ML Function Examples: Polymorphism, Recursion, Patterns, Wildcard Variables, As-bindings, Let-environments, Options, and Basic I/O 
(COP 4020/6021: Programming Languages)

(1) Type variables (i.e., variables ranging over types) must be consistent within a type.

```ml
fun identity(x) = x;
identity: 'a -> 'a
```

Or:

```ml
identity: α→α
```

(I.e., the argument and return types can be anything, but they must be the same.)

(2) Only certain types of values can be tested for equality. Values containing functions or reals (e.g., a list of reals) can’t be tested for equality.

```ml
fun f() = 3.4=3.5;
stdIn:1.12-1.19 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
  3.4 = 3.5
```

```ml
fun f(x,y) = x=y;
stdIn:1.16 Warning: calling polyEqual
val f = fn : ''a * ''a -> bool
```

Or:

```ml
f: α×α → bool
```

(In SML/NJ, two apostrophes before a type variable refers to an equality type.)
(3) ML functions may be recursive.

```ml
fun factorial(n) = (* assumes nonnegative n *)
  if n=0 then 1 else n*factorial(n-1)
```

(4) It’s often more convenient to specify parameters with patterns.

```ml
fun factorial(0) = 1 (* assumes nonnegative n *)
  | factorial(n) = n*factorial(n-1)
```

(5) Patterns are very useful with list parameters.

```ml
fun r(nil) = nil
  | r(x::xs) = r(xs) @ [x];
```

What is r’s type?

What does r do?

Patterns can be: Identifiers (like regular parameters), constants, wildcards (using the symbol: _), or tuples or lists of patterns.

(6) Let’s implement function r using difference lists. One parameter keeps track of work remaining to be done, while another parameter keeps track of work already done.

```ml
fun rDiffLists(nil, processed) = processed
  | rDiffLists(x::xs, processed) = rDiffLists(xs, x::processed);
fun r(L) = rDiffLists(L, nil);
```
More examples of patterns:

- \texttt{fun f(3)=4} \\
  = \quad | \ f(n)=7; \\
val f = fn : int -> int \\
- \texttt{f(5)}; \\
val it = 7 : int \\
- \texttt{f(3)}; \\
val it = 4 : int \\

- \texttt{fun f(3)=4;} \\
  stdIn:1.5-1.11 Warning: match nonexhaustive \\
  3 => ... \\
val f = fn : int -> int \\
- \texttt{f(3)}; \\
val it = 4 : int \\
- \texttt{f(4)}; \\
uncaught exception Match [nonexhaustive match failure]...

Wildcard, a.k.a. anonymous, variables/patterns can replace unused parameters, to unclutter code.

- \texttt{fun f(3)=4} \\
  = \quad | \ f(_)=7; \\
val f = fn : int -> int \\
- \texttt{f 4;} \\
val it = 7 : int \\

As-bindings can prevent having to reconstruct parameters.

\begin{verbatim}
fun inList(pair, nil) = false \\
  | inList(pair as (n,_), (n2,_)::L) = \\
    if n=n2 then true else inList(pair,L);
\end{verbatim}

Equivalently:

\begin{verbatim}
fun inList(pair, nil) = false \\
  | inList((n,n3), (n2,_)::L) = \\
    if n=n2 then true else inList((n,n3),L);
\end{verbatim}

inList : _________________________________

- \texttt{inList( (5,4), [(3,2),(1,0),(4,5)] )}; \\
val it = ________

- \texttt{inList( (5,4), [(3,2),(1,0),(5,5)] )}; \\
val it = ________
(10) Functions can define local values (variables and functions) with let-environments.

fun r(L) = 
  let
    fun rDiffLists(nil, processed) = processed
        | rDiffLists(x::xs, processed) = rDiffLists(xs, x::processed)
  in rDiffLists(L, nil)
end;

(11) Another let-environment example, also illustrating static, versus dynamic, scope.

val v = 5;

fun f(x) = 
  let
    fun g(x) = x+v

    fun h(x) = 
      let val v = 3
      in g(v)
      end

    val v=6
    val _ = v+1
    fun pair(x) = (x,x)
    val (a,b) = pair(5)
  in
    h(v)
  end;

f(1);
Another, more practical example:

```ml
fun maxMiddle(L) = 
    let
        fun findMax(n,nil) = n
            | findMax(n, (_,k,_):L) = findMax(if k>n then k else n, L)
    in findMax(0,L)
end;
```

- `maxMiddle ([ (true,8,5), (true,12,12), (false,4,3) ]);`
  val it = ______

- `maxMiddle [ (5,8,5.0), (5,12,4.3), (4,4,3.0) ];`
  val it = ______

Options are a predefined data type in ML. Options can either be empty or filled with some expression. Values having type “T option” can either be NONE or SOME v (for a value v of type T).

- `SOME(5);`
  val it = SOME 5 : int option
- `NONE;`
  val it = NONE : 'a option
- `SOME "hi";`
  val it = SOME "hi" : string option

- `isSome(NONE);`
  val it = false : bool
- `isSome(SOME 5);`
  val it = true : bool
- `isSome;`
  val it = ____________________________

- `valOf(SOME 5);`
  val it = 5 : int
- `valOf(NONE);`
  ...uncaught exception Option...
- `valOf;`
  val it = ____________________________

As with lists, patterns are convenient for analyzing option arguments.

```
fun sumList(nil) = 0
  | sumList(NONE::ns) = sumList(ns)
  | sumList(SOME(n)::ns) = n+sumList(ns);
```

```
sumList(NONE::SOME(4)::NONE::NONE::SOME(3)::SOME(2)::SOME(1)::[]);
```

The only ML I/O we’ll use in this class is to print strings.
- `print(if true then "hi" else "bye")`;
  hival it = () : unit
- `print`;
  val it = fn : string -> unit

Expression sequences \((e_1;e_2;\ldots;e_n)\) are expressions that allow one subexpression to be executed after another. The result of the expression sequence is the result of executing the last expression, \(e_n\). Expressions \(e_1\) to \(e_{n-1}\) get evaluated just for their side effects (like I/O and memory updates using pointers, which we’ll discuss later in the semester).

- `(print("hi"); "hi")`;
  hival it = "hi" : string

Exercise: Implement a function `printAndAdd : int list->int`, which prints all the elements of the argument list (separated by spaces) and then a newline, and returns the sum of the list elements.