• *Function evaluation* is the basic concept for a programming paradigm that has been implemented in *functional programming languages*.

• The language ML ("Meta Language") was originally introduced in the 1970's as part of a theorem proving system, and was intended for describing and implementing proof strategies. Standard ML of New Jersey (SML) is an implementation of ML.

• The basic mode of computation in SML, as in other functional languages, is the use of the *definition* and *application* of functions.

• The basic cycle of SML activity has three parts:
  – *read* input from the user,
  – *evaluate* it, and
  – *print* the computed value (or an error message).
First SML example

• Here is a simple example:

   - 3;
   val it = 3 : int

• The first line contains the SML prompt, followed by an expression typed in by the user and ended by a *semicolon*.

• The second line is SML’s response, indicating the value of the input expression and its *type*. 
Interacting with SML

• SML has a number of built-in operators and data types.

• For instance, it provides the standard arithmetic operators.

  - 3+2;
  val it = 5 : int
  - sqrt(2.0);
  val it = 1.41421356237309 : real

• The Boolean values true and false are available, as are logical operators such as not (negation), andalso (conjunction), and orelse (disjunction).

  - not(true);
  val it = false : bool
  - true andalso false;
  val it = false : bool
Types in SML

• SML is a *strongly typed* language in that all (well-formed) expressions have a type that can be determined by examining the expression.

• As part of the evaluation process, SML determines the type of the output value using suitable methods of *type inference*.

• Simple types include `int`, `real`, `bool`, and `string`.

• One can also associate identifiers with values,

```
  - val five = 3+2;
  val five = 5 : int
```

and thereby establish a new *value binding*,

```
  - five;
  val it = 5 : int
```
Function Definitions in SML

- The general form of a function definition in SML is:
  
  \[
  \text{fun } \langle \text{identifier} \rangle \left( \langle \text{parameters} \rangle \right) = \langle \text{expression} \rangle;
  \]

- For example,
  
  - `fun double(x) = 2*x;`
  
  \[
  \text{val double = fn : int } \rightarrow \text{ int}
  \]
  
  declares `double` as a function from integers to integers, i.e., of type `\text{int } \rightarrow \text{ int}`.

  - `double(222);`

  \[
  \text{val it = 444 : int}
  \]

- Apply a function to an argument of the wrong type results in an error message:

  - `double(2.0);`
  
  Error: operator and operand don’t agree ...

- The user may also `explicitly` indicate types

  - `fun max(x:int,y:int,z:int) =`  
    
  \[
  = \text{ if } ((x>y) \text{ andalso } (x>z)) \text{ then } x
  = \text{ else } (\text{if } (y>z) \text{ then } y \text{ else } z);
  \]

  \[
  \text{val max = fn : int } \ast \text{ int } \ast \text{ int } \rightarrow \text{ int}
  \]

  - `max(3,2,2);`

  \[
  \text{val it = 3 : int}
  \]
Recursive Definitions

- The use of recursive definitions is a main characteristic of functional programming languages, and these languages encourage the use of recursion over iterative constructs such as while loops.

- Example

  - fun factorial(x) = if x=0 then 1
    = else x*factorial(x-1);
  val factorial = fn : int -> int

  - The definition is used by SML to evaluate applications of the function to specific arguments.

    - factorial(5);
      val it = 120 : int
    - factorial(10);
      val it = 3628800 : int
The greatest common divisor (gcd) of two positive integers can be defined recursively based on the following observations:

1. $gcd(n,n) = n$,
2. $gcd(m,n) = gcd(n,m)$, and
3. $gcd(m,n) = gcd(m-n,n)$, if $m > n$.

These identities suggest the following recursive definition:

```plaintext
fun gcd(m,n):int = if m=n then n
else if m>n then gcd(m-n,n)
else gcd(m,n-m);
```

val gcd = fn : int * int -> int

- gcd(12,30);
val it = 6 : int
- gcd(1,20);
val it = 1 : int
- gcd(125,56345);
val it = 5 : int
**Tuples in SML**

- In SML *tuples* are finite sequences of arbitrary but fixed length, where different components need not be of the same type.

**Examples**

- `val t1 = (1,2,3);`
  
  `val t1 = (1,2,3) : int * int * int`

- `val t2 = (4,(5.0,6));`
  
  `val t2 = (4,(5.0,6)) : int * (real * int)`

- The components of a tuple can be accessed by applying the built-in functions `#i`, where `i` is a positive number.
  
  - `#1(t1);`
    
    `val it = 1 : int`
  
  - `#1(t2);`
    
    `val it = 4 : int`

- `#2(t2);`
  
  `val it = (5.0,6) : real * int`

- `#2(#2(t2));`
  
  `val it = 6 : int`

- If a function `#i` is applied to a tuple with fewer than `i` components, an error results.
Lists in SML

• A list in SML is a finite sequence of objects, all of the same type.

