil |
=   Integer of int |
=   String of string |
=   Variable of string |
=   Tuple of Expr list |
=   BinOpExpr of {arg1:Expr,
=                 operator:string,
=                 arg2:Expr} |
=   FnCall of {function:string, arg:Expr};
datatype Expr = ...

(*  (3, "hi")  *)
- val expr1 = Tuple [Integer(3), String("hi")];
val expr1 = Tuple [Integer 3,String "hi"]
          : Expr
```

(**Shadows** previous `Nil`, `Integer`, and `String` bindings)

Another example expression value

```
(*  f(3+x, "hi")  *)
= val expr2 =
=  FnCall {
=    function="f",
=    arg=Tuple [
=      BinOpExpr {arg1=Integer(3),
=                 operator="+",
=                 arg2=Variable("x")},
=      String("hi")]};
val expr2 = ... : Expr
```


Recursive functions over recursive datatypes

Often manipulate recursive datatypes with recursive functions

- pattern of recursion in function matches pattern of recursion in datatype

```
- fun toString(Nil) = "nil"
=   | toString(Integer(i)) = Int.toString(i)
=   | toString(String(s)) = "\"" ^ s ^ "\""
=   | toString(Variable(name)) = name
=   | toString(Tuple(elems)) =
=         "(" ^ listToString(elems) ^ ")"
=   | toString(BinOpExpr{arg1,operator,arg2})=
=         toString(arg1) ^ " " ^ operator
=           ^ " " ^ toString(arg2)
=   | toString(FnCall{function,arg}) =
=         function ^ "(" ^ toString(arg) ^ ")"
= and listToString([]) = ""
=   | listToString([elem]) = toString(elem)
=   | listToString(e::es) =
=         toString(e) ^ "," ^ listToString(es);
val toString = fn : Expr -> string
val listToString = fn : Expr list -> string
```

Mutually recursive functions and datatypes

If two or more functions are defined in terms of each other, recursively, then must be declared together, and linked with `and`

E.g.

```
fun toString(...) = ... listToString ...
and listToString(...) = ... toString ...
```

If two or more mutually recursive datatypes, then declare them together, linked by `and`

E.g.

```
datatype Stmt = ... Expr ...
and Expr = ... Stmt ...
```


A convenience: record pattern syntactic sugar

Instead of writing `{a=a, b=b, c=c}` as a pattern, can write `{a,b,c}`

E.g.

```
... BinOpExpr{arg1,operator,arg2} ...
```

is short-hand for

```
... BinOpExpr{arg1=arg1,
              operator=operator,
              arg2=arg2} ...
```


Polymorphic datatypes

Datatypes can be polymorphic

```
- datatype 'a List = Nil
                   | Cons of 'a * 'a List;
datatype 'a List = Cons of 'a * 'a List | Nil

- val lst = Cons(3, Cons(4, Nil));
val lst = Cons (3,Cons (4,Nil)) : int List

- fun Null(Nil) = true
    | Null(Cons(_,_)) = false;
val Null = fn : 'a List -> bool

- fun Hd(Nil) = raise Empty
    | Hd(Cons(h,_)) = h;
val Hd = fn : 'a List -> 'a

- fun Sum(Nil) = 0
    | Sum(Cons(x,xs)) = x + Sum(xs);
val Sum = fn : int List -> int
```


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

In ML, `structure` declarations 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 add_first_team(name) =
  let
    open Assoc_List
      (* declares assoc_list, empty, store, etc. *)
    val init = {wins=0, losses=0}
  in
    store(empty, name, init)
    (* Assoc_List.store(
         Assoc_List.empty, name, init) *)
  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 as part of 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
```


An extended example: binary trees

Stores elements in sorted order

- enables faster membership testing, printing out in sorted order

```
datatype 'a BTree
   = EmptyBTree
   | BTNode of 'a * 'a BTree * 'a BTree
```


Some functions on binary trees

```
fun insert(x, EmptyBTree) =
       BTNode(x, EmptyBTree, EmptyBTree)
  | insert(x, n as BTNode(y,t1,t2)) =
       if x = y then n
       else if x < y then
          BTNode(y, insert(x, t1), t2)
       else
          BTNode(y, t1, insert(x, t2))

fun member(x, EmptyBTree) = false
  | member(x, BTNode(y,t1,t2)) =
       if x = y then true
       else if x < y then member(x, t1)
       else member(x, t2)
```

What are the types of these functions?

First-class functions

Can make code more reusable by parameterizing it by functions as well as values and types

Simple technique: treat functions as first-class values

- function values can be created, used, passed around, bound to names, stored in other data structures, etc., just like all other ML values

```
- fun int_lt(x:int, y:int) = x < y;
val int_lt = fn : int * int -> bool

- int_lt(3,4);
val it = true : bool

- val f = int_lt;
val f = fn : int * int -> bool

- f(3,4);
val it = true : bool
```

Passing functions to functions

A function can often be made more flexible
if takes another function as an argument

E.g.:

- parameterize binary tree insert & member functions by the = and < comparisons to use
- parameterize the quicksort algorithm by the < comparison to use
- parameterize a list search function by the search criterion

```
(* find(test_fn:'a -> bool, lst:'a list):'a *)
- exception NotFound;
- fun find(test_fn, nil) = raise NotFound
    | find(test_fn, elem::elems) =
        if test_fn(elem) then elem
        else find(test_fn, elems);
val find = fn : ('a -> bool) * 'a list -> 'a

- fun is_good_grade(g) = g >= 90;
val is_good_grade = fn : int -> bool
- find(is_good_grade, [85,72,92,98,84]);
val it = 92 : int
```


Binary tree functions, revisited

```
fun insert(x, EmptyBTree, eq, lt) =
      BTNode(x, EmptyBTree, EmptyBTree)
  | insert(x, n as BTNode(y,t1,t2), eq, lt) =
      if eq(x,y) then n
      else if lt(x,y) then
         BTNode(y, insert(x, t1, eq, lt), t2)
      else
         BTNode(y, t1, insert(x, t2, eq, lt))
val insert = fn
           : 'a * 'a BTree *
             ('a * 'a -> bool) *
             ('a * 'a -> bool) -> 'a BTree

fun member(x, EmptyBTree, eq, lt) = false
  | member(x, BTNode(y,t1,t2), eq, lt) =
      if eq(x,y) then true
      else if lt(x,y) then
         member(x, t1, eq, lt)
      else
         member(x, t2, eq, lt)
val member = fn
           : 'a * 'a BTree *
             ('a * 'a -> bool) *
             ('a * 'a -> bool) -> bool
```


Calling binary tree functions

```
- val t = insert(5, EmptyBTree, op=, op<);
val t = BTNode (5,EmptyBTree,EmptyBTree)
      : int BTree
- val t = insert(2, t, op=, op<);
val t = ...
- val t = insert(3, t, op=, op<);
- val t = insert(7, t, op=, op<);
- member(2, t, op=, op<);
val it = true : bool
- member(4, t, op=, op<);
val it = false : bool

- ... definitions of person type, person_eq & person_lt
functions, and p1 value
- val pt = insert(p1, EmptyBTree,
                  person_eq, person_lt);
val pt = ... : person BTree
```


Storing functions in data structures

It's a pain to keep passing around the `eq` and `lt` functions to all calls of `insert` and `member`

It's unreliable to depend on clients to pass in the right functions

Idea: store the functions in the tree itself

Storing functions in data structures (cont.)

```
local
  datatype 'a BT = EmptyBT
                 | BTNode of 'a * 'a BT * 'a BT
  fun ins(x, tree, eq, lt) = ... old insert ...
  fun mbr(x, tree, eq, lt) = ... old member ...
in
  datatype 'a BTree
    = BTree of {tree:'a BT,
                eq:'a * 'a -> bool,
                lt:'a * 'a -> bool}
  fun emptyBTree(eq,lt) =
    BTree{tree=EmptyBT, eq=eq, lt=lt}
  fun insert(x, BTree{tree, eq, lt}) =
    BTree{tree=ins(x, tree, eq, lt),eq=eq,lt=lt}
  fun member(x, BTree{tree, eq, lt}) =
    mbr(x, tree, eq, lt)
end
```

(`local ... in ... end` allows hiding some declarations while "exporting" others)

Records containing functions are SML's version of objects!

A common idiom: map

Idiom: 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 idiom

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

- [not quite the type of ML's `map`; stay tuned]

Example:

- have a list of fahrenheit temperatures for Seattle days
- want to give a list of temps to friend in England

```
- 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 idiom: filter

Idiom: 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 idiom

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

- [not quite the type of ML's `filter`; stay tuned]

Example:

- have a list of day temps
- want a list of nice day temps

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

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


Another common idiom: find

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

`find` captures this idiom

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

- [not quite the type of ML's `find`; stay tuned]

Example: find first nice day

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

Anonymous functions

Mapping and predicate functions often simple, only used once; don't merit their own name

Can directly write anonymous function **expressions**:
fn $pattern_{formal}$ **=>** $expr_{body}$

Call syntax allows arbitrary expression as function operand:
$expr_{fn}$ $expr_{arg}$

```
- 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` and `val rec` declarations are primitive notions
`fun` declarations are just a convenient syntax for `val` + `fn`

fun *f arg* = *expr*
is sugar for
val [**rec**] *f* = (**fn** *arg* => *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
- variables declared outside a scope are called "free variables" (w.r.t. that scope)

Makes nested functions a lot more useful in practice
- more than just hiding helper functions
- `map`, `filter`, `find` arguments often have free variables

Beyond what can be done with function pointers in C/C++
- C functions only have globals as free variables

Akin to inner classes in Java

Returning functions from functions

If functions are first-class,
then should be able to return them from other functions

