**Polymorphism**

- 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)`?

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

- Functions that restrict 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?

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

  reverse cannot be used. But `reverse1` can be used thanks to pattern matching.
  ```

  ```
  fun reverse1(nil) = nil
  reverse1(x::L) = reverse1(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 `('a -> 'b) -> 'a list -> 'b list`.

  ```sml
  fun apply(f,L) = 
    if (L=[]) then []
  else f(hd(L))::(apply(f,tl(L)));
  val apply = fn : ('a -> 'b) * 'a list -> 'b list
  ```

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

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

Exceptions

- Many functions are **partial**, they do not produce a value for some of the possible arguments of the function's domain type. It is essential that we be able to catch such errors. This is done by generations and handling of exceptions.

- Examples of predefined exceptions in SML:

  ```sml
  5 div 0;
  uncaught exception Div
  
  hd(nil);
  uncaught exception Empty
  ```

Partially instantiated functions

- Curried functions are useful to construct new functions by applying the function to arguments for some but not all of its parameters.

- Examples:

  ```sml
  exp1 3.0 is a function of type int -> real that computes the powers of 3.0.
  exp1 10.0 is a function of type int -> real that computes the powers of 10.0
  ```

Exceptions

- Declaration of an exception

  ```sml
  exception < name of the exception > (of < type >)
  ```

- Examples:

  ```sml
  exception Foo;
  exception Alert of string;
  exception Number of int;
  exception Tuple of int*real;
  ```

- The type of an exception is `exn`.

- Raising an exception

  ```sml
  raise < name of the exception( < parameters > ) >
  ```

- Examples:

  ```sml
  raise Foo
  raise Alert("Stop!")
  raise Number(5)
  raise Tuple(1,5)
  ```
Handling an exception

Syntax:

\(<\text{expression}>\) handle <match>

Examples:

Design a function \(g(n,m)\) that prints "Division by 0" in case \(m = 0\) using exceptions.

```plaintext
exception Division of int;
exception Division of int

fun f(n,m) =
  if m = 0
  then raise Division(0)
  else n div m;
val f = fn : int * int -> int

fun g(n,m) = f(n,m) handle
Division(0) => (print "Division by 0"; print "n"; 0);
val g = fn : int * int -> int

g(1,0);
Division by 0
val it = 0 : int
g(2,1);
val it = 2 : int
```

Local environment

- It is possible to create local variables inside a function using \(let \ldots end\).

Syntax:

```plaintext
let
val <var1> = <val1>;
val <var2> = <val2>;
...
val <varn> = <valn>
in
<expression>
end
```

Example:

\(\rho_1 = \{(x,1), (y,2), (z,3)\}\)

```plaintext
let val u=x*y; val v=x*y+x; val z=1
in u*z+v*z
end;
val it = 9 : int
```

```
stdIn:60.1 Error: unbound variable or constructor: u
```

\(\rho_2 = \rho_1 \cup \{(u,3), (v,5), (z,1)\}\)

Variables and environment

- A variable is represented by a name and a value.

- The environment is a list of pairs \((variable,\ value)\).
  \(\rho = \{(v,\ val) \mid v\ is\ a\ variable\ and\ val\ in\ Value(v)\}\)

- A new variable may be added to the environment using:

  ```plaintext
  val <variable> = <value>;
  ```

Example:

```plaintext
Initial environment: \(\rho_0\)
val x = 1;
val y = 3;
val z = y+x+3;
Environment: \(\rho_1 = \rho_0 \cup \{(x,1), (y,3), (z,7)\}\)
val y = 1.2;
Environment: \(\rho_2 = \rho_1 \cup \{(y,1.2)\}\)
```

Functions

- One can use \(let\) in expressions to allow us to use common subexpressions.

