Elements Of ML Programming, Jeffrey D. Ullman

Chapter 1: A Perspective on ML and SML/NJ

Why programming in SML:

- A functional language, meaning that the basic mode of computation is the definition and application of functions.
- Side – effect freedom, because there aren't assignments, it evaluates only expressions.
- High – order functions, these are functions that take functions as arguments.
- Polymorphism, which is the ability of a function to take arguments of various types.
- Abstract data type, called structures, offer the power of "classes" used in object – oriented programming languages like Java.
- Recursion, iteration is done by recursion.
- Rule – based programming, done by pattern – matching.
- Strong – typing, meaning that all valued and variables have a type that can be determined at "compile time", so many errors can be caught by the compiler.

Chapter 2: Getting Started in ML

Integers:

- Positive integers: 123, 0, 3111
- Negative integers: ~123, ~10, ~1111
- Hexadecimal: 0x1234 $\Rightarrow$ *val it = 4660:int*

Reals:

- Positive reals: 123.01 $\Rightarrow$ *val it = 123.01:real*
- Negative reals: ~100.0
- Scientific reals: 3E~3 $\Rightarrow$ 0.003 , 3.14e12 $\Rightarrow$ $3.14 \cdot 10^{12}$

Boolean:

- *true* $\Rightarrow$ *val it = true: bool*
- *false*

Strings:

- Example: "R2D2" $\Rightarrow$ *val it = "R2D2":string*
- *\n* means new line

Characters:

- Example: #"x" represents character x $\Rightarrow$ *val it = #"x": char*

Arithmetic Operators:

- Additive operators: +, -
- Multiplicative operators: *, / (division of reals), div (division of integers), mod
- Unary minus operator: ~

String Operators:

- Concatenate two strings: ^
- Example: "house" ^ "cat"; $\Rightarrow$ *val it = "housecat": string*

Comparison Operators:

- =, < , >, <=, >=, <>
- They can be used to compare integers, reals, characters and strings.
- Reals may not be compared using = or <>. How can you test equality instead?
  ⇒ Use: r <= s and s <= r

Logic Operators:

- Negation: *not*
- And: *andalso*
- Or: *orelse*
- ML doesn't evaluate the right term if the left term is true in an *orelse*. (the "same" for andalso).

IF – THEN – ELSE expression:

- *if E then F else G*
- Example: *if* 1 < 2 *then* 3 + 4 *else* 5 + 6; ⇒ *val it = 7:int*
- There is no *if … then…* , p.18.

Type Consistency:

- The operator + can take either integer or real arguments. However, both operands must be the same type.
- Every operator needs arguments of the same type.
- The expression following *then* end *else* can be of any one type, but they must be of the same type.
- The word "tycon" is short for "type constructor", that is, a way of constructing types from simpler types.

Type Converting:

- Integer to real:
  - *real*(4); ⇒ *val it = 4.0:real*
- Real to integer:
  - *floor*(x), produces the greatest integer that is no larger than x.
  - *ceil*(x), produces the smallest integer no less than x.
  - *round*(x), produces the closest integer with 0.5 raised to the next highest integer.
  - *trunc*(x), drops digits to the right of the decimal point.
- Characters to integer:
  - *ord*(#"a"); ⇒ *val it = 97:int*
- Integers to characters:
  - *chr*(97); ⇒ *val it = #"a": char*
- Characters to strings:
  - *str*(#"a"); ⇒ *val it = "a": string*
- String to characters:
  - *explode*("abcde"); ⇒ *val it = [#"a", #"b", #"c", #"d", #"e"]: char list*
  - *explode*(""); ⇒ *val it = []: char list*
- Characters to string:
  - *implode*([#"a", #"b", #"c", #"d", #"e"]); ⇒ *val it = "abcde": string*
  - *implode*(nil); ⇒ va*l it = "": string*

Type Variables:

- Example: 'a, 'b, 'a list
- This variables cannot be bound to values like 3, 4.5, "size", or any of the values we normally think of as the values of variables. It can only be bound to a type, like *int* or *real*.
- For instance, 'a might in some context be given the type integer as its "value".

