

# SML

---

CSE 216 – PROGRAMMING ABSTRACTIONS

[HTTPS://PPAWAR.GITHUB.IO/CSE216-S19/INDEX.HTML](https://ppawar.github.io/cse216-s19/index.html)

SLIDES COURTESY:

DR. PAUL FODOR

STONY BROOK UNIVERSITY

# Functional Programming

---

*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 in the Logic for Computable Functions (LCF) theorem prover.

Standard ML of New Jersey (SML) is an implementation of ML.

The basic mode of computation in SML is the use of the definition and application of functions.

# Interacting with SML

---

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

- it provides the standard arithmetic operators

```
- 3+2;
```

```
val it = 5 : int
```

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

---

As part of the evaluation process, SML determines the type of the output value using 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:

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

For example,

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

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

# Function Definitions in SML

---

The user may also **explicitly** indicate types:

```
- 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, and these languages encourage the use of recursion over iterative constructs such as while loops:

```
- 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
```

# Example: Greatest Common Divisor

---

The greatest common divisor (gcd) of two positive integers can be defined recursively based on the following observations:

$\text{gcd}(n, n) = n,$

$\text{gcd}(m, n) = \text{gcd}(n, m),$  if  $m < n,$  and

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

These identities suggest the following recursive definition:

```
- 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);      - gcd(1,20);      - gcd(125,56345);
```

```
val it = 6 : int    val it = 1 : int    val it = 5 : int
```



# More recursive functions

---

```
- fun exp(b,n) = if n=0 then 1.0  
    else b * exp(b,n-1);
```

```
val exp = fn : real * int -> real
```

```
- exp(2.0,10);
```

```
val it = 1024.0 : real
```

# Tuples in SML

---

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

```
- (1, "two");  
val it = (1,"two") : int * string  
- 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  
- #2(t2);  
val it = (5.0,6) : real * int
```

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

# Polymorphic functions

---

```
- fun id x = x;  
val id = fn : 'a -> 'a  
- (id 1, id "two");  
val it = (1,"two") : int * string  
- fun fst(x,y) = x;  
val fst = fn : 'a * 'b -> 'a  
- fun snd(x,y) = y;  
val snd = fn : 'a * 'b -> 'b  
- fun switch(x,y) = (y,x);  
val switch = fn : 'a * 'b -> 'b * 'a
```

# Polymorphic functions

---

'**a**' means "any type", while '**'a**' means "any type that can be compared for equality" (see the **concat** function later which compares a polymorphic variable list with **[]**).

There will be a "*Warning: calling polyEqual*" that means that you're comparing two values with polymorphic type for equality.

- Why does this produce a warning? Because it's less efficient than comparing two values of known types for equality.
- How do you get rid of the warning? By changing your function to only work with a specific type instead of any type.
- Should you do that or care about the warning? Probably not. In most cases having a function that can work for any type is more important than having the most efficient code possible, so you should just ignore the warning.

# Lists in SML

---

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

- `[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.

# Lists in SML

---

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 `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 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.

# Operations on Lists

---

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
```

The arguments must be of the right type (such that the result is a list of elements of the same type):

```
- [1]::[2,3];
```

```
Error: operator and operand don't agree
```



# Operations on Lists

---

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`

# Concatenation

---

In designing a function for concatenating two lists  $\mathbf{x}$  and  $\mathbf{y}$  we thus distinguish two cases, depending on the form of  $\mathbf{x}$ :

- If  $\mathbf{x}$  is an empty list  $[]$ , then concatenating  $\mathbf{x}$  with  $\mathbf{y}$  yields just  $\mathbf{y}$ .
- If  $\mathbf{x}$  is of the form  $\mathbf{x1} :: \mathbf{x2}$ , then concatenating  $\mathbf{x}$  with  $\mathbf{y}$  is a list of the form  $\mathbf{x1} :: \mathbf{z}$ , where  $\mathbf{z}$  is the result of concatenating  $\mathbf{x2}$  with  $\mathbf{y}$ .
- We can be more specific by observing that  $\mathbf{x} = \mathbf{hd}(\mathbf{x}) :: \mathbf{tl}(\mathbf{x})$

# Concatenation

---

```
- fun concat(x,y) = if x=[] then y  
  else hd(x)::concat(tl(x),y);
```

```
val concat = fn : 'a list * 'a list -> 'a list
```

Applying the function yields the expected results:

```
- 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
```

# Length

---

The following function computes the length of its argument list:

```
- 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
```

# doubleall

---

The following function 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
```

# 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 reverse 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
```

This method is not efficient:  $O(n^2)$

# Reversing a List

---

This way (using an accumulator) is better:  $O(n)$

```
- fun reverse_helper(L,L2) =  
  if L = nil then L2  
    else reverse_helper(tl(L),hd(L)::L2) ;  
  
- fun reverse(L) = reverse_helper(L,[]) ;
```



# Removing List Elements

---

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
```

# Removing Duplicates

---

The remove function can be used 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)::removedupl(remove(hd(L),tl(L)));  
val removedupl = fn : 'a list -> 'a list  
  
- removedupl([3,2,4,6,4,3,2,3,4,3,2,1]);  
val it = [3,2,4,6,1] : int list
```