Type synonyms

Can give a name to a type, for convenience

- 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

Users can define their own (polymorphic) data structures

- a new type, unlike type synonyms

Simple example: ML's version of enumerated types

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

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
```

Values can be used in patterns, too

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


Datatypes 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

The only way to access components of a value of a datatype is via pattern-matching

Constructor "calls" can be used in patterns to test for and take apart values with that "tag"

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


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
```

(`Nil`, `Integer`, and `String` of `LiteralExpr` are **shadowed**)

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

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 ...
```


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
- exception Empty;
exception Empty
- 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
```


An example: a very general tree datatype

Design:

- two kinds of non-empty trees: leaf nodes or interior nodes
  - an interior node has a list of children trees
- also have an empty tree
- interior nodes store some data, of type `'a`
- leaf nodes store some data, of type `'b`

Binary trees

A special kind of tree that stores elements in sorted order

- enables faster membership testing, printing out in sorted order

One way: make up fresh datatype for binary trees

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

Another way: reuse the general tree datatype

```
type 'a BTree = ('a,unit) Tree
```

Reuse is good if there are functions on general trees that we can reuse

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 sometimes be made more flexible if takes a 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 pattern being searched for

```
(* 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

```
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

```
datatype 'a BT
   = EmptyBT
   | BTNode of 'a * 'a BT * 'a BT
fun ins(x, tree, eq, lt) = ... previous insert ...
fun mbr(x, tree, eq, lt) = ... previous member ...

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)
```

Records containing functions are ML's version of objects!