Polymorphism

- Default type.
  Example: `fun f(x) = x + x; f` is of type `int -> int`.

- The ability of a function to allow arguments of different types is called polymorphism.
  Examples:
  - `hd : 'a list -> 'a`.
  - `fun id(x) = x; id : 'a -> 'a`
  What is the type of `(id,id)`?

- Functions that restricts polymorphism:
  - `+`, `-`, `*`, `~`
  - `/`, `div`, `mod`
  - `<`, `<=`, `>=`, `>`
  - `andalso`, `orelse`, `not`
  - string concatenation.
  - Conversion type operators: `ord`, `chr`, `real`, `str`, `floor`, `ceiling`, `round`, `truncate`.

Equality operators

- Type variables whose values are restricted to be an equality type are distinguished by having names that begin with two quote marks rather than only one ("a).

- Examples:
  ```
  (1,2) = (2,3);
  val it = false : bool
  [1,2] = [1,2,3];
  val it = false : bool
  [1,2] <> [1,2,3];
  val it = true : bool
  1.2 = 1.2;
  stdIn:1.1-2.5 Error: operator and operand don't agree
  [equality type required]
  operator domain: "'Z * 'Z
  operand: real * real
  in expression:
  1.2 = 1.2
  ```

Pattern matching and equality operators

- Pattern matching permits us to design more general functions.

- Example: Can we use the function `reverse` (previously defined) to reverse a list of reals? No.

  ```
  fun reverse(L) = 
  if L = nil then nil
  else reverse(tl(L)) @ [hd(L)];
  val reverse = fn : `'a list -> `'a list
  ```

  But `reverse1` can be used thanks to pattern matching.

  ```
  fun reverse1(nil) = nil |
  reverse1(x::L) = reverse(L) @ [x];
  val reverse1 = fn : `'a list -> `'a list
  ```
Curried functions

- Until now we defined functions having only one parameter that is a tuple. A curried function is a function having several parameters (a list of parameters with no parentheses or commas).

- Example 1:
  map (built-in function in SML) is the curried version of apply. The type of map is:
  \((\text{"a} \rightarrow \text{\textquotesingle b\textquoteright \text{"}} \rightarrow \text{\textquotesingle a\textquoteright list \rightarrow \text{'b list}}\)).

  ```
  fun apply(f,l) = 
  if (l=[]) then []
  else f(hd(l));(apply(f,t1(l)));
  val apply = fn : (\text{\textquotesingle \text{\textquotesingle a \rightarrow \text{'b}} \text{'\text{"}} \text{\textquotesingle a\textquoteright list \rightarrow \text{'b list}}

- Example 2:
  Let us consider the function \(\text{exp}\) that computes \(x^y\) (\(x\) is a real and \(y\) is an int). \(\text{exp1}\) is the curried form of \(\text{exp}\).

  ```
  fun exp(x,0) = 1.0
  | exp(x,y) = x * exp(x,y-1);
  val exp : fn: real * int \rightarrow real
  fun exp1 x 0 = 1.0
  | exp1 x y = x * exp1 x y-1;
  val exp1 : fn: real \rightarrow int \rightarrow real

Let and Functions

- One can use let in expressions to allow us to use common sub-expressions.

- Examples:

  ```
  fun hundredthpower(x:real) = 
  let val four = x*x*x*x;
  val twenty = four*four*four*four
  in 
  twenty*twenty*twenty*twenty end;
  val hundredthpower = fn : real \rightarrow real

  fun fib(n)=
  if ((n = 0) or else (n = 1)) then 1
  else let
  val n1 = fib(n-1);
  val n2 = fib(n-2)
  in n1 + n2 end;

```

Let and Pattern matching

- In SML it is possible to split apart the values returned by a function.

- Example:

  ```
  fun f(x) = (x,x,x);
  val f = fn : 'a \rightarrow 'a * 'a * 'a

  val (a,b,c) = f(3);
  val a = 3 : int
  val b = 3 : int
  val c = 3 : int

  The variable a is linked with 3, the variable b is linked with 3 and the variable c is linked with 3.
  val t = f(3);
  val t = (3,3,3) : int * int * int
```
• This can be used in functions.

    fun g(nil) = 0
    | g(L) =
      let
        val (a,b,c) = f(hd(L)*2)
      in a
      end;
    val g = fn : int list -> int

    g(nil);
    val it = 0 : int
    g([4,6,7]);
    val it = 8 : int

### Datatypes

• SML has also a very powerful mechanism for defining new types called **datatypes**.

• A datatype definition involves:
  
  - A **type constructor** that is the name of the datatype.
  
  - One or more **data constructors**, which are identifiers used as operators to build values belonging to a new datatype.

• Functions having parameters of user defined datatypes are generally defined using **pattern matching**.

• Syntax:

  datatype (list of type parameters) <identifier> =
  <first constructor expression> |
  <second constructor expression> |
  ...
  <last constructor expression>;

### Enumerated types

• We can build new types in ML using "old" types using:

  - the "product of types" T1*T2
  
  - function types: T1 -> T2
  
  - list

Examples:

    type intpair = int * int;
    (5,7);
    val it = (5,7) : int * int
    (5,7): intpair;
    val it = (5,7) : intpair
    type funomint = int -> int;
    type intlist = int list;

• Parametrized type definitions

  Syntax:

    type (list of parameters) <identifier> = <type expression>;

Examples:

    type ('a,'b) relation = ('a * 'b) list;
    val R = [(1,2),(2,3)] : (int,int)relation;

  datatype primary_color = Red | Blue | Green;
  datatype primary_color = Red | Blue | Green
  datatype primer_color = Yellow | Magenta | Cyan;
  datatype primer_color = Yellow | Magenta | Cyan
  datatype fruit = Apple | Pear | Grape;
  datatype fruit = Apple | Pear | Grape
  datatype bin = zero | one;
  datatype bin = one | zero

  fun isApple(x) = (x=Apple);
  val isApple = fn : fruit -> bool
 Constructors in datatype definitions and functions

- Example:

  ```sml
datatype Nat = Zero | Suc of Nat;

  The constructors are Zero and Suc. The constructor Suc takes one parameter. Zero does not take parameters. This datatype is recursive: Nat is used in the rhs of its own definition.
```

- Define the Addition of 2 Nats, the multiplication of 2 Nat, the conversion from a Nat to an int and from an int to a Nat (curried forms).

  ```sml
  fun plus n Zero = n
  | plus n (Suc(p)) = Suc(suc n p);

  fun times n Zero = Zero
  | times n (Suc(p)) = plus (times n p) n;

  val plus = fn : Nat -> Nat -> Nat
  val times = fn : Nat -> Nat -> Nat
  val it = Suc (Suc Zero) : Nat
  ```

Binary Search Trees

A binary search tree is a binary tree where each node contains a key such that:

- All keys in the left subtree precede the key in the root.
- All keys in the right subtree succeed the key in the root.
- The left and right subtrees of the root are again binary search trees.

Binary Search Trees in SML

The `datatype` constructor in SML permits us to define new data types, like a `BinaryTree`:

```sml
  - datatype 'a BinaryTree = bempty |
    = bt of 'a * 'a BinaryTree * 'a BinaryTree ;

  val bt : 'a BinaryTree
  = bempty : 'a BinaryTree

  Each BinaryTree consists of a header `bt`, a node value or `key`, and left / right subtrees, both of which must be BinaryTree.

  The special case of a tree with no keys in it, the empty tree, is denoted `bempty`.

  ```sml
  - val Tree = bt (2,bempty,b)
  - bt(5,bt(3,bt(2,bempty,bempty))
  - bt(6,bt(3,bt(2,bempty,bempty)))
  - bempty
  ```
```
The lookup Function

To look up an element in a binary search tree, we must test the label of the root against our desired key, and the recursively search the appropriate side’s subtree:

- fun lookup (bempty,_) = false
  = | lookup(bt(root:int,left,right),x:int) =
  =  if (x = root) then true
  =  else (if (x <= root) then lookup(left,x)
  =    else lookup(right,x) );
- val lookup = fn : int BinaryTree * int -> bool

The search ends in failure whenever the subtree is bempty.

- lookup(Tree,6);
  val it = true : bool
- lookup(Tree,1);
  val it = false : bool
- lookup(Tree,9);
  val it = false : bool
- lookup(Tree,8);
  val it = true : bool
- lookup(bempty,6);
  val it = false : bool

Example

```
+   *
 2 5 3 4 1 6
```

- val Expression =
  = bt("+",
    - bt("*",
      - bt("2",bt_empty,bt_empty),
      - bt("5",bt_empty,bt_empty) ),
      - bt("*",
        - bt("3",bt_empty,bt_empty),
        - bt("4",bt_empty,bt_empty) ) ),
    - bt("*",
      - bt("1",bt_empty,bt_empty),
      - bt("6",bt_empty,bt_empty) ) );
- val Expression = bt ("+", bt ("*",bt #,bt #),bt ("*",bt #,bt #)) : string BinaryTree

Tree Traversal in ML

Note how only the order of the recursive calls changes in the three traversals.

- fun inorder (bt_empty) = []
  | inorder(bt(root:'a,left,right)) =
  | inorder(left) @ (root :: inorder(right));
- fun preorder (bt_empty) = []
  | preorder(bt(root:'a,left,right)) =
  | (preorder(left) @ preorder(right)) @ (root :: []);
- fun postorder (bt_empty) = []
  | postorder(bt(root:'a,left,right)) =
  | (postorder(left) @ postorder(right)) @ (root :: []);

These traversal functions could be easily modified to compute the value of the arithmetic tree instead of just returning the formula.
Inorder Traversals of Binary Search Trees

Why is an in-order traversal called in order?

Note that an in-order traversal of a binary search tree lists all the keys in alphabetical order:

```haskell
val it = [2,3,5,6,7,8] : int list
```

```haskell
val Tree3 = bt(7, bt(4, bt_empty, bt_empty),
    bt(12, bt_empty, bt_empty));

val Tree4 = bt("*",
    bt("/",
        bt("-", bt("7", bt_empty, bt_empty),
            bt("a", bt_empty, bt_empty) ),
        bt("5", bt_empty, bt_empty) ),
    bt("*", bt("a", bt_empty, bt_empty),
        bt("b", bt_empty, bt_empty) ),
    bt("3", bt_empty, bt_empty) ));

val Expression = bt("+",
    bt("*",
        bt("-",
            bt("2", bt_empty, bt_empty),
                bt("5", bt_empty, bt_empty) ),
        bt("-",
            bt("3", bt_empty, bt_empty),
                bt("4", bt_empty, bt_empty) ),
        bt("-",
            bt("1", bt_empty, bt_empty),
                bt("6", bt_empty, bt_empty) ) ));
```

```haskell
(* Binary tree processing *)
datatype 'a BinaryTree = bt_empty |
    bt of 'a * 'a BinaryTree * 'a BinaryTree;

fun left_subtree bt_empty = bt_empty |
    left_subtree(bt(_,left,)) = left;

fun right_subtree bt_empty = bt_empty |
    right_subtree(bt(_,right,)) = right;

eception label_has_nil_argument;

fun label bt_empty = raise label_has_nil_argument |
    label(bt(value,_,_)) = value;

(* Sample binary trees *)
val Tree = bt(2, bt_empty, 
    bt(3, bt_empty, 
        bt(7, bt(6, bt(5, bt_empty, bt_empty), 
            bt_empty),
        bt(8, bt_empty, bt_empty))
    )
);

val Tree1 = bt(3, bt_empty, bt_empty);
val Tree2 = bt(5, bt(1, bt_empty, bt_empty), bt_empty);

inorder(Tree);
preorder(Tree);
postorder(Tree);
inorder(Expression);
preorder(Expression);
postorder(Expression);

lookup(Tree,6);
lookup(Tree,1);
lookup(Tree,8);
lookup(Tree,9);
lookup(bt_empty,6);
```
- Array is a structure in SML that gives us the ability to create and manipulate arrays.

The structure is opened using:

```
open Array;
```

- opening Array

```sml
val modify : ('a -> 'a) -> 'a array -> unit
val appi : (int * 'a -> unit) -> 'a array * int * int option -> unit
val foldl1 : (int * 'a * 'b -> 'b) -> 'b -> 'a array * int * int option -> 'b
val foldr : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
```

- Example:

```
val A = array(10,4);
val l = length(A);
val l = 10 : int
val v = sub(A,3);
val v = 4 : int
update(A,5,0);
val it = [4,4,4,4,4,0,4,4,4,4] : int array
```

---

**Records**

- Aggregate structures, where each component has a name.

- Example:

```
type king = {name : string, born : int, crowned : int, died : int};

val HenryV = {
  name = "Henri",
  born = 1387,
  crowned = 1413,
  died = 1422
};
```

- Accessing components in a record:
  - by pattern matching:
    ```
    val {born = x, ...} = HenryV;
    val x = 1387 : int
    ```
  - by operators:
    ```
    val byear = #born(HenryV);
    val byear = 1387 : int
    ```

- Functions over records:

```
fun livetime(k:king) = (#died(k)) - (#born(k));
val livetime = fn : king -> int
livetime(HenryV);
val it = 35 : int
```