Example: function composition

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

- fun square x = x*x; fun double y = y+y;
val square = fn : int -> int
val double = fn : int -> int
- val double_square = compose(double, square);
val double_square = fn : int -> int
- double_square 3;
val it = 18 : int
- (compose(square,double)) 3;
val it = 36 : int
```

The infix o operator is ML's predefined `compose`:

```
- map(square o double, [3,4,5]);
val it = [36,64,100] : int list
```


Currying

A curried function takes some arguments and then
computes & returns a function which takes additional arguments

The result function can be applied to many different arguments,
without having to pass in the first arguments again

Example: a curried version of `map`:

```
- fun map f =
    (fn nil => nil
      | x::xs => f x :: map f xs);
val map = fn : ('a->'b) -> 'a list -> 'b list

- map square [3,4,5];  (* left-to-right assoc. *)
val it = [9,16,25] : int list

- val squares = map square;  (* partial application *)
val squares = fn : int list -> int list
- squares [3,4,5];
val it = [9,16,25] : int list
- squares [9,10];
val it = [81,100] : int list
```


Clean syntactic sugar for currying

Allow multiple formal argument patterns ⇒ curried function

```
- fun map f  nil    = nil
    | map f (x::xs) = f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list

- fun filter pred  nil    = nil
    | filter pred (x::xs) =
       let val rest = filter pred xs in
       if pred x then x::rest else rest end;
val filter =
  fn : ('a->bool) -> 'a list -> 'a list

- fun find pred  nil    = raise NotFound
    | find pred (x::xs) =
        if pred x then x else find pred xs;
val find = fn : ('a->bool) -> 'a list -> 'a
```

Curried is the normal way to define ML functions

- syntactically cleaner than using argument tuples
- semantically more flexible

ML's predefined `map`, `filter`, and `find` are defined like this

A general idiom: fold

Abstracts the general case of recursive traversal over lists

Recursive list traversal idiom:

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

Parameters of this idiom, 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`)

`fold` captures this idiom

```
foldl,foldr:('a*'b -> 'b) -> 'b -> 'a list -> 'b
```

- 3 curried arguments
- iterate over elements left-to-right: `foldl`
- iterate over elements right-to-left: `foldr`
  - for associative combining operators, order doesn't matter
- [which is the recursive traversal idiom above?]

Examples using fold

```
foldl/foldr:('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 =
    foldl (fn(rain,subtotal)=>rain+subtotal)
          0.0 rainfall;
val total_rainfall = 4.0 : real
```

What do these do?

```
- foldl (fn(x,ls)=>x::ls) nil [3,4,5];

- foldr (fn(x,ls)=>x::ls) nil [3,4,5];

- foldr (fn(x,ls)=>x::ls) [1,2,3] [4,5,6];
```


First-class procedures and scoping

Invoking a function with free variables is tricky!

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

- val double_square = compose(double, square);
- val square_double = compose(square, double);

- double_square(3);
val it = 18 : int
- square_double(3);
val it = 36 : int
```

Same code invoked in both calls, so how are they distinguished?

- where do bindings for `f` and `g` come from?

Many anonymous `fn` args (to `map` et al.) and all non-trivial curried functions have free variables like this, so important to support

- allow these idioms, allow static typechecking, etc.

Lexical/static vs. dynamic scoping

Lexical/static scoping:
free references in nested function are resolved when the nested function is **created**,
based on lexically enclosing bindings

- programmer & typechecker can tell statically what each reference will resolve to
- need some way to remember bindings when creating the function

Dynamic scoping:
free references in nested function are resolved when **evaluated** dynamically, by looking at bindings in caller and the rest of the dynamically enclosing call stack

- easy to implement, in interpreter
- can't compute statically ⇒
  - can't statically typecheck!
  - hard to reason about as human!
  - can't compile efficiently!
- sometimes useful, e.g. floating-point rounding modes/ precision/..., exception handlers

Closures

An implementation technique supporting lexical scoping for first-class nested functions

A **closure** is a pair of a code address and an **environment**

- environment records bindings of free variables when function value was created
  - the function value is now self-contained
- if function has arbitrary lifetime, the closure, and therefore the environment, must be heap-allocated
  - no stack-allocated stack frames!

Many variations, differing in the details

Used with most high-level languages, e.g. ML, Scheme, Haskell, Smalltalk, Cecil, ...

Java's inner class instances look a lot like closures...

Restricted versions

Can use cheaper implementation strategies if language only supports restricted version of first-class functions

E.g. only allow nested functions to be passed down, not returned
- environment can be stack-allocated, not heap-allocated
- e.g. Pascal, Modula-3

E.g. allow nested procedures but not first-class procedures
- do not need pair, just extra implicit environment argument
- e.g. Ada

E.g. allow first-class procedures but no nesting
- implement with just a code address a.k.a. function pointer
- e.g. C, C++