Chapter 5: Products and Records

1. Product Types
    a. Data structures (tuples, lists, arrays, trees) can be created and manipulated easily
    b. Not necessary in ML to be concerned with allocation and deallocation of data structures, nor with any particular representation strategy involving
        i. n-tuple (finite ordered sequence of values of the form (val_1, …, val_n)
        ii. n-tuple is a value of a product type of the form
        iii. Example
            1. val pair : int * int = (2,3)
            2. val triple: int * int * string = (2, 2.0, "2")
            3. val pair_of_pairs = (int * int) * (real * real) = ((2,3), (2.0,3.0))
            4. val null_tuple : unit = ()
        iv. ML also provides general tuple expression of the form (exp_1, …, exp_n) where each exp_i is an arbitrary expression, not a value.
        v. The expression is calculated from left to right
        vi. Example
            1. val pair: int * int = (1+1, 5-2)
        vii. We can write the same expressions as
            let
                val x1 = exp1
                …
                val xn = expn
            in
                (x1, …, xn)
            end
        viii. Powerful feature of ML is the usage of pattern matching to access components of aggregate data structures
        ix. Suppose that we have type of val as (int * string) * (real * char)
        x. Getting first component of second component can be written as val ((_, _), (r:real, _)) = val → instead of navigating to the position to retrieve it, simply use generalized form of value binding
        xi. The left hand side of val is a tuple pattern that describes pair of pairs
        xii. Underscores indicate "don't care" positions in the pattern → values are not bounded to any variable
        xiii. Give names to all components
            val ((i:int, s:string), (r:real, c:char)) = val
            val (is: int* string, rc: real*char) = val
        xiv. General form of value binding is
            val pat = exp where pat is pattern and exp is expression

xv. Pattern of
    1. A variable pattern of the form var : typ
    2. A tuple pattern of the form (pat_1, …, pat_n), where each pat is pattern. This includes as a special case of the null-tuple pattern
    3. A wildcard pattern of the form _

xvi. Type of a pattern is determined by inductive analysis of the form of the pattern:
    1. A variable pattern var:typ is of type typ
    2. A tuple pattern(pat_1, …,pat_n) has the type typ1 * … * typn, where each pat is a pattern of type typ.
    3. The wildcard pattern _ has any type whatsoever

xvii. Value bindings are evaluated using the bind-by-value principle mentioned earlier.

xviii. First, evaluate the right-hand side of the binding to a value

xix. Second, we perform pattern matching o determine the bindings for the variables in the pattern

xx. The rules are following:
    1. The variable binding val var = val is irreducible
    2. The wildcard binding val _ = val is discarded
    3. The tuple binding val(pat1,...,patn) = (val1,...,valn) is reduced to set of n bindings
       val pat1 = val1
       …
       val patn = valn

xxi. For example
    val ((m:int, n:int), (r:real, s:real)) = ((2,3), (2.0,3.0))
    1. First, compose binding into two bindings
       val (m:int, n:int) = (2,3)
       and (r:real, s:real) = (2.0, 3.))
    2. Decompose each in turn, resulting in
       val m: int 2
       and n:int = 3
       and r:real = 2.0
       and s:real = 3.0

2. Record Types
    a. Tuples are most useful when the number of positions is small
    b. When number of components grows beyond small number, it is more natural to attach a label to each component of the tuple that mediates access to it, which is record type
    c. {lab1:typ1, …, labn: typn} where n >= 0 and all labels are distinct

