SML

Paul Fodor
CSE260, Computer Science B: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse260
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 1977 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
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
Interacting with SML

- SML has a number of built-in operators and data types.
  - it provides the standard arithmetic operators
    - \( 3+2; \)
    - \texttt{val it = 5 : int} 
  - The boolean values \texttt{true} and \texttt{false} are available, as are logical operators such as: \texttt{not} (negation), \texttt{andalso} (conjunction), and \texttt{orelse} (disjunction)
    - \texttt{not(true);} 
    - \texttt{val it = false : bool} 
    - \texttt{true andalso false;}
    - \texttt{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 → 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):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:

  \[ \text{gcd}(n, n) = n, \]

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

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

- 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
  ```
### Basic operators on the integers

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int × int → int</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>−</td>
<td>int × int → int</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>div</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>mod</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>=</td>
<td>int × int → bool *</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>int × int → bool *</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>~</td>
<td>int → int</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>int → int</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

- The infix operators associate to the left
- The operands are always all evaluated
## Basic operators on the reals

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>−</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>/</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>~</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.sqrt</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.In</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

*Unary operator − is represented by ~*
### Type conversions

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>int $\rightarrow$ real</td>
</tr>
<tr>
<td>ceil</td>
<td>real $\rightarrow$ int</td>
</tr>
<tr>
<td>floor</td>
<td>real $\rightarrow$ int</td>
</tr>
<tr>
<td>round</td>
<td>real $\rightarrow$ int</td>
</tr>
<tr>
<td>trunc</td>
<td>real $\rightarrow$ int</td>
</tr>
</tbody>
</table>

- `real(2) + 3.5 ;`
  \[ \text{val it} = 5.5 : \text{real} \]
- `ceil(23.65) ;`
  \[ \text{val it} = 24 : \text{int} \]
- `ceil(~23.65) ;`
  \[ \text{val it} = \sim 23 : \text{int} \]
- `floor(23.65) ;`
  \[ \text{val it} = 23 : \text{int} \]
- 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.
Tuples in SML

- Functions using tuples should completely define the type of tuples, otherwise SML cannot detect the type, e.g., nth argument

  - fun firstThird(Tuple:'a * 'b * 'c):'a * 'c = (#1(Tuple), #3(Tuple));
  - val firstThird = fn : 'a * 'b * 'c => 'a * 'c
  - firstThird((1,"two",3));
  - val it = (1,3) : int * int

- Without types, we would get an error:
  - fun firstThird(Tuple) = (#1(Tuple), #3(Tuple));
  - stdIn: Error: unresolved flex record (need to know the names of ALL the fields in this context)
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

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

- There will be a "\textit{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 = \([1,2,3] : \text{int list}\)
  - \([\text{true}, \text{false}, \text{true}];;\)
  val it = \([\text{true}, \text{false}, \text{true}] : \text{bool list}\)
  - \([[[1,2,3],[4,5],[6]];;\)
  val it = \([[1,2,3],[4,5],[6]] : \text{int list list}\)

• 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

- 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
Operations on Lists

- 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`
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 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

(c) Paul Fodor (CS Stony Brook)
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):

```plaintext
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: \( O(n^2) \)

\[
T(N) = T(N-1) + (N-1) = \\
= T(N-2) + (N-2) + (N-1) = \\
= 1 + 2 + 3 + \ldots + N-1 = N \cdot (N-1)/2
\]
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,[]);
- 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

```haskell
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,[4,2,4,2,4,2,2,2]);
  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
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] *)
Sets UNION with patterns

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;
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] *)
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 as parameters or return value in definitions of other (called \textit{higher-order}) functions.

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

  ```plaintext
  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:

  ```plaintext
  - 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

- McCarthy's 91 function:
  
  ```ml
  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), [~1,0,1,2,3,~2,4]);
val it = [1,2,3,4] : 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,[3]);
  val it = [[1,3],[3,1]] : 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));
  flattens the list

- 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 = partial application

- fun f a b c = a+b+c;

OR

- fun f(a)(b)(c) = a+b+c;

val f = fn : int -> int -> int -> int

val f = fn : int -> (int -> (int -> int))

- 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
Currying and Lazy evaluation

- fun mult x y = if x = 0 then 0 else x * y;

Eager evaluation: reduce as much as possible before applying the function

\[
\text{mult} \ (1-1) \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ x => (\text{fn} \ y => \text{if} \ x = 0 \ \text{then} \ 0 \ \text{else} \ x \times y)) \ (1-1) \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ x => (\text{fn} \ y => \text{if} \ x = 0 \ \text{then} \ 0 \ \text{else} \ x \times y)) \ 0 \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ y => \text{if} \ 0 = 0 \ \text{then} \ 0 \ \text{else} \ 0 \times y) \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ y => \text{if} \ 0 = 0 \ \text{then} \ 0 \ \text{else} \ 0 \times y) \ \text{error}
\]