Environments:

- The Top – Level Environment is the collection of predefined "functions" in ML, for example: *floor*, ^, *trunc*, …
- Above this environment we can add our own identifiers bound to some types.

foo	10
pi	3.14159
…	…
^	function to concatenate
floor	compute floor of real.
…	…

- It is possible to add an identifier to the current environment and bind it to a value:
  - *val* <*identifier*> = <*value*>
  - Example: *val* pi = 3.14159 ⇒ *val pi = 3.14159:real*
- In ML, the *val* – declaration causes a new entry in the environment to be created (doesn't matter when the name already exists, a new one is created), associating what is to the left of the equal sing with the value to the right of the equal sing, p32, example 2.24. ⇒ There are no side effects as in imperative programming languages exists.

Tuples:

- The simplest and possibly most important ways of constructing types in ML are notations for forming tuples, which are similar to record types in Pascal or C, and for forming lists of elements of a given type.
- A tuple is formed by taking a list of two or more expressions of any type, separating them by commas, and surrounding them by round parentheses. Thus, a tuple looks something like a record, but the fields are named by their position in the tuple rather than by declared field names.
- Example:
  - *val* t = (4, 5.0, "six"); ⇒ *val t = (4, 5.0, "six"): int * real * string*
  - *val* u = (1, (2, 3.0)); ⇒ *val u = (1, (2, 3.0)): int * (int * real)*
- Accessing components of tuples:
  - To get the first value of t: #1(t); ⇒ *val it = 4:int*
  - To get the second value of t: #2(t); ⇒ *val it = 5.0:real*

Lists:

- ML provides a simple notation for lists whose elements are all of the same type. We take a list of elements, separate them by commas, and surround them with square brackets.
- The :: operator needs a constant time to be performed, while the @ operator always needs time proportional to the length of the list.
- Precedence: *string * int list * int* ⇒ *string * (int list) * int*
- Example:
  - [1, 2, 3]; ⇒ *val it = [1, 2, 3]: int list*
  - [[(1, 2), (3, 4)], [(5, 6)], nil]; ⇒ *val it = [[(1, 2), (3, 4)], [(5, 6)], nil]:(int * int) list list*
- Empty list: *nil* or []
- Head and Tail:
  - Head of [2, 3, 4] is 2 and Tail is [3, 4].
  - Head of [5] is 5, and Tail is [].
  - Access to head and tail:
    - *val* L = [2, 3, 4]; *hd*(L); ⇒ *val it = 2:int*
    - *val* L = [2, 3, 4]; *tl*(L); ⇒ *val it = [3, 4]: int list*

- Concatenation of lists with @ operator:
  - @ needs two lists as arguments.
  - Example: [1, 2] @ [3, 4]; ⇒ *val it = [1, 2, 3, 4]: int list*
- Cons operator:
  - The left operand must be element and the right operand must be a list.
  - Example:
    - 2 :: [3, 4]; ⇒ *val it = [2, 3, 4]: int list*
    - 2.0 :: nil; ⇒ *val it = [2.0]: real list*
  - Cascading of cons operator:
    - 1 :: 2 :: 3 :: nil ⇒ 1 :: (2 :: (3 :: nil))
    - 1 :: 2 :: 3 :: nil ⇒ [1, 2, 3]
- Concatenation of string lists:
  - Example: *concat*([“ab”, “cd”, “e”]); ⇒ *val it = “abcde”: string*

Chapter 3: Defining Functions

Definitions and Properties:

- Polymorphic Functions: Functions that can take arguments of different types.
- High – Order Functions: Functions that take functions as arguments or produce functions as results.
- Currying of High – Order Functions: Writing a function so new functions may be created by instantiating one of its arguments.
- ML is strongly typed, meaning it is possible to determine the type of any variable or the value returned by any function by examining the program, but without running the program. Put another way, an ML program for which it is not possible to determine the types of variables and function return – values is an incorrect program.
- The algorithm whereby ML deduces the types of variables is complex.
- Although we must be able to tell the types of all variables in a complete program, we can define functions whose types are partially or completely flexible; these are the Polymorphic functions.