d. Record value has form {lab1:val1, …, labn: valn} where val has type typ
e. Record pattern has the form {lab1:pat1, …, labn: patn} which has type {lab1:typ1, …, labn: typn}
f. Record value binding of the form
   val {lab1:pat1, …, labn: patn} = {lab1:val1, …, labn: valn} is decomposed to
   val pat1 = val1
   and …
   and patn = valn
g. Components of record are identified by name, not position, and therefore the order in which they occur in a record value or record pattern is insignificant
h. However, in record expression, fields are evaluated from left to right
i. Example
   i. type hyperlink = {protocol: string, address: string, display: string}
   ii. The record binding
       val mailto_rwh: hyperlink = {protocol = “mailto”, address =”rwh@cs.cmu.edu”, display = “Rober Harper”} defines variable of type hyperlink
   iii. The record binding
       val {protocol = prot, display = disp, address = addr} - mailto_rwh decomposes into three variable bindings
       val prot = “mailto”
       val addr = “rwh@cs.cmu.edu”
       val disp = “Robert Harper” which extract the values of the fields of mailto_rwh
j. Using wild cards, we can extract selected fields from record
   val {protocol = prot, address = _, display = _} = mailto_rwh
k. When having tons of fields, we can select one or two fields by using
   val {protocol = prt, …} = mailto_rwh, which is ellipsis patterns in records
l. The … stands for expanded pattern
m. For this to occur, compiler must be able to determine unambiguously the type of the record pattern
n. Finally, ML provides convenient abbreviated form of record pattern
   {lab1, … labn} which stands for pattern {lab1 = var1,..., labn = varn}
o. For example,
   val {protocol, address, display} = mailto_rwh
   decomposes into sequence of atomic bindings
   val protocol = “mailto”
   val address = “rwh@cs.cmu.edu”
   val display = “Robert Harper”

3. Multiple Arguments and Multiple Results
    a. Function may bind more than one argument by using pattern, rather than a variable
    b. For example,
       val dist: real * real -> real
       = fn (x:real, y:real) => Math.sqrt(x*x + y*y)
    c. The function may then be applied to a two-tuple (pair) of arguments
    d. For example, dist(2.0, 3.0) evaluates to approximately 4.0
    e. The following can also be written as
       fun dist(x:real, y:real): real => Math.sqrt(x*x + y*y)
    f. Keyword parameter passing is supported through the use of record patterns
    g. For example,
       fun dist' {x = x:real, y=y:real} = Math.sqrt(x*x + y*y)
    h. The expression dist' {x=2.0, y=3.0} invokes this function with indicated x and y values
    i. Functions can also yield tuples or records
    j. Example,
       fun dist2(x:real, y:real): real * real =
       (Math.sqrt(x*x + y*y), Math.abs(x-y))
    k. We can also get certain element of tuple by using #i of type (sharp notation)
    l. For example, fun #i (_, …,_, x, _, …,_) = x where x occurs in the ith position of the tuple
    m. Thus, we may refer to the second field of three-tuple val by writing #2(val)
    n. However, it is bad style, and is strongly discouraged

Chapter 6: Case Analysis

1. Homogeneous and Heterogeneous Type
    a. Tuple types have property that all values of that type have the same form (homogeneous)
    b. For example, values of type int*real are pairs whose first component is an integer and whose second component is real
    c. Error will occur if we try to match type int*real*string (fail at compile time)
    d. Other types have values of more than one form (heterogeneous type)
    e. Value of type int might be 0, 1, ~1, … or a value of type char might be #"a" or #"z"
    f. Corresponding to each of the values of these types is a patten that matches only that value
    g. For example,
       val 0 = 1-1

val (0,x) = (1-1, 34)
val(0, #”0”) = (2-1, #”0”) → fails

2. Clausal Function Expressions
    a. Define functions over heterogeneous types → achieved in ML using clausal function expression that has form
       fn pat1 => exp1 | … |patn => expn
    b. Each component pat => exp is called clause or rule. Entire rule is called match
    c. The typing rules for matches ensure consistency of the clauses. There must exist typ1 and typ2 such that
        i. Each pattern pat has type typ1
        ii. Each expression exp has typ2, given the types of the variables in pattern pat
    d. Example
       val recip: int -> int =
       fn 0 => 0 | n:int => 1 div n
    e. This defines reciprocal function, where reciprocal of 0 is artbirarily defined to be 0. The function has two clauses, one for argument 0 and other for non-zero arguments n
    f. Using function notation, we can write
       fun recip 0 = 0 | recip (n:int) = 1 div n
3. Booleans and Conditionals, Revisited
    a. Type bool of booleans is perhaps the most basic example of heterogeneous type
    b. The conditional expression if exp then exp1 else exp2 is short-hand for
       case exp of true => exp1 | false => exp2 which is
       (fn true => exp1 | false => exp2) exp
    c. “Short-circuit” conjunction and disjunction operations are defined as
       if exp1 then exp2 else false
       and the expression exp1 orelse exp2 is short for
       if exp1 then true else exp2
4. Exhaustivness and Redundancy
    a. Exhaustiveness checking ensures that well-formed match covers its domain type in the sense that every value of the domain must match one of its clauses
    b. Redundancy checking ensures that no clause of a match is subsumed by the caluses that precede it. It means that the set of values covered by a clause in a match must not be contained entirely within the set of values covered by the preceding clauses of that match
    c. For example,
       fun recip(n: int) = 1 div n | recip 0 = 0 → the second clause is redundant since 0 is included in the integer
    d. Inexhaustive matches may or may not be in error

e. Error example

i.
```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
```

ii. When applied #”a”, function fails → function never returns false for any argument

iii. Need catch-all clause at the end

iv.
```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
    | is_numeric _ = false
```

v. Addition of final catch-all clause renders match exhaustive, because any value not matched by the first ten clauses will surely be matched by eleventh

Chapter 7: Recursive Functions

1. Self-Reference and Recursion
    a. Function calling itself → function refer to itself
    b. Simplest form of recursive value binding
       val rec var:typ = val
    c. Example
        i. val rec factorial: int -> int =
           fn 0 => 1 | n: int => n* factorial (n-1)
        ii. fun factorial 0 = 1
           | factorial (n:int) = n * factorial (n-1)
    d. Type checking is important in recursion

e. To check that binding for factorial is well-formed, we must check that each clause has type int->int by checking for each clause that its pattern has type int and that its expression has type int.
f. Then, we check n * factorial(n-1) has type int

2. Iteration
   a. Definition of factorial given above should be contrasted with the following two-part definition
   b. fun helper(0, r:int) = r | helper (n:int, r:int) = helper(n-1, n*r)
      fun factorial (n:int) = helper (n,1)
   c. Helper function here takes two parameters, an integer argument, and accumulator that records running partial result of the computation
   d. Accumulator re-associates pending multiplications in evaluation trace given above so that they can be performed prior to the recursive call
   e. Programming style → conceal definitions of helper functions using local/let declaration
   f. Example
      i.
```
local
      fun helper (0,r:int) = r
        | helper (n:int,r:int) = helper (n-1,n*r)
in
      fun factorial (n:int) = helper (n,1)
end
```
3. Inductive Reasoning
   a. Time and space usage are important
   b. Key to the correctness of a recursive function is an inductive argument establishing its correctness
   c. Critical ingredients:
      i. An input-output specification of the intended behavior stating pre-conditions on the arguments and post-condition on the result
      ii. Proof that the specification holds for each clause of the function, assuming that it holds for any recursive calls
      iii. Induction principle that justifies the correctness of the function as a whole, given the correctness of its clauses
   d. Example of complete induction → Fibonacci function on integers n >= 0
      i. fun fib 0 = 1
         | fib 1 = 1
         | fib (n: int) = fib(n-1) + fib(n-2)
4. Mutual Recursion
   a. Useful to define two functions simultaneously, each of which calls the other to compute its result → called mutually recursive
   b. Example -> test whether the number is even or odd

i. fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even (n-1)

ii. Here, the integer is odd only if n-1 is even and integer is even only if n-1 is odd → they are mutually related, which is why we use two mutually-recursive procedures