

Defining Functions in SML

- **Function evaluation** is the basic concept for a programming paradigm that has been implemented in such **functional programming languages** as ML.
- 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 ML, as in other functional languages, is the use of the **definition** and **application** of functions.
- The basic cycle of ML 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**.

- SML 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 are:
 - `real`
 - **Examples:** ~ 1.2 and $1.5e12$ (1.5×10^{12}) are reals.
 - `int`
 - **Examples:** ~ 12 and 14 are integers. $3 + 5$ is an integer.
 - `bool`
 - **Examples:** *true* and *not(true)* are booleans.
 - `string`
 - **Examples:** "nine" and "" are strings.

Binding Names to Values

- In SML one can associate identifiers with values,

```
- val three = 3;  
val three = 3 : int
```

and thereby establish a new *value binding*,

```
- three;  
val it = 3 : int
```

- More complex *expressions* can also be used to bind values to names,

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

- Names can then be used in other expressions,

```
- three + five;  
val it = 8 : int
```

Defining Functions in SML

- The general form of a function definition in SML is:

```
fun <identifier> (<parameters>) = <expression>;
```

- The corresponding **function type** is

type of parameters \rightarrow *type of expression*

- **Example:**

```
- fun double(x) = 2*x;  
val double = fn : int -> int
```

declares *double* as a function from integers to integers, i.e., of type $\text{int} \rightarrow \text{int}$.

```
- double(222);  
val it = 444 : int
```

- If we apply *double* to an argument of the wrong type, we get an error message:

```
- double(2.0);  
Error: operator and operand don't agree [tycon  
mismatch]  
operator domain:  int  
operand:  real  
in expression:  
double 2.0
```

- The user may also **explicitly** specify types.
- **Example:**

```
- fun max(x:int,y:int,z:int) =  
=   if ((x>y) andalso (x>z)) then x  
=   else (if (y>z) then y else z);  
val max = fn :  int * int * int -> int  
- max(3,2,2);  
val it = 3 :  int
```

Recursive Definitions

- The use of recursive definitions is a main characteristic of functional programming languages.
- These languages strongly encourage the use of recursion as a structuring mechanism in preference to 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 type of the function *factorial* is:

$$\text{int} \rightarrow \text{int}$$

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
```

Greatest Common Divisor

- The calculation of the **greatest common divisor (gcd)** of two positive integers can also be done recursively based on the following observations:

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

- A possible definition in SML is as follows:

```
- 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(126,2357);
val it = 1 : int
- gcd(125,56345);
val it = 5 : int
```

Tuples in SML

- SML provides two ways of defining data types that represent sequences.
 - **Tuples** are finite sequences of arbitrary but fixed length, where different components *need not be of the same type*.
 - **Lists** are finite sequences of elements of the *same type*.
- Some examples of tuples and the corresponding types are:

```
- 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)  
- val t3 = (7,8.0,"nine");  
val t3 = (7,8.0,"nine") : int * real * string
```

The type of $t1$ is `int * int * int`. The type of $t2$ is `int * (real * int)`. The type of $t3$ is `int * real * string`.

- The components of a tuple can be accessed by applying the built-in function `#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  
- #3(t3);  
val it = "nine" : string
```

If a function `#i` is applied to a tuple with fewer than *i* components, an error results:

```
- #4(t3);  
... Error: operator and operand don't agree
```

Lists in SML

- Another **built-in data structure** to represent *sequences* in SML are **lists**.
- A **list** in SML is essentially 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, in SML notation `int list list`.

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

Empty Lists

- **Empty lists** are denoted by the following symbols:
 - `[]`;
 - `val it = [] : 'a list`
 - `nil`;
 - `val it = [] : 'a list`
- Note that the type is described in terms of a *type variable* 'a, as a list of objects of type '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 `hd` returns the first element of its argument list.

```
- hd[1,2,3];  
val it = 1 : int  
- hd[[1,2],[3]];  
val it = [1,2] : int list
```

Applying this function to the empty list will result in an **exception** (error).

- The function `tl` removes the first element of its argument lists, and returns the remaining list.

```
- tl[1,2,3];  
val it = [2,3] : int list  
- 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 list processing functions.
- For example, consider the problem of defining an SML function, call it *concat*, that takes as arguments two lists of the same type and returns the concatenated list.
- For instance, the following applications of the function *concat* should yield the indicated responses.

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

- What is the SML type of *concat*?

- In defining such list processing functions, it is helpful to keep in mind that a list is either
 - the empty list, [], or
 - of the form $x::y$.
- The *empty list* and $::$ are the constructors of the type `list`.

For example,

```
- [1,2,3]=1::[2,3];  
val it = true : bool
```

Concatenation of Lists

- 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 $x1::x2$, then concatenating x with y is a list of the form $x1::z$, where z is the results of concatenating $x2$ with y . In fact we can even be more specific by observing that $x = hd(x)::tl(x)$.

- This suggests the following recursive definition.