Function Declaration:

- *fun <identifier> (<paramter list>) = <expression>;*
- Example:
  - *fun* upper(c) = *chr*(*ord*(c) – 32); ⇒ *val upper = fn: char ® char*
  - upper(#”a”); ⇒ *val it = #”A”: char*
- Function types:
  - *fn: <domain type> ® <range type>*
- Operator → is right – associative, so $T_1 \rightarrow T_2 \rightarrow T_3$ is interpreted as: $T_1 \rightarrow (T_2 \rightarrow T_3)$
- Declaring function types:
  - *fun* square(x: real) = x * x; ⇒ *val square = fn: real ® real*
- Example of functions that reference external variables at p.52, example 3.7

Linear Recursion: inverts a list

```
(1) fun reverse(L) =
(2)         if  L = nil then nil
(3)         else  reverse(tl(L)) @ [hd(L)];
```

Nonlinear Recursion: Computes n choose m

```
(1) fun comb(n, m) =
(2)         if  m = 0 orelse m = n then 1
(3)         else  comb(n – 1, m) + comb(n – 1, m – 1);
```

Mutual Recursion:

- *fun*
  <definition of first function>
  and
  <definition of second function>
  and
  …
  and
  <definition of nt function>
- This is used, if several recursive functions call each other.
- Example:
  (1) fun
  (2) take(L) =
  (3) *if* L = nil *then* nil
  (4) *else* hd(L)::skip(tl(L))
  (5) and
  (6) skip(L) =
  (7) *if* L = nil *then* nil
  (8) *else* take(tl(L));
- take([1, 2, 3, 4, 5]); ⇒ *val it = [1, 3, 5]: int list*
- skip([1, 2, 3, 4, 5]); ⇒ *val it = [2, 4]: int list*

How ML Deduces Types:

- The types of the operands and result of arithmetic operators must all agree. For example, if the use of + produces a real, then both its operands are real. They will also have a real value any other place they are used, which can help make further inference.
- When we apply an arithmetic comparison, we can be sure the operands are of the same type, although the result is a boolean and therefore not necessarily of the same type as the operands. For example, in the expression a <= 10, we can deduce that a is an integer.
- In a conditional expression, the expression itself and the sub expressions following the then and else must be of the same type.
- If a variable or expression used as an argument of a function is of a known type, then the corresponding parameter of the function must be of that type. Similarly, if the function parameter is of known type, then the variable or expression used as the corresponding argument must be of the same type.
- If the expression defining the function is of a known type, the the function returns a value of that type.
- If no way to determine the type of a particular use of an overloaded operator exists, then the type of that operator is defined to be the default for that operator, normally integer.

Patterns as Function Parameters:

- How ML matches patterns, look at page 72.
- *fun <identifier> (<first pattern>) = <first expression>*
  | <identifier> (<second pattern>) = <second expression>
  | …
  | <identifier> (<last pattern>) = <last expression>;
- Example 1:
  (1) *fun* reverse(nil) = nil
  (2) | reverse(x::xs) = reverse(xs) @ [x] ;
- Example 2 (good example for nested lists!):
  (1) *fun* sumLists(nil) = 0
  (2) | sumLists(nil::Ys) = sumLists(Ys)
  (3) | sumLists((x::xs)::Ys) = x + sumLists(xs::Ys);
  - sumList([[1, 2], nil, [3, 4, 5], [6]]); ⇒ *val it = 21: int*

- Multiple uses of the same variable in pattern is illegal
  - …. comb(n, n) = 1 …
  - This is illegal, because n occurs twice.