Examples:

```plaintext
fun hundredpower(x:real) =
  let val four = x*x*x*x;
  val twenty = four*four*four*four
  in twenty*twenty*twenty*twenty*twenty
  end;
val hundredpower = fn : real -> 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;
```
Pattern matching

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

- Example:

```sml
fun f(x) = (x,x,x);  
val f = fn : 'a -> '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 = (3,3,3);     
val t = (3,3,3) : int * int * int
```

User defined types

- We can build new types in ML using "old" types using:
  - the "product of types" T1*T2
  - function types: T1 -> T2
  - list

Examples:

```sml
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:

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

Examples:

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

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 easily using pattern matching.

- Syntax:

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

- An **enumerated datatype** declaration consists of the keyword `datatype`, a name for the datatype, and a list of data constructors separated by `|`.

- Examples:

  ```haskell
datatype primary.color = Red | Blue | Green;
datatype primary.color = Red | Blue | Green
datatype printer.color = Yellow | Magenta | Cyan;
datatype printer.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
```

```
fun natint Zero = 0
| natint (Suc(n)) = natint(n) + 1;
val natint = fn : Nat -> int
val it = 3 : int

fun intnat 0 = Zero
| intnat n = Suc(intnat(n-1));
val intnat = fn : int -> Nat
val it = Suc (Suc (Suc Zero)) : Nat
```

Constructors in datatype definitions and functions

- Example:

  ```haskell
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 Nats, the conversion from a Nat to an int and from an int to a Nat (curried forms).

  ```haskell
  fun plus n Zero = n
  | plus n (Suc(p)) = Suc(plus p n);
  val plus = fn : Nat -> Nat -> Nat
  fun plus (Suc(Zero)) (Suc(Zero))
  val it = Suc (Suc Zero) : Nat

  fun times n Zero = Zero
  | times n (Suc(p)) = plus (times n p) n ;
  val times = fn : Nat -> Nat -> Nat
  times (Suc(Zero)) (Suc(Zero))
  val it = 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.

```
      3
     / \
    2  6
   /   /
  1  5  8
```

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 ;

con bt : 'a * 'a BinaryTree * 'a BinaryTree ->
    'a BinaryTree
con 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,
    bt(3, bempty,
        bt(7, bt(6, bt(bempty, bempty),
            bempty),
        bempty)) );
val Tree = bt(2, bempty, bt(3, bempty, bt(#))): int
```

Tree Traversal in ML

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

```ml
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)) =
        root :: (preorder(left) @ preorder(right));

fun postorder (bt.empty) = []
    | postorder(bt(root:a,left,right)) =
        (postorder(left) @ postorder(right)) @ (root :: []);
```

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:

```sml
fun lookup (bt, x) = 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.

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

Example

```
- 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
```
- inorder(Expression);  
val it = ["2","+","5","*","3","*","4","+","1","*","6"]
- preorder(Expression);  
val it = ["+","*","+","2","5","*","3","4","*","1","6"]
- postorder(Expression);  
val it = ["2","5","+","3","4","*","1","6","*","+"]

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:
- inorder(Tree);  
val it = [2,3,5,6,7,8] : int list

(* 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;

exception label_has_no_argument ;

fun label bt_empty = raise label_has_no_argument  
| label(bt(value,..)) = value;

(* Sample binary trees *)

val Tree = bt(2, bt_empty,  
bt(3, bt_empty,  
bt(7, bt(6, 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);

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("exp",  
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))  
);

lookup(Tree,6);  
lookup(Tree,4);  
lookup(Tree,8);  
lookup(Tree,9);  
lookup(bt_empty,6);  
lookup(bt_empty,6);
Arrays

- Array is a structure in SML that gives us the ability to create and manipulate arrays.
  The structure is opened using:

  ```sml
  open Array;
  ```

- opening Array
  ```sml
type 'a array = 'a ?.array
  type 'a vector = 'a ?.vector
  val max: int
  val array : int * 'a -> 'a array
  val tabulate : int * (int -> 'a) -> 'a array
  val fromList : 'a list -> 'a array
  val length : 'a array -> int
  val sub : 'a array * int -> 'a
  val update : 'a array * int * 'a -> unit
  val extract : 'a array * int * int option -> 'a vector
  val copy: di:int, dst:'a array, len:int option, si:int, src:'a array -> unit
  val copyVec:di:int, dst:'a array, len:int option, si:int, src:'a vector -> unit
  val app : ('a -> unit) -> 'a array -> unit
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
  ```

val modify : ('a -> 'a) -> 'a array -> unit
val appi : (int * 'a -> unit) -> 'a array * int * int option -> unit
val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a array * int * int option -> 'b
val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a array * int * int option -> 'b
val modifyi : (int * 'a -> 'a) -> 'a array * int * int option -> unit

- Example:

  ```sml
  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); A;
  val it = [|4,4,4,4,0,4,4,4,4,4|] : int array
  ```

Records

- Aggregate structures, where each component has a name.

- Example:

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

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

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

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