```
- fun concat(x,y) = if x=[] then y
=                   else hd(x)::concat(tl(x),y);
val concat = fn :  ''a list * ''a list -> ''a
list
```

- This seems to work (*at least on some examples*):

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

More List Processing Functions

- **Recursion** often yields simple and natural definitions of functions on lists.
- The following function computes the *length* of its argument *list* by distinguishing between:
 - the empty list (the basis case) and
 - non-empty lists (the general case).

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

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

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

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

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

This function is of type `int list → int list`. Why?

The Reverse of 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 produces the *reverse* of 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
```

Pattern Matching

- We have previously used pattern matching when applying inference rules or logical equivalences.
- Informally, a **pattern** is an expression containing **variables**, for which other expressions may be substituted. The problem of matching a pattern against a given expression consists of finding a suitable substitution that makes the pattern identical to the expression.
- For example, we may apply De Morgan's Law,

$$\sim(\alpha \vee \beta) \equiv (\sim\alpha \wedge \sim\beta),$$

to the formula

$$\sim\sim(\sim p \vee q),$$

to obtain an equivalent formula

$$\sim(\sim\sim p \wedge \sim q).$$

Here the “meta-variables” α and β are replaced by the formulas $\sim p$ and q , respectively, to make the left-hand side of De Morgan's law identical to the subformula

$$\sim(\sim p \vee q)$$

of the given formula.

Function Definition by Patterns

- In SML there is an alternative form of defining functions via **patterns**.
- The general form of such definitions is:

```
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, an alternative definition of the reverse function is:

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

```
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.

Removing Elements from Lists

- The following function removes **all** occurrences of its first argument from its second argument list.

```
- 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  
- remove(2,nil); val it = [] : int list
```

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

```
- fun removedupl(L) =  
=   if (L=[]) then []  
=   else hd(L)::remove(hd(L),removedupl(tl(L)));  
  
val removedupl = fn : 'a list -> 'a list
```

Constructing Sublists

- A **sublist** of a list L is any list obtained by deleting some (i.e., zero or more) elements from L .
- For example, $[], [1], [2],$ and $[1,2]$ are all the sublists of $[1,2]$.
- Let us **design** an SML function that constructs **all** sublists of a given list L . The definition will be **recursive**, based on a case distinction as to whether L is the empty list or not.
- If L is non-empty, it has a first element x . There are two kinds of sublists: those containing x , and those not containing x .
- For instance, in the above example we have sublists $[1]$ and $[1,2]$ on the one hand, and $[], [2]$ on the other hand.
- Note that there is a one-to-one correspondence between the two kinds of sublists, and that each sublist of the latter kind is also a sublist of $\text{tl}(L)$.

Constructing Sublists (cont.)

- These observations lead to the following definition.

```
- fun sublists(L) =  
=   if (L=[]) then [nil]  
=   else sublists(tl(L))  
=       @ insertL(hd(L),sublists(tl(L)));  
  
val sublists = fn : 'a list -> 'a list list  
  
- sublists[];  
val it = [[]] : 'a list list  
- sublists[1,2];  
val it = [[],[2],[1],[1,2]] : int list list  
- sublists[1,2,3];  
val it = [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]  
: int list list  
- sublists[4,3,2,1];  
val it = [[],[1],[2],[2,1],[3],[3,1],[3,2],  
[3,2,1],[4],[4,1],...
```

- Recall that @ denotes concatenation of lists. The function insertL inserts its first argument at the front of all elements in its second argument (which must be a list). Its definition is left as an exercise.

- If we change the expression in the else-branch to

```
= else insertL(hd(L),sublists(tl(L)))  
=           @ sublists(tl(L))
```

all sublists will still be generated, but in a different order.

Higher-Order Functions

- In functional programming languages, parameters may denote **functions** and be used in definitions of other, so-called **higher-order**, functions.
- One example of a higher-order function is the function `apply` defined below, which applies its first argument (a function) to all elements in its second argument (a list of suitable type).

```
- 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.

```
- 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 `doubleall` we defined may be considered a special case of supplying `apply` with first argument `double` (a function we defined in a previous lecture).

```
- apply(double, [1,3,5,7]);  
val it = [2,6,10,14] : int list
```

- The function `apply` is predefined in SML and is called `map`.

Mutual Recursion

- Sometimes the most convenient way of defining (two or more different) functions is in **mutual dependence of each other**.
- Consider the functions, `even` and `odd` that test if a number is even and odd. We can define them in the following way.

```
- fun even(0) = true
= |   even(n) = odd(n-1)
= and
=     odd(0) = false
= |   odd(n) = even(n-1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

SML uses the keyword `and` (not to be confused with the logical operator `andalso`) for such mutually recursive definitions.

Neither of the two definition is acceptable by itself.