The *as* Operator:

- *<identifier> as <pattern>*
- Example:

```
(1) fun  merge(nil, M) = M
(2) |    merge(L, nil) = L
(3) |    merge(L as x::xs, M as y::ys) =
(4)         if x = y then x::merge(xs, M)
(5)         else  y::merge(L, ys);
```

Anonymous Variables:

- The symbol _ can be used in pattern to stand for an anonymous or wildcard variable.
- Example: *fun* comb(_, 0) = 1 ….

What aren't Patterns:

- The @ operator: length(xs @ [x]) = 1 + length(xs); Doesn't work!
- The arithmetic operators: square(x + 1) = 1 + 2*x + square(x); Doesn't work!
- Real constants: *fun* f(0.0) = 0; Doesn't work!

Local Environments Using *let*:

- Sometimes we need to create some temporary values – that is, local variables – inside a function.
- *let*
  val < first variable> = <first expression>;
  val <second variable> = <second expression>;
  …
  val <last variable> = <last expression>;
  in
  <expression>
  end;
- Example 1:

```
(1) fun  hundredthPower(x: real) =
(2)    let
(3)       val four = x * x * x * x;
(4)       val twenty = four * four * four * four * four;
(5)    in
(6)       twenty * twenty * twenty * twenty * twenty * twenty
(7)    end;
```

- Example 2:

```
(1) fun split(nil) = (nil, nil)
(2) |   split([a]) = ([a], nil)
(3) |   split(a :: b :: cs) =
(4)        let
(5)           val (M, N) = split(cs);
(6)        in
(7)           (a :: M, b :: N)
(8)        end;
```

  - split([1, 2, 3, 4, 5]); ⇒ *val it = ([1, 3, 5], [2, 4]): int list * int list*

Functions Using Difference Lists:

- There is a trick known to LISP programmers as difference lists, in which one manipulates lists more efficiently by keeping, as an extra parameter of your function, a list that represents in some way what you have already accomplished.
- Example:
  (1) *fun* rev1(nil, M) = M
  (2) | rev1(x :: xs, ys) = rev1(xs, x :: ys);
  (3) *fun* reverse(L) = rev1(L, nil);
- This function to reverse a list takes only time O(n) whereas the reverse function we used earlier takes $O(n^2)$.

Chapter 4: Input and Output

The *print* Function:

- To print a string of characters to the standard I/O use *print*(x).
- Example:
  (1) *fun* testZero(0) = *print*("zero\n")
  (2) | testZero(_) = *print*("not zero\n");
  - *val testZero = fn: int ® unit*

The Type *unit*:

- The *unit* type is another of the basic types of the ML system, like *int*. In a sense it is like the C type void. However, while there is no value for a "void" in C, the ML *unit* type has exactly one value, ().
- Example: *fun* hello( ) = "hello world"; ⇒ *val hello = fn: unit ® string*

Printing Nonstring Values:

- Reel number printing:
  (1) *val* x = 1.0E50; ⇒ *val = 1e50: real*
  (2) *print*(*Real.toString*(x)); ⇒ *1e50val it = ( ): unit*
- Integer number printing:
  - *print*(*Int.toString*(123)); ⇒ *123val it = ( ): unit*
  - "123val" because we used no new line.
- Boolean printing:
  - *print*(*Bool.toString*(true)); ⇒ *trueval it = ( ): unit*

"Statement" Lists:

- Technically, there is no such thing as a "statement" in ML, only expressions. However, expressions that cause side – effects behave much like statements of ordinary languages.
- *(<first expression>; … ; <last expression>)*
- Example:
  (1) *fun* printList(nil) = ( )
  (2) | printList(x :: xs) = (
  (3) print(Int.toString(x));
  (4) print("\n");
  (5) printList(xs)
  (6));
  - *val printList = fn: int list ® unit*
  - printList([1, 2, 3, 4]);
    1
    2
    3
    val it = ( ): unit

Opening Structures:

- An entire structure can be "raised" to the surface by the *open* command.
- Example:
  (1) *open Int*;
  (2) *print*(*toString*(123)); (* is now legal *)
  (3) *print*(*toString*(12.34)); (* is illegal *)

Reading Input From a File and Writing to a File:

- Look at pages: 108 – 120.

Chapter 5: More About Functions

Matches and Patterns:

- A match expression consists of one or more rules, which are pairs of the form:
- *<pattern 1> => <expression 1>* |
  <pattern 2> => <expression 2> |
  …
  <pattern n> => <expression n>
- Each of the expression following the => must be of the same type.
- Using matches to define functions:
  - *val rec f = fn <match>*
  - The keyword *rec*, short for "recursive", is necessary only if the function f is recursive, that is, if the identifier f appears in one or more expressions of the match.
  - Exapmle:
    (1) *val rec* reverse = fn
    (2) nil => nil |
    (3) x :: xs => reverse(xs) @ [x];
  - *fun* f(P1) = E1
    | f(P2) = E2
    ….
    | f(Pn) = En;
  - This is equivalent to :
  - *val rec* f = fn
    P1 => E1 |
    P2 => E2 |
    …
    Pn => En;

Anonymous Functions:

- (*fn* x => x + 1) (3); ⇒ *val it = 4: int*

Case Expression:

- *case <expression> of <match>*
- First version:
  (1) *case* x < y of
  (2) *true* => #"a" |
  (3) *false* => #"b";
- This second version, is equivalent to the first version:
  (1) *if* x < y *then* #"a" *else* #"b";

Exceptions:

- First you have to declare the exception (two ways):
  - *exception* Foo;
  - *exception* Foo *of string*;
- Then later in the program, you can raise an exception.
  - *raise* Foo;
  - *raise* Foo("bar");
- Example: look at p.135

Handling Exceptions:

- <*expression*> *handle* <*match*>
- Example: look at p.138

Polymorphic Functions:

- The ability of a function to allow arguments of different types is called polymorphism, and such a function is called Polymorphic.
- Example:
  - *fun* identity(x) = x; ⇒ *val identity = fn: 'a ® 'a*
  - identity(2); ⇒ *val it = 2: int*
  - identity(*ord*); ⇒ *val it = fn: char ® int*
    - § We can even give the identity function a function as an argument; it will produce that function as result.
  - identity(2) + *floor*(identity(3.5)); ⇒ *val it = 5: int*
    - § We can even apply the function identity twice in the same expression, using it on values of different types, as long as no type error is thereby introduced.
- A type variable, such as 'a, has two meanings:
  - A type variable 'a can say "for every type T, there is an instance of this object with type T in place of 'a". Such a type variable is called generalizable. The primary example of such a use is in descriptions of the type of Polymorphic functions. For instance, the type 'a → 'a used to describe the type of the function identity in the example above represents such a type schema, where the function identity can be used with any type, even in the same expression.
  - A type variable 'a can represent any one type that we choose. However, once that type is selected, the type cannot change, even if we reuse the object whose type was described using the type variable 'a. A type variable of this kind is nongeneralizable.
- What is illegal:
  - If we apply a function to an argument (could also be a function), and the type of the result has type variables (as e.g. 'a), then these expression is illegal.
    - § identity(identity); ⇒ illegal
    - § identity(identity: *int* → *int*); ⇒ legal
  - It is illegal to build a list of functions which returns type variables.
    - § [identity, identity] ⇒ illegal
  - If the expression is not at top level; that is, the expression is a subexpression of some larger expression, then this expression is legal in any case (A restriction at p.148, figure 5.8).