Lazy evaluation:

\[
\text{mult} \ (1-1) \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ x => (\text{fn} \ y => \text{if} \ x = 0 \ \text{then} \ 0 \ \text{else} \ x \times y)) \ (1-1) \ (3 \div 0)
\]

\[
\rightarrow (\text{fn} \ y => \text{if} \ (1-1) = 0 \ \text{then} \ 0 \ \text{else} \ (1-1) \times y) \ (3 \div 0)
\]

\[
\rightarrow \text{if} \ (1-1) = 0 \ \text{then} \ 0 \ \text{else} \ (1-1) \times (3 \div 0)
\]

\[
\rightarrow \text{if} \ 0 = 0 \ \text{then} \ 0 \ \text{else} \ (1-1) \times (3 \div 0)
\]

\[
\rightarrow 0
\]
Currying and Lazy evaluation

- Argument evaluation as late as possible (possibly never)
- Evaluation only when indispensable for a reduction
- Each argument is evaluated at most once
- Lazy evaluation in Standard ML for the primitives: `if then else , andalso , orelse`, and pattern matching
- Property: If the eager evaluation of expression `e` gives `n1` and the lazy evaluation of `e` gives `n2` then `n1 = n2`
- Lazy evaluation gives a result more often
Sum sequence

- fun sum f n = 
  if n = 0 then 0 
  else f(n) + sum f (n-1);
val sum = fn : (int → int) → int → int

- sum (fn x => x * x) 3 ;
val it = 14 : int
because
f(3) + f(2) + f(1) + f(0) = 9 + 4 + 1 + 0 = 14
Composition

- Composition is another example of a higher-order function:

```plaintext
- 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
```
Pick only the first element of a list that satisfies a given predicate:

- fun myFind pred nil = raise Fail "No such element"
  | myFind pred (x::xs) =
    if pred x then x
    else myFind pred xs;
val myFind = fn : ('a -> bool) -> 'a list -> 'a

- myFind (fn x => x > 0.0) [~1.2, ~3.4, 5.6, 7.8];
val it = 5.6 : real
Reduce (aka. foldr)

- We can generalize the notion of recursion over lists as follows: all recursions have a base case, an iterative case, and a way of combining results:

  - fun reduce f b nil = b
    | reduce f b (x::xs) = f(x, reduce f b xs);

  - fun sumList aList = reduce (op +) 0 aList;

  val sumList = fn : int list -> int

  - sumList [1, 2, 3];

  val it = 6 : int
foldl

- fun foldl(f: ''a*'b->'b, acc: 'b, l: ''a list): 'b =
  if l=[] then acc
  else foldl(f, f(hd(l),acc), tl(l));

- fun sum(l:int list):int =
  foldl((fn (x,acc) => acc+x),0,l);

- sum[1, 2, 3];
  val it = 6 : int

  it walks the list from left to right
foldl vs. reduce (foldr)

foldr (op ^) "" ["a", "b", "c"]
type: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

foldl (op ^) "" ["a", "b", "c"]
type: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

(c) Paul Fodor (CS Stony Brook)
Numerical integration

- Computation of $\int_a^b f(x) \, dx$ by the trapezoidal rule:

   $n$ intervals
   
   $h = \frac{b - a}{n}$

   $\approx h \left( f(a) + f(a+h) \right) / 2$
Numerical integration

- fun integrate (f,a,b,n) =
  if n <= 0 orelse b <= a then 0.0
  else ((b−a) / real n) * ( f(a) + f(a+h) ) / 2.0 +
    integrate (f,a+((b−a) / real n),b,n−1);
val integrate = fn : (real → real) * real * real * int
  → real

- fun cube x:real = x * x * x ;
val cube = fn : real → real
- integrate ( cube , 0.0 , 2.0 , 10 ) ;
val it = 4.04 : real
Collect like in Java streams

- fun collect(b, combine, accept, nil) = accept(b)
  | collect(b, combine, accept, x::xs) =
  | collect(combine(b,x), combine, accept, xs);

- fun average(aList) = collect((0,0),
  (fn ((total,count),x) => (total+x,count+1)),
  (fn (total,count) => real(total)/real(count)),
  aList);

- average [1, 2, 4];
val it = 2.3333333333333333 : 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
Sorting

- **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));
  val take = fn : ''a list -> ''a list 
  val skip = fn : ''a list -> ''a list 
  - 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 
  ```
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],[2,3,6,8,10]);
  val it = [1,2,3,5,6,7,8,9,10] : 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
Local declarations

```
- fun fraction (n,d) =
  let val k = gcd (n,d)
  in
    ( n div k , d div k )
  end;
```

- The identifier `k` is local to the expression after `in`
- Its binding exists only during the evaluation of this expression
- All other declarations of `k` are hidden during the evaluation of this expression
Sorting with comparison

- How to sort a list of elements of type α?
  - We need the comparison function/operator for elements of type α!

```plaintext
fun sort order [] = []
| sort order [x] = [x]
| sort order xs =
    let fun merge [] M = M
    | merge L [] = L
    | merge (L as x::xs) (M as y::ys) =
        if order(x, y) then x::merge xs M
        else y::merge L ys
    val (ys, zs) = split xs
    in merge (sort order ys) (sort order zs) end;

- sort (op >) [5.1, 3.4, 7.4, 0.3, 4.0];
val it = [7.4, 5.1, 4.0, 3.4, 0.3] : real list
```
Sorting with comparison

- fun split_helper(L: 'a list, Acc:'a list * 'a list) :
  'a list * 'a list =
  if L=[] then Acc
  else split_helper(tl(L), (#2(Acc), (hd(L)) :: #1(Acc)));

- fun split(L) = split_helper(L, ([], []));

- split([1,2,3,4,5,6]);
  split([1,2,3,4,5,6])
  split_helper([1,2,3,4,5,6], ([],[]))
  split_helper([2,3,4,5,6], ([],[1]))
  split_helper([3,4,5,6], ([1],[2]))
  split_helper([4,5,6], ([2],[3,1]))
  split_helper([5,6], ([3,1],[4,2]))
  split_helper([6], ([4,2],[5,3,1]))
  split_helper([], ([5,3,1],[6,4,2]))
  ([5,3,1],[6,4,2])
Quicksort

- C.A.R. Hoare, in 1962: Average-case running time: $\Theta(n \log n)$

fun sort [ ] = [ ]

| sort (x::xs) =

  let val (S,B) = partition (x, xs)
  in (sort S) @ (x :: (sort B))
  end;

Double recursion and no tail-recursion

- fun partition (p,[ ]) = ([ ],[ ])

| partition (p,x::xs) =

  let val (S,B) = partition (p, xs)
  in if x < p then (x::S,B) else (S,x::B)
  end
Nested recursion

For \( m, n \geq 0 \):

\[
\text{acker}(0,m) = m+1 \\
\text{acker}(n,0) = \text{acker}(n-1, 1) \text{ for } n > 0 \\
\text{acker}(n,m) = \text{acker}(n-1, \text{acker}(n,m-1)) \text{ for } n,m>0
\]

- fun acker 0 m = m+1
  | acker n 0 = acker (n-1) 1
  | acker n m = acker (n-1) (acker n (m-1));

It is guaranteed to end because of *lexicographic order*:

(\( n',m' \)) < (\( n,m \)) iff \( n' < n \) or (\( n'=n \) and \( m'< m \))
Nested recursion

- **Knuth's up-arrow operator** $\uparrow^n$ (invented by Donald Knuth):
  
  \[
  a \uparrow^1 b = a^b \\
  a \uparrow^n b = a \uparrow^{n-1} (b \uparrow^{n-1} b) \text{ for } n > 1
  \]

  - \text{fun opKnuth 1 a b = Math.pow (a,b)}
    \[| \text{opKnuth n a b = opKnuth (n-1) a}
    \text{ (opKnuth (n-1) b b);} \]
  
  - \text{opKnuth 2 3.0 3.0 ;}
  
  val it = 7.62559748499E12 : real
  
  - \text{opKnuth 3 3.0 3.0 ;}
  
  ! Uncaught exception: Overflow;

- **Graham's number** (also called the “largest” number):
  
  - \text{opKnuth 63 3.0 3.0},
Recursion on a generalized problem

- It is impossible to determine whether \( n \) is prime via the reply to the question “is \( n - 1 \) prime”?
- It seems impossible to directly construct a recursive program
- We thus need to find another function that is more general than prime, in the sense that prime is a particular case of this function
  - for which a recursive program can be constructed

- \[
\text{fun } \text{ ndivisors } n \text{ low up } = \text{ low } > \text{ up } \text{ orelse } \\
(n \mod \text{ low}) \neq 0 \text{ andalso } \text{ ndivisors } n \text{ (low+1) up; }
\]
- \[
\text{fun } \text{ prime } n = \text{ if } n \leq 0 \\
\text{ then error } \text{ "prime: non-positive argument" }
\text{ else if } n = 1 \text{ then false }
\text{ else ndivisors } n \text{ 2 floor(Math.sqrt(real n)) ; }
\]
- The discovery of divisors requires imagination and creativity
Tail recursion

- fun length [ ] = 0
  | length (x::xs) = 1 + length xs;

- The recursive call of length is nested in an expression: during the evaluation, all the terms of the sum are stored, hence the memory consumption for expressions & bindings is proportional to the length of the list!

  length [5,8,4,3]
  -> 1 + length [8,4,3]
  -> 1 + (1 + length [4,3])
  -> 1 + (1 + (1 + length [3]))
  -> 1 + (1 + (1 + (1 + length [ ])))
  -> 1 + (1 + (1 + (1 + 0)))
  -> 1 + (1 + (1 + 1))
  -> 1 + (1 + 2)
  -> 1 + 3
  -> 4
Tail recursion

- fun lengthAux [ ] acc = acc
  | lengthAux (x::xs) acc = lengthAux xs (acc+1);
- fun length L = lengthAux L 0;
- length [5,8,4,3];
  -> lengthAux [5,8,4,3] 0
  -> lengthAux [8,4,3] (0+1)
  -> lengthAux [8,4,3] 1
  -> lengthAux [4,3] (1+1)
  -> lengthAux [4,3] 2
  -> lengthAux [3] (2+1)
  -> lengthAux [3] 3
  -> lengthAux [ ] (3+1)
  -> lengthAux [ ] 4
  -> 4

- Tail recursion: recursion is the outermost operation
  - Space complexity: constant memory consumption for expressions & bindings (SML can use the same stack frame/activation record)
  - Time complexity: (still) one traversal of the list
Tail recursion

- fun factAux 0 acc = acc
  | factAux n acc = factAux (n-1) (n*acc);
- fun fact n =
  if n < 0 then error "fact: negative argument"
  else factAux n 1;

- fact(3);
  -> factAux(3,1)
  -> factAux(2,3)
  -> factAux(1,6)
  -> factAux(0,6)
  6
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\};

val it = true : bool

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

  (* ^ is the string concatenation operator *)

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
Functional programming in SML

• Covered fundamental elements:
  • Evaluation by reduction of expressions
  • Recursion
  • Polymorphism via type variables
  • Strong typing
  • Type inference
  • Pattern matching
  • Higher-order functions
  • Tail recursion
Beyond functional programming

- **Relational programming** (aka *logic programming*)
  - For which triples does the `append` relation hold?
    
    ?- append ([1,2], [3], X).
    Yes
    X = [1,2,3]

    ?- append ([1,2], X, [1,2,3]).
    X = [3]

    ?- append (X, Y, [1,2,3]).
    X = [], Y = [1,2,3];
    X = [1], Y = [2,3];
    ...
    X = [1,2,3], Y = [];

- No differentiation between arguments and results!
Beyond functional programming

- **Backtracking** mechanism to enumerate all the possibilities
- **Unification** mechanism, as a generalization of pattern matching
- Power of the logic paradigm / relational framework
Beyond functional programming

**Constraint Processing:**

- Constraint Satisfaction Problems (CSPs)
  - Variables: $X_1, X_2, \ldots, X_n$
  - Domains of the variables: $D_1, D_2, \ldots, D_n$
  - Constraints on the variables: examples: $3 \cdot X_1 + 4 \cdot X_2 \leq X_4$
  - What is a solution?
    - An assignment to each variable of a value from its domain, such that all the constraints are satisfied

**Objectives:**

- Find a solution
- Find all the solutions
- Find an optimal solution, according to some cost expression on the variables
Beyond functional programming

• The n-Queens Problem:
  • How to place n queens on an n × n chessboard such that no queen is threatened?
  • Variables: X1, X2, . . . , Xn (one variable for each column)
  • Domains of the variables: Di = {1, 2, . . . , n} (the rows)
  • Constraints on the variables:
    • No two queens are in the same column: this is impossible by the choice of the variables!
    • No two queens are in the same row: Xi != Xj, for each i != j
    • No two queens are in the same diagonal: | Xi − Xj | != | i − j | , for each i != j
    • Number of candidate solutions: n^n

• Exhaustive Enumeration
  • Generation of possible values of the variables.
  • Test of the constraints.

• Optimization:
  • Where to place a queen in column k such that it is compatible with rk+1, . . . , rn?
  • Eliminate possible locations as we place queens
Beyond functional programming

- **Applications:**
  - Scheduling
  - Planning
  - Transport
  - Logistics
  - Games
  - Puzzles

- **Complexity**
  - Generally these problems are NP-complete with exponential complexity
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));