```
- even(2);
val it = true : bool
- odd(3);
val it = true : bool
```

- Consider two functions, `take` and `skip`, both of which extract alternate elements from a given list, with the difference that `take` starts with the first element (and hence extracts all elements at odd-numbered positions), whereas `skip` skips the first element (and hence extracts all elements at even-numbered positions, if any).

```
- 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
```

Sorting

- We next design a function for **sorting a list of integers**.

- More precisely, we want to define an SML function,

```
sort : int list -> int list
```

such that `sort(L)` is a sorted version (in non-descending order) of `L`.

- Sorting is an important problem for which a large variety of different algorithms have been proposed.
- The method we will explore is based on the following idea. 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 evidently requires us to define suitable functions for
 - splitting a list into two sublists and
 - merging two sorted lists into one sorted list.

Merging

- First we consider the problem of merging two sorted lists.
- A corresponding recursive definition can be easily defined by distinguishing between the different cases, as to whether one of the argument lists is empty or not.
- The following SML definition is formulated in terms of **patterns** (against which specific arguments in applications of the function will be matched during evaluation).

```
- 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
```

- How do we split a list? Recursion seems to be of little help for this task, but fortunately we have already defined suitable functions that solve the problem.

Merge Sort

- Using `take` and `skip` to split a list, we obtain the following function for sorting.

```
- 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.

```
- 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.

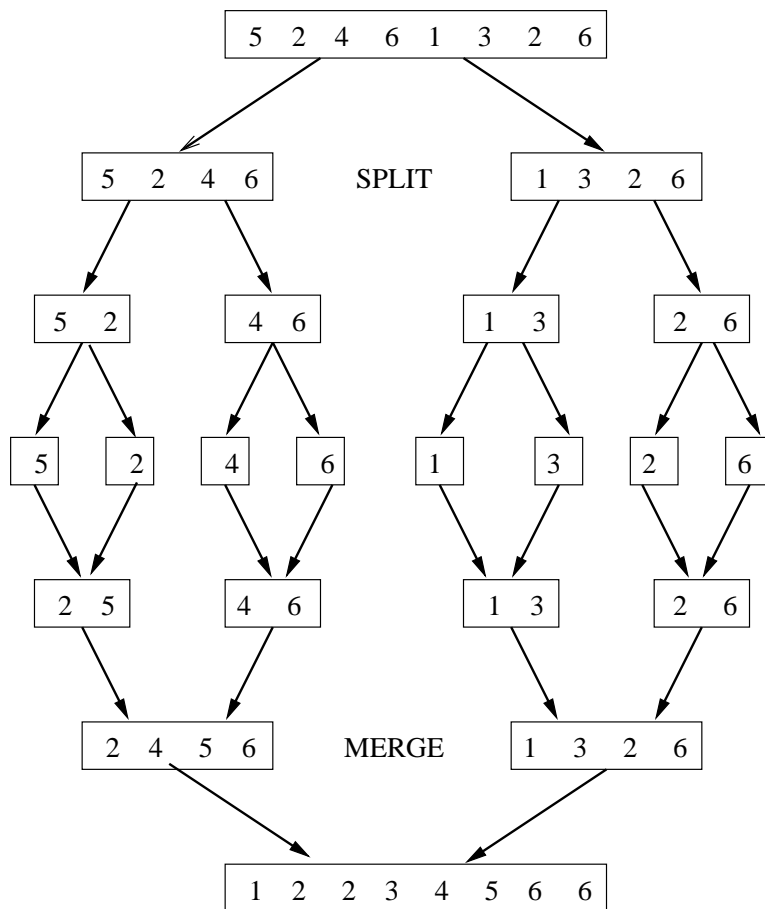
```
- fun sort(L) =  
=   if L=[] then []  
=   else if tl(L)=[] then L  
=   else merge(sort(take(L)),sort(skip(L)));  
val sort = fn : int list -> int list
```

Finally, some examples:

```
- sort[];
val it = [] : int list
- sort[1];
val it = [1] : int list
- sort[1,2];
val it = [1,2] : int list
- sort[2,1];
val it = [1,2] : int list
- sort[1,2,3,4,5,6,7,8,9];
val it = [1,2,3,4,5,6,7,8,9] : int list
- sort[9,8,7,6,5,4,3,2,1];
val it = [1,2,3,4,5,6,7,8,9] : int list
- sort[1,2,1,2,2,1,2,1,2,1];
val it = [1,1,1,1,1,2,2,2,2,2] : int list
```

Tracing Mergesort

- It is important to be able to trace the execution of mergesort to convince yourself that program works correctly.



- In the course of executing recursive function calls the computer needs to keep track of what work still needs to be done when the evaluation requires nested recursive calls.