    (1) let
    (2) *val* x = identity(identity)
    (3) in
    (4) x(1)
    (5) *end*;
    ⇒ *val it = 1: int*

The Equality Operator:

- Most basic types – integer, boolean, character and string – are equality types. The two ways to form more equality types are:
    1. Forming products of equality types (tuples)
    2. Forming a list whose elements are of an equality type.
- Note that rules (1) and (2) can be applied recursively.
- Examples for equality types:
    - *int * int*
    - *(int * int) list*
    - *int list * string*

    (1) *val* x = (1, 2);
    (2) *val* y = (2, 3);
    (3) x = y; ⇒ *val it = false: bool*
    (4) x = (1, 2); ⇒ *val it = true: bool*
- Restriction: Functions cannot be compared with = or <>. Exception, look at p.153, figure 5.10.
- ML has a build – in function *null* that tests whether a list is empty without requiring that list to be of an equality type.
- Example:
    - … *if null*(L) *then* nil …
    - This should be used, when list L contains functions as elements
- Remember that type variables (like ‘a) whose values are restricted to be an equality type are distinguished by having names that begin with two quote marks (like ‘’a) rather than only one.
- Both *types* and *datatypes* may be used to define more equality types.
    - A type is an equality type if the type that it stands for is an equality type.
    - A datatype is an equality type if its constructor expressions, if any, involve only equality types or the datatype itself.
    - A mutually recursive collection of datatypes are equality types if their constructor expressions involve only equality types and the datatypes in the collection.

Simulating Iterations by Recursion:

- The general idea is to write a function that, as a basic case, tests if the loop is done. For the induction it does one iteration of the loop and then calls itself recursively to do whatever iterations of the loop remain. The arguments of the function are the loop index and any other variables that are needed in the loop. The hard part in designing the function often is deciding how to express the result of the loop as a value to be returned by the function.

High – Order Functions:

- Functions that takes functions as arguments and/or produce functions as values are called higher – order functions.
- The following 3 functions (map, reduce, filter) are such higher – order functions.

Simple Map Function:

- The map function takes a function F and a list $[a_1, a_2, ..., a_n]$ and produces the list $[F(a_1), F(a_2), ..., F(a_n)]$. That is, it applies F to each element of the list and returns the list of resulting values.
- Definition:

(1) *fun* simpleMap(F, nil) = nil
(2) | simpleMap(F, x :: xs) = F(x) :: simpleMap(F, xs);
(3) ⇒ *val map = fn: (‘a ® ‘b) * ‘a list ® ‘b list*

- Example 1:
  (1) *fun* square(x:*real*) = x * x;
  (2) simpleMap(square, [1.0, 2.0, 3.0]);
  (3) ⇒ *val it = [1.0, 4.0, 9.0]: real list*
- Example 2:
  (1) simpleMap(~, [1, 2, 3]);
  (2) ⇒ *val it = [~1, ~2, ~3]: int list*
- Example 3:
  (1) simpleMap(*fn* x => x * x, [1.0, 2.0, 3.0]);
  (2) ⇒ *val it = [1.0, 4.0, 9.0]: int list*
- Comment: Notice that in the definition of the squaring function as a value, *fn* x = x * x, we did not have to declare x to be real. ML was able to figure out that * represents real multiplication from the fact that the second parameter of simpleMap is a *real list*.

The Function reduce:

- The reduce function takes a function F with two arguments and a list $[a_1, a_2, ..., a_n]$. The function F normally is assumed to compute some associative operation such as addition, that is, F(x, y) = x + y. The result of reduce on F and $[a_1, a_2, ..., a_n]$ is: $F(a_1, F(a_2, F(..., F(a_{n-1}, a_n)...)))$.
- Thinking of F as an associative binary infix operator, we have the simpler expression $a_1 F a_2 F ... F a_n$. For example, if F is the sum function, the $reduce(F, [a_1, a_2, ..., a_n])$ is $a_1 + a_2 + ... + a_n$, the sum of the elements on the list. ML has two functions *foldl* and *foldr* (fold from the left or right) that are similar in spirit to the function reduce.
- Usually the function F defines an associative operator, in which case it does not matter in what order we group the list elements. For instance:
  - The reduction of a list with F equal to the addition function produces the sum of the elements of the list.
  - The reduction by the product function produces the product of the elements of the list.
  - The reduction by the logical AND operator produces the value true if all the elements of a boolean list are true and produces false otherwise.
  - The reduction by the function max (larger of two elements) produces the largest element on the list.
- Definition:
  (1) *exception* EmptyList;
  (2) *fun* reduce(F, nil) = *raise* EmptyList
  (3) | reduce(F, [a]) = a
  (4) | reduce(F, x :: xs) = F(x, reduce(F, xs));
  (5) ⇒ *val reduce = fn: (‘a * ‘a ® ‘a) * ‘a list ® ‘a*
- Example: Computes $(\sum_{i=1}^{n} a_i^2) / n$
  (1) *fun* square(x: real) = x * x;
  (2) *fun* plus(x: real, y) = x + y;
  (3) *fun* sum(L) =
  (4) *let*
  (5) *val* n = real(length(L))
  (6) *in*
  (7) reduce(plus, simpleMap(square,L)) / n
  (8) *end*;

- We might expect that we could use the + operator in place of the function plus. But this doesn't work. The problem is that ML, like most languages, defines the usual arithmetic operators to be infix. That is, they appear between their operands. To allow an infix operator to be used as the name of a function, we precede it by the keyword *op*. Examples:
  - *op* + (2, 3); ⇒ *val it = 5: int*
  - … reduce(*op* +, simpleMap(square, L)) /n …

The Function Filter:

- The function filter takes a predicate P, that is, a function whose value is boolean, and a list $[a_1, a_2, ..., a_n]$. The result is the list of all those elements on the given list that satisfy the predicate P.
- Definition:
  (1) *fun* filter(P, nil) = nil
  (2) | filter(P, x :: xs) =
  (3) *if* P(x) *then* x :: filter(P, xs)
  (4) *else* filter(P, xs);
  (5) ⇒ *val filter = fn: ('a ® bool) * 'a list ® 'a list*
  (6) filter(*fn*(x) => x > 10, [1, 10 , 23, 5, 16]);
  (7) ⇒ *val it = [23, 16]: int list*

Curried Functions:

- Example: computes x^y
  - Uncurried version:
    (1) *fun* exponent1(x, 0) = 1.0
    (2) | exponent1(x, y) = x * exponent1(x, y – 1);
    (3) ⇒ *val exponent1 = fn: real * int ® real*
  - Curried version:
    (1) *fun* exponent2 x 0 = 1.0
    (2) | exponent2 x y = x * exponent2 x (y – 1);
    (3) ⇒ val *exponent2 = fn: real ® int ® real* (this is the same as: …fn: real →(int → real))
    (4) *val* g = exponent2 3.0;
    (5) ⇒ *val g = fn: int ® real*
      - Now g is a function that takes an integer y as argument and returns 3^y.

    (6) *val* v = g 4;
    (7) ⇒ *val v = 81.0: real*
    (8) *fun* cube x = exponent2 x 3;
      - Now cube is a function that takes a real x as argument and returns x^3.

Composition of Functions:

- Definition:
  (1) *fun* comp(F, G, x) = G(F(x));
  (2) ⇒ *val comp = fn: (('a ® 'b) * ('b ® 'c) * 'a) ® 'c*
- Example:
  (1) comp(*fn* x => x + 3, fn y => y * y + 2 * y, 10);
  (2) ⇒ *val it = 195: int*
  (3) (* other way *)
  (4) *fun* F x = x + 3;
  (5) *fun* G y = y * y + 2 * y;
  (6) *val* H = G o F ;
- Example : Look at p.176, figure 5.20
- Read p.156 – 181, good material about functions.

Folding Lists:

- ML provