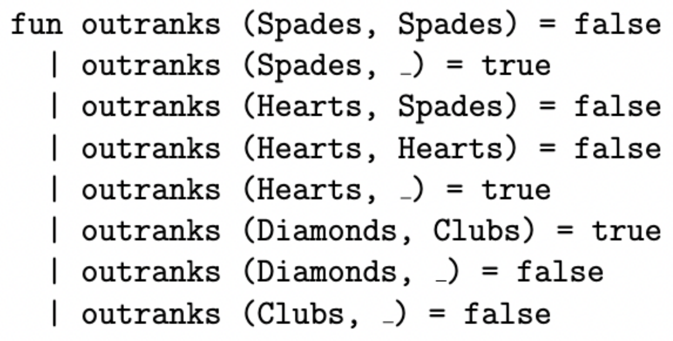
Chapter 9: Programming with Lists

1. List Primitives
    a. Values of type typ list are finite lists of values of type typ
        i. nil is the value of type typ list
        ii. if h is a value of type typ, and t is a value of type typ list, then h::t is a value of type typ list
        iii. Nothing else is a value of type typ list
    b. List is kind of "function" mapping types to types: given a type typ, we may list to it to get another type
    c. When saying h::t, we can say "h cons t"
    d. Example
    val nil : typ list
    val (op ::) : typ * typ list -> typ list
    e. The :: operation takes as its second argument a value of type typ list, and yields a result of type typ list
    f. The op :: constructs a non-empty list independently of the type of elements of the list
    g. The definition of the values of type typ list given above is an example of inductive definition. It is said to be recursive because this definition is "self-referential" in the sense that the values of type typ list are defined in terms of (other) values of the same type
    h. Value val of type typ list has the form
    val1 :: (val2:: (... :: (valn:nil) …)) or
    val1 :: val2 ::...::valn::nil
2. Computing with Lists
    a. Functions on lists are natural to be defined recursively using clausal definition
    b. Example → finding length of list
    fun length nil = 0
    | length (_::t) = 1 + length(t) → do not need to give name to the head
    c. The type of length is ‘a list -> int (Defined for lists of values of any type)
    d. Another example → append list
    fun append(nil, 1) = 1
    | append(h::t, 1) = h::append(t,1)
    e. Another example → reverse list in O(n^2)
    fun rev nil = nil
    | rev(h::t) = rev t @ [h] → @ is the concatenation of lists
    f. Reverse list in O(n)
    local
        fun helper(nil, a) = a

```
            | helper(h::t, a) = helper(t, h::a)
    in
            fun rev' l = helper(l, nil)
    end
```
g. Showing that a function works correctly for every list l
    i. Show correctness of function for empty list, nil
    ii. Show the correctness of function for h::t, assuming its correctness for t

Chapter 10: Concrete Data Types

1. Datatype Declarations
    a. Datatype declaration provides means of introducing new type that is distinct from all other types and that does not merely stand for some other type
    b. Means by which the ML type system may be extended by programmer
    c. Datatype declaration introduces
        i. One or more new type constructors. The type constructors introduced may or may not be mutually recursive
        ii. One or more new value constructors for each of the type constructors introduced by the declaration
    d. Type constructors may take zero or more arguments
    e. Each value constructor may also take zero or more arguments
    f. Nullary value constructor is just a constant
2. Non-Recursive Datatypes
    a. Sample of nullary type constructor with four nullary value constructors
        i. datatype suit = Spades | Hearts | Diamonds | Clubs
    b. No significance to the ordering of constructors in the declaration
    c. Conventional to capitalize names of value constructors
    d. Given declaration, we can define functions on it by case
    e. Example → suit ordering in the card game
        i. 

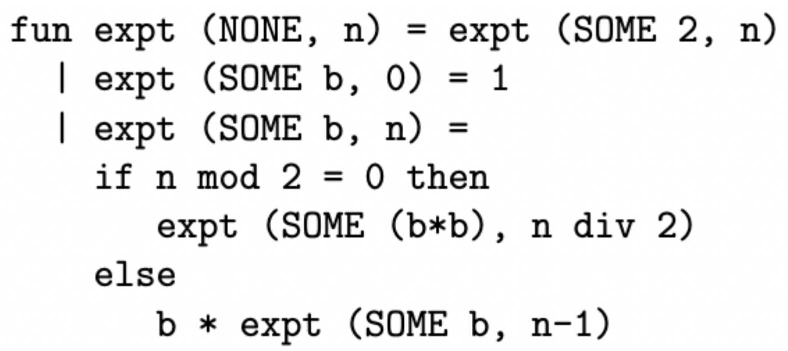
```
fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false
```

        ii. Defines a function of type suit * suit -> bool that determines whether or not the first suit outranks the second
    f. Data types may be parameterized by a type

g. Example
   datatype ‘a option = NONE | SOME of ‘a
   Introduces unary type constructor ‘a option with two value constructors, NONE with no argument and SOME of one argument
   i. The values of the type typ option are
      Constant NONE and
      Values of the form SOME val, where val is a value of type typ