• Examples
  - [1,2,3];
    val it = [1,2,3] : int list
  - [true,false,true];
    val it = [true,false,true] : bool list
  - [[1,2,3],[4,5],[6]];
    val it = [[1,2,3],[4,5],[6]] : int list list

  The last example is a list of lists of integers.

• All objects in a list must be of the same type:
  - [1,[2]];
    Error: operator and operand don’t agree

• An empty list is denoted by one of the following expressions:
  - [];
    val it = [] : 'a list
  - nil;
    val it = [] : 'a list

• Note that the type is described in terms of a type variable 'a. Instantiating the type variable, by types such as int, results in (different) empty lists of corresponding types.
Operations on Lists

- SML provides various functions for manipulating lists.

- The function $\text{hd}$ returns the first element of its argument list.
  
  - $\text{hd}[1,2,3]$;
  val it = 1 : int
  - $\text{hd}[[1,2],[3]]$;
  val it = [1,2] : int list

  Applying this function to the empty list will result in an error.

- The function $\text{tl}$ removes the first element of its argument lists, and returns the remaining list.
  
  - $\text{tl}[1,2,3]$;
  val it = [2,3] : int list
  - $\text{tl}[[1,2],[3]]$;
  val it = [[3]] : int list list

  The application of this function to the empty list will also result in an error.
More List Operations

- Lists can be constructed by the (binary) function :: (read cons) that adds its first argument to the front of the second argument.

  - 5::[];
  val it = [5] : int list
  - 1::[2,3];
  val it = [1,2,3] : int list
  - [1,2]::[[3],[4,5,6,7]];
  val it = [[1,2],[3],[4,5,6,7]] : int list list

  Again, the arguments must be of the right type:

    - [1]::[2,3];
    Error: operator and operand don't agree

- Lists can also be compared for equality:

  - [1,2,3]=[1,2,3];
  val it = true : bool
  - [1,2]=[2,1];
  val it = false : bool
  - tl[1] = [];
  val it = true : bool
Defining List Functions

• Recursion is particularly useful for defining functions that process lists.

• For example, consider the problem of defining an SML function that takes as arguments two lists of the same type and returns the concatenated list.

• In defining such list functions, it is helpful to keep in mind that a list is either
  – an empty list or
  – of the form x::y.
In designing a function for concatenating two lists \( x \) and \( y \) we thus distinguish two cases, depending on the form of \( x \):

- If \( x \) is an empty list, then concatenating \( x \) with \( y \) yields just \( y \).
- If \( x \) is of the form \( x_1::x_2 \), then concatenating \( x \) with \( y \) is a list of the form \( x_1::z \), where \( z \) is the results of concatenating \( x_2 \) with \( y \). In fact we can even be more specific by observing that \( x = hd(x)::tl(x) \).

This suggests the following recursive definition.

\[
\text{fun concat}(x,y) = \begin{cases} 
  y & \text{if } x=[] \\
  \text{hd}(x)::\text{concat}(\text{tl}(x),y) & \text{otherwise}
\end{cases}
\]

\[
\text{val concat} = \text{fn} : \text{'a list * 'a list -> 'a list}
\]

Applying the function yields the expected results:

\[
\begin{align*}
\text{concat}([1,2],[3,4,5]) & : \text{int list} \\
\text{val it} & = [1,2,3,4,5] \\
\text{concat}([], [1,2]) & : \text{int list} \\
\text{val it} & = [1,2] \\
\text{concat}([1,2], []) & : \text{int list} \\
\text{val it} & = [1,2]
\end{align*}
\]
More List Functions

- The following function computes the *length* of its argument list:

```haskell
- fun length(L) =
  = if (L=nil) then 0
  = else 1+length(tl(L));
```

```haskell
val length = fn : ''a list -> int
```

- length[1,2,3];
  val it = 3 : int
- length[[5],[4],[3],[2,1]];
  val it = 4 : int
- length[];
  val it = 0 : int

- The next function has a similar recursive structure. It *doubles* all the elements in its argument list (of integers).

```haskell
- fun doubleall(L) =
  = if L=[] then []
  = else (2*hd(L))::doubleall(tl(L));
```

```haskell
val doubleall = fn : int list -> int list
```

- doubleall[1,3,5,7];
  val it = [2,6,10,14] : int list
Reversing a List

• Concatenation of lists, for which we gave a recursive definition, is actually a built-in operator in SML, denoted by the symbol @.

• We use this operator in the following recursive definition of a function that reverses a list.

  - fun reverse(L) =
    =       if L = nil then nil
    =       else reverse(tl(L)) @ [hd(L)];

  val reverse = fn : ''a list -> ''a list

  - reverse [1,2,3];
  val it = [3,2,1] : int list
Definition by Patterns

• In SML functions can also be defined via patterns.

• The general form of such definitions is:

   ```sml
   fun <identifier>(<pattern1>) = <expression1>
   | <identifier>(<pattern2>) = <expression2>
   | ...
   | <identifier>(<patternK>) = <expressionK>;
   ```

   where the identifiers, which name the function, are all the same, all patterns are of the same type, and all expressions are of the same type.

• For example, here is an alternative definition of the reverse function:

   ```sml
   - fun reverse(nil) = nil
   = | reverse(x::xs) = reverse(xs) @ [x];
   ```

   ```sml
   val reverse = fn : 'a list -> 'a list
   ```

• In applying such a function to specific arguments, the patterns are inspected in order and the first match determines the value of the function.
The following function removes all occurrences of its first argument from its second argument list.

```ml
- fun remove(x,L) = 
  = if (L=[]) then [] 
  = else (if (x=hd(L)) 
  = then remove(x,tl(L)) 
  = else hd(L)::remove(x,tl(L))); 

val remove = fn : ''a * ''a list -> ''a list

- remove(1,[5,3,1]);
  val it = [5,3] : int list
- remove(2,[4,2,4,2,4,2,2]);
  val it = [4,4,4] : int list
```

We will use it as an auxiliary function in the definition of another function that removes all duplicate occurrences of elements from its argument list.

```ml
- fun removedupl(L) = 
  = if (L=[]) then [] 
  = else hd(L)::remove(hd(L),removedupl(tl(L))); 

val removedupl = fn : ''a list -> ''a list
```
Higher-Order Functions

• In functional programming languages functions can be used in definitions of other, so-called higher-order, functions.

• The following function, apply, applies its first argument (a function) to all elements in its second argument (a list of suitable type).

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

We may apply apply with any function as argument.

```ml
- fun square(x) = (x:int)*x; 
val square = fn : int -> int 
- apply(square,[[2,3,4]]); 
val it = [4,9,16] : int list
```

• The function apply is predefined in SML and is called map.
Sorting

- We next design a function for sorting a list of integers, using the following approach.

- The function is recursive and based on a method known as Merge-Sort. To sort a list $L$,
  - first split $L$ into two disjoint sublists (of about equal size),
  - then (recursively) sort the sublists, and
  - finally merge the (now sorted) sublists.

This recursive method is known as Merge-Sort.

- It requires suitable functions for
  - splitting a list into two sublists and
  - merging two sorted lists into one sorted list.
Splitting

- We split a list by applying two functions, `take` and `skip`, which extract alternate elements; respectively, the elements at odd-numbered positions and the elements at even-numbered positions (if any).

- The definitions of the two functions mutually depend on each other, and hence provide an example of *mutual recursion*, as indicated by the SML-keyword and:

  ```sml
  - fun take(L) =
    = if L = nil then nil
    = else hd(L)::skip(tl(L))
    = and
    =     skip(L) =
    =     if L=nil then nil
    =     else take(tl(L));
  val take = fn : ''a list -> ''a list
  val skip = fn : ''a list -> ''a list
  - take[1,2,3];
  val it = [1,3] : int list
  - skip[1,2,3];
  val it = [2] : int list```
A function for merging two sorted lists can easily be defined by recursion.

We give a definition by patterns:

- fun merge([],M) = M
  = | merge(L,[]) = L
  = | merge(x::xl,y::yl) =
  =   if (x:int)<y then x::merge(xl,y::yl)
  =     else y::merge(x::xl,yl);
val merge = fn : int list * int list -> int list
- merge([1,5,7,9],[2,3,5,5,10]);
  val it = [1,2,3,5,5,5,7,9,10] : int list
- merge([],[1,2]);
  val it = [1,2] : int list
- merge([1,2],[]);
  val it = [1,2] : int list
Merge Sort

- Using the above auxiliary functions we obtain the following function for sorting.

  ```ml
  fun sort(L) =
  =   if L=[] then []
  =   else merge(sort(take(L)),sort(skip(L)));
  val sort = fn : int list -> int list
  ``

  Don’t run this function, though, as it doesn’t quite work. Why?

- To see where the problem is, observe what the result is of applying `take` to a one-element list.

  ```ml
  - take[1];
  val it = [1] : int list
  ```

  Thus in this case, the first recursive call to `sort` will be applied to the same argument!

- Here is a modified version in which one-element lists are handled correctly.

  ```ml
  - fun sort(L) =
  =   if L=[] then []
  =   else if tl(L)=[] then L
  =   else merge(sort(take(L)),sort(skip(L)));
  val sort = fn : int list -> int list
  ```
The standard version of mergesort simply splits a given list into a first and a second half. (If the given list is of odd length, one sublist will have one more element.)

The following graph traces the execution of the standard mergesort algorithm on a specific input list of eight elements: