CSE 307 – Principles of Programming Languages
Stony Brook University

http://www.cs.stonybrook.edu/~cse307
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 1970 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 (whose language, pplambda, a combination of the first-order predicate calculus and the simply typed polymorphic lambda calculus, had ML as its metalanguage)

- 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
Install Standard ML

- Download from:
  - [http://www.smlnj.org](http://www.smlnj.org)

- Start Standard ML:
  - Type `sml` from the shell (run command line in Windows)

- Exit Standard ML:
  - `Ctrl-Z` under Windows
  - `Ctrl-D` under Unix/Mac
Standard ML

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

- SML prompt:

  -

- 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
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:
  \[
  \text{fun} \quad \text{<identifier>} \quad (\text{<parameters>}) \quad = \quad \text{<expression>};
  \]

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

- Apply a function to an argument of the wrong type results in an error message:
  \[
  \text{double}(2.0);
  \]
  \text{Error: operator and operand don’t agree ...}
Function Definitions in SML

- The user may also explicitly indicate types:
  
  ```sml
  fun max(x:int,y:int,z:int):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:

- \texttt{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:

- \texttt{factorial(5);}

val it = 120 : int

- \texttt{factorial(10);}

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

\[ \text{gcd}(n, n) = n, \]
\[ \text{gcd}(m, n) = \text{gcd}(n, m), \text{ if } m < n, \text{ and } \]
\[ \text{gcd}(m, n) = \text{gcd}(m - n, n), \text{ if } m > n. \]

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);  - 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
  - \((1, "two")\);
  - \(val \ \text{it} = (1,"two") : int \ast string\)
  - \(val \ \text{t1} = (1,2,3)\);
  - \(val \ \text{t1} = (1,2,3) : int \ast int \ast int\)
  - \(val \ \text{t2} = (4,(5.0,6))\);
  - \(val \ \text{t2} = (4,(5.0,6)) : int \ast (real \ast int)\)

- The components of a tuple can be accessed by applying the built-in functions \(#i\), where \(i\) is a positive number
  - \(#1(\text{t1})\);
  - \(val \ \text{it} = 1 : int\)
  - \(#2(\text{t2})\);
  - \(val \ \text{it} = (5.0,6) : real \ast int\)

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
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

• IMPORTANT: 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
Lists can also be compared for equality:

- \([1,2,3] = [1,2,3]\);
  \[
  \text{val it = true : bool}
  \]

- \([1,2] = [2,1]\);
  \[
  \text{val it = false : bool}
  \]

- \(\text{tl\[1\]} = []\);
  \[
  \text{val it = true : bool}
  \]
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 \( x \) and \( y \) we thus distinguish two cases, depending on the form of \( x \):
  • If \( x \) is an empty list \([\text{}]\), 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 result of concatenating \( x_2 \) with \( y \).
  • We can be more specific by observing that

\[
x = x_1 :: x_2 = \text{hd}(x) :: \text{tl}(x)
\]
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:
  
  ```ml
  - 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

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

- reverse [1,2,3];
calls
  - concat(reverse([2,3]), [1])
  - concat([3,2], [1]);
val it = [3,2,1] : 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 can use this operator in reversing:

```sml
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
Reversing a List

- fun reverse(L) =
  
  if L = nil then nil

  else concat(reverse(tl(L)), [hd(L)])

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

$T(N) = T(N-1) + (N-1) =$

$= T(N-2) + (N-2) + (N-1) =$

$= 1 + 2 + 3 + \ldots + N-1 = N * (N-1)/2$
Reversing a List

- This way (using an **accumulator**) is better: $O(n)$
- \textbf{fun} reverse\_helper(L,L2) =
  \begin{align*}
  \text{if } L = \text{nil} & \text{ then } L2 \\
  \text{else } & \text{ reverse\_helper}(\text{tl}(L),\text{hd}(L)::L2);
  \end{align*}
- \textbf{fun} reverse(L) = reverse\_helper(L,[]);
- reverse [1,2,3];
- reverse\_helper([[1,2,3],[[]]]);
- reverse\_helper([[2,3],[1]]);
- reverse\_helper([[3],[2,1]]);
- reverse\_helper([[]],[3,2,1]);
- [3,2,1]
Removing List Elements

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

```ocaml
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]);`
  ```ocaml```
  val it = [5, 3] : int list
  ```ocaml```

- `remove (2, [4, 2, 4, 2, 4, 2, 2]);`
  ```ocaml```
  val it = [4, 4, 4] : int list
  ```ocaml```
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
Definition by Patterns

- In SML functions can also be defined via patterns.
  - The general form of such definitions is:

    ```ml
    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.

- Example:

  ```ml
  - fun reverse(nil) = nil
  | reverse(x::xs) = reverse(xs) @ [x];
  val reverse = fn : 'a list -> 'a list
  ```

  The patterns are inspected in order and the first match determines the value of the function.
fun member(X,L) = 
    if L=[] then false 
    else if X=hd(L) then true 
    else member(X,tl(L));

OR with patterns:

fun member(X,[]) = false |
    member(X,Y::Ys) = 
        if (X=Y) then true 
        else member(X,Ys);

member(1,[1,2]); (* true *)
member(1,[2,1]); (* true *)
member(1,[2,3]); (* false *)
fun union(L1,L2) =
    if L1=[] then L2
    else if member(hd(L1),L2)
        then union(tl(L1),L2)
        else hd(L1)::union(tl(L1),L2);

union([1,5,7,9],[2,3,5,10]); (* [1,7,9,2,3,5,10] *)
union([],[1,2]); (* [1,2] *)
union([1,2],[]); (* [1,2] *)
fun union([],L2) = L2
    | union(X::Xs,L2) = 
        if member(X,L2) then union(Xs,L2)
        else X::union(Xs,L2);
fun intersection(L1,L2) =
  if L1=[] then []
  else if member(hd(L1),L2) then hd(L1)::intersection(tl(L1),L2)
  else intersection(tl(L1),L2);

intersection([1,5,7,9],[2,3,5,10]);
(* [5] *)
fun intersection([], L2) = []
| intersection(L1, []) = []
| intersection(X::Xs, L2) =
    if member(X, L2)
    then X::intersection(Xs, L2)
    else intersection(Xs, L2);
fun subset(L1,L2) = if L1=[] then true
    else if L2=[] then false
    else if member(hd(L1),L2)
        then subset(tl(L1),L2)
        else false;

subset([1,5,7,9],[2,3,5,10]);(* false *)
subset([5],[2,3,5,10]); (* true *)
Sets subset patterns

fun subset([],L2) = true

| subset(L1,[]) = if(L1=[])
  then true
  else false
| subset(X::Xs,L2) = 
  if member(X,L2)
    then subset(Xs,L2)
    else false;

(c) Paul Fodor (CS Stony Brook)
fun setEqual(L1,L2) =
    subset(L1,L2) andalso subset(L2,L1);

setEqual([1,5,7],[7,5,1,2]); (* false *)
setEqual([1,5,7],[7,5,1]); (* true *)
fun minus([],L2) = []
    | minus(X::Xs,L2) = 
        if member(X,L2)
            then minus(Xs,L2)
        else X::minus(Xs,L2);

minus([1,5,7,9],[2,3,5,10]);
(* [1,7,9] *)
Sets Cartesian product

fun product_one(X, []) = []
| product_one(X, Y::Ys) =
  (X, Y)::product_one(X, Ys);

product_one(1, [2, 3]);
(* [ (1,2), (1,3) ] *)

fun product([], L2) = []
| product(X::Xs, L2) =
  union(product_one(X, L2),
        product(Xs, L2));

product([[1, 5, 7, 9], [2, 3, 5, 10]]);
(* [ (1,2), (1,3), (1,5), (1,10), (5,2),
    (5,3), (5,5), (5,10), (7,2), (7,3), ... ] *)
fun insert_all(E,L) = 
  if L=[] then []
  else (E::hd(L)) :: insert_all(E,tl(L));
insert_all(1,[[[],[2],[3],[2,3]]]);
  (* [ [1], [1,2], [1,3], [1,2,3] ] *)

fun powerSet(L) = 
  if L=[] then [[]]
  else powerSet(tl(L)) @
      insert_all(hd(L),powerSet(tl(L)));

powerSet([]);
powerSet([1,2,3]);
powerSet([2,3]);
Higher-Order Functions

- In functional programming languages functions (called \textit{first-class functions}) can be used in definitions of other, called \textit{higher-order}, functions:

  - The following function, \texttt{map}, applies its \texttt{first argument (a function)} to all elements in its second argument (a list of suitable type):

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

  - We may apply \texttt{map} with any function as argument:

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

- **Anonymous functions**:
  - `map(fn x=>x+1, [1,2,3,4,5]);`
  - `val it = [2,3,4,5,6] : int list`

  - `fun incr(list) = map (fn x=>x+1, list);`
  - `val incr = fn : int list -> int list`

  - `incr[1,2,3,4,5];`
  - `val it = [2,3,4,5,6] : int list`
McCarthy's 91 function:

- `fun mc91(n) = if n>100 then n-10 else mc91(mc91(n+11));`
- `val mc91 = fn : int -> int`

- `map mc91 [101, 100, 99, 98, 97, 96];`
- `val it = [91,91,91,91,91,91,91] : int list`
Filter

- Filter: keep in a list only the values that satisfy some logical condition/boolean function:

  - fun filter(f,l) =
    
    if l=[] then []
    
    else if f(hd l)
    
      then (hd l)::(filter (f, tl l))
    
    else filter(f, tl l);
    
    val filter = fn : ('a -> bool) * 'a list -> 'a list

  - filter((fn x => x>0), [~1,0,1]);
  
  val it = [1] : int list
Permutations

- fun myInterleave(x,[]) = [[x]]
  | myInterleave(x,h::t) =
  | (x::h::t)::(
  |   map((fn l => h::l), myInterleave(x,t)));

- myInterleave(1,[]);
  val it = [[1]] : int list list

- myInterleave(1,[2]);
  val it = [[1,2],[2,1]] : int list list

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

- fun appendAll(nil) = nil
  | appendAll(z::zs) = z @ (appendAll(zs));

- appendAll([[1,2],[2,1]]);
val it = [[1,2],[2,1]] : int list list

- fun permute(nil) = [[]]
  | permute(h::t) = appendAll(
    map((fn l => myInterleave(h,l)), permute(t)));

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

- fun f(a)(b)(c) = a+b+c;
val f = fn : int -> int -> int -> int
val f = fn : int -> (int -> (int -> int))

OR

- fun f a b c = a+b+c;
- val inc1 = f(1);
val inc1 = fn : int -> int -> int
val inc1 = fn : int -> (int -> int)
- val inc12 = inc1(2);
val inc12 = fn : int -> int
- inc12(3);
val it = 6 : int
Composition

- Composition is another example of a higher-order function:
  - `fun comp(f,g)(x) = f(g(x));`
  - `val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b`
  - `val f = comp(Math.sin, Math.cos);`
  - `val f = fn : real -> real`
  
  **SAME WITH:**
  - `val g = Math.sin o Math.cos;`
  
  (* Composition "o" is predefined *)
  - `val g = fn : real -> real`
  - `f(0.25);`
  - `val it = 0.824270418114 : real`
  - `g(0.25);`
  - `val it = 0.824270418114 : real`
Mutually recursive function definitions

- `fun odd(n) = if n=0 then false
  else even(n-1)`

  and

  `even(n) = if n=0 then true
  else odd(n-1);`

- `val odd = fn : int -> bool`
- `val even = fn : int -> bool`

- `even(1);`
- `val it = false : bool`
- `odd(1);`
- `val it = true : bool`
• **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

  • 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.

- 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));
```

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

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

(c) Paul Fodor (CS Stony Brook)
Merging

- Merge pattern definition:
  - 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

- 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
Records

- Records are structured data types of heterogeneous elements that are labeled

  - \( \{x=2, \ y=3\} \);
  - The order does not matter:

  - \{make="Toyota", \ model="Corolla", \ year=2017, \ color="silver"\}  
    = \{model="Corolla", \ make="Toyota", \ color="silver", \ year=2017\};

- \texttt{val it = true : bool}

- \texttt{fun full_name\{first:string, last:string, \ age:int, balance:real\}:string =}
  \texttt{first ^ " " ^ last;}

(* ^ is the string concatenation operator *)

- \texttt{val full_name=fn:\{age:int, balance:real, first:string, \ last:string\} -> string}
string and char

- "a";
val it = "a" : string
- #"a";
val it = #"a" : char
- explode("ab");
val it = ["a","b"] : char list
- implode(["a","b"]);
val it = "ab" : string
- "abc" ^ "def" = "abcdef";
val it = true : bool
- size ("abcd");
val it = 4 : int
string and char

- `String.sub("abcde",2);`
  `val it = #"c" : char`
- `substring("abcdefghij",3,4);`
  `val it = "defg" : string`
- `concat ["AB"," ","CD"];`
  `val it = "AB CD" : string`
- `str(#"x");`
  `val it = "x" : string`
The program of Young McML

fun tartan_column(i,j,n) =
  if j=n+1 then "\n"
  else if (i+j) mod 2=1 then
      concat(['"* "',tartan_column(i,j+1,n)])
  else concat(['"+ "',tartan_column(i,j+1,n)]);

fun tartan_row(i,n) =
  if i=n+1 then ""
  else concat(['tartan_column(i,1,n),
               tartan_row(i+1,n)]);

fun tartan(n) = tartan_row(1,n);
print(tartan(30));