h. Common use of option types is handling functions with optional argument

i. Example → compute base-b exponential function for natural number exponents that defaults to base 2

```
fun expt (NONE, n) = expt (SOME 2, n)
  | expt (SOME b, 0) = 1
  | expt (SOME b, n) =
    if n mod 2 = 0 then
       expt (SOME (b*b), n div 2)
    else
       b * expt (SOME b, n-1)
```

   i.
   ii. The benefit of option type in this sort of situation is that it avoids the need to make a special case of particular argument (using 0 as first argument)

j. Other examples
   i. type entry = {name:string, spouse:string option}
      1. Everybody has name but not spouse, and therefore spouse can have field of NONE
   ii. Reciprocal of function
       fun reciprocal 0 = NONE
       | reciprocal n = SOME (1 div n)
   iii.

3. Recursive Datatypes
   a. Example - Tree of binary trees with values of type typ at nodes
      i. datatype ‘a tree =
         Empty | Node of ‘a tree * ‘a * ‘a tree
      ii. This declaration means that
          1. Empty tree Empty is a binary tree
          2. If tree1 and tree2 are binary trees, and val is value of type typ, then Node(tree1, val, tree2) is binary tree
   b. It is recursive in the sense that binary trees are constructed out of other binary trees
   c. To compute recursive type, use recursive function
   d. Example → function to compute height of binary tree

        i. fun height Empty = 0
           | height (Node(lft, _, rht)) = 1 + max(height lft, height rht)
    e. Example → size of binary tree (number of nodes occurring in it)
        i. fun size Empty = 0
           | size (Node(lft, _, rht)) = 1 + size lft + size rht
    f. Variadic tree → define type of trees with a variable number of children
    g. datatype ‘a tree =
       Empty | Node of ‘a * ‘a tree list
    h. Each node has list of children, so that distinct nodes may have different number of children
    i. Another approach is to simultaneously define trees and “forests” → forest is either empty or variadic tree together with another forest
    j. datatype ‘a tree =
            Empty | Node of ‘a * ‘a forest
       and ‘a forest =
            None | Tree of ‘a tree * ‘a forest
    k. This example illustrates the introduction of two mutually recursive datatypes
    l. Definition of size of variadic tree
        i. fun size_tree Empty = 0
                | size_tree (Node(_,f)) = 1 + size_forest f
           and size_forest None = 0
                | size_forest(Tree (t, f’)) = size_tree t + size_forest f’
    m. Other variation exists
    n. datatype ‘a tree =
            Empty | Node of ‘a branch * ‘a branch
       and ‘a branch =
            Branch of ‘a * ‘a tree
    o. Branches are themselves explicit since data is attached to them
    p. Collect into list the data items labeling the branches of such a tree
    q. fun collect Empty = nil
            | collect(Node(Branch(ld, lt), Branch(rd, rt))) =
                ld::rd::(collect lt) @ (coolect rt)
4. Heterogeneous Data Structures
    a. Returning to original definition of binary trees, type of the data items at the nodes must be same for every node of the tree
    b. For example, a value of type int tree has an integer at every node, and a value of type string tree has a string at every node
    c. Therefore, Node(Empty, 43, Node(Empty, “43”, Empty)) is ill-typed
    d. Creating heterogeneous tree → data item must be labeled with sufficient information so that we may determine the type of the item at run-time

    e. This can be achieved using datatype declaration
    f. datatype int_or_string:
        Int of int | String of string
    g. Then, we can define the type of interest as
       type int_or_string_tree: int_or_string tree
5. Abstract Syntax
    a. Datatype declarations and pattern matching are extremely useful for defining and manipulating the abstract syntax of a language
    b. Example
        i. datatype expr =
            Numeral of int | Plus of expr * expr | Times of expr * expr
    c. Definition of function to evaluate expressions of the language of arithmetic expressions written using pattern matching
        i.

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1+n2)
    end
  | eval (Times (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1*n2)
    end
```

    d. When extending the type expr with new components for reciprocal of a number, we can do
        i. datatype expr =
            Numeral of int | Plus of expr * expr | Times of expr * expr
            | Recip of expr
        ii.

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) = ... as before ...
  | eval (Times (e1, e2)) = ... as before ...
  | eval (Recip e) =
    let
        val Numeral n = eval e
    in
        Numeral (1 div n)
    end
```

Chapter 11: Higher-Order Functions

1. Functions as Values
    a. Functions that take functions as arguments or yield functions as results are known as higher-order functions
    b. Their use may be classified into two broad categories
        i. Abstracting patterns of control: higher-order functions are design patterns that “abstract out” the details of a computation to lay bare the skeleton of solution
    c. Staging computation
        i. It arises frequently that computation may be staged by expending additional effort “early” to simplify the computation of “later” results
2. Binding and Scope
    a. Use of the variable var is considered to be a reference to the nearest lexically enclosing declaration of var
    b. Three important points
        i. Binding is not assignment
        ii. Scope resolution is lexical, not temporal
        iii. “Shadowed’ bindings are not lost. The “old” binding for a variable is still available, even though a more recent binding has shadowed it
3. Returning Functions
    a. Standard example of passing function as argument is the map’ function, which applies a given function to every element of a list
    b. fun map’ (f, nil) = nil
       | map’ (f, h::t) = (f h) :: map’ (f,t)
    c. For example, the application
       map’ (fn x => x +1, [2,3,4,5]) evaluates to [2,3,4,5]
    d. Functions may also yield functions as results.
    e. We can also create new functions during execution, not just return functions that have been previously defined
    f. Most basic example is function constantly that creates constant functions
       val constantly = fn k => (fn a => k)
       OR
       fun constantly k a = k
    g. The function constantly has type ‘a -> (‘b -> ‘a).
    h. The value of the application constantly 3 is the function that is constantly 3 → always yields 3 when applied
    i. The resulting function is created by application of constantly to the argument 3, rather than merely retrieved off the shelf of previously-defined functions
    j. It takes a function and list as argument, yielding a new list as result

k. Instead we would prefer to create a instance of map specialized to the given function that can be applied to many different lists → function map

l. fun map f nil = nil
   | map f (h::t) = (f h) :: (map f t)

m. The function has type (‘a -> ‘b) -> ‘a list -> ‘b list

n. The passage from map’ to map is called currying

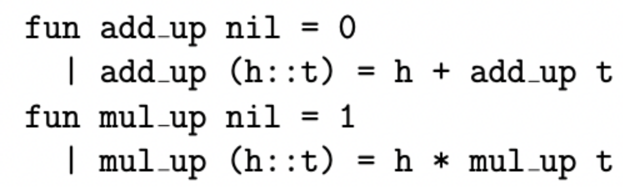
o. We have changed a two-argument function into a function that takes two arguments in succession, yielding after the first a function that takes the second as its sole argument

p. fun curry f x y = f(x,y)

q. The type of curry is (‘a * ‘b -> ‘c) -> (‘a -> (‘b -> ‘c))

4. Patterns of Control

   a. There is similarity between the following two functions

   b. 

```
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t
```

      i. One view is that in each case, we have ab inary operation and a unit element for it. The result on the empty list is the unit element and the result on a non-empty list is the operation applied to the head of the list and the result on the tail

   c. This pattern can be abstracted as function reduce defined as
      fun reduce (unit, opn, nil) =
      unit | reduce (unit, opn, h::t) = opn(h, reduce(unit, opn, t))

   d. Type of reduce: ‘b * (‘a * ‘b -> ‘b) * ‘a list -> ‘b

   e. Using reduce, we may re-define add_up and mul_up as
      fun add_up l = reduce (0, op +, 1)
      fun mul_up l = reduce(1, op *, 1)

   f. Another view is that they are both defined by induction on the structure of the list argument, with base case for nil and inductive case for h::t → they are defined in terms of a unit element and a binary operation

   g. In other words, the function reduce abstracts the pattern of defining a function by induction on the structure of a list

   h. The definition of reduce leaves something to be desired. Arguments unit and opn are carried unchanged through the recursion but only the list parameter changes on each recursive call → each time around the loop we are constructing a new tuple whose first and second components remain fixed, but third component varies

```
fun better_reduce (unit, opn, l) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red l
    end
```

i.

5. Staging
    a. Interesting variation on reduce may be obtained by staging the computation
    b. The motivation is that unit and opn often remain fixed for many different lists → they are "early" arguments and the list is said to be a "late" argument
    c. The idea of staging is to perform as much computation as possible on the basis of the early arguments, yielding function of the late arguments alone
    d. In reduce, this amounts to building red on the basis of unit and opn

```
fun staged_reduce (unit, opn) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red
    end
```

    e.
    f. The only difference is that the creation of the closure bound to red occurs as soon as unit and opn are known, rather than each time the list argument is supplied
    g. The overhead closure creation is "factored out" of multiple applications of the resulting function to list arguments
    h. Consider the following definition of append function for lists that takes both arguments at once
    i. fun append(nil, l) = l
            | append(h::t, l) = h :: append(t,1)
    j. Naive solution that merely curries append:
    k. fun curried_append nil l = l
            | curried_append (h::t) l = h :: curried_append t l
    l. This solution doesn't exploit the fact that the first argument is fixed for many second arguments.
    m. Each application of the result of applying curried_append to a list results in the first list being traversed so that the second can be appended to it

n. Improved version:

```
fun staged_append nil = (fn l => l)
  | staged_append (h::t) =
    let
        val tail_appender = staged_append t
    in
        fn l => h :: tail_appender l
    end
```

o. Here, first list is traversed once for all applications to a second argument and therefore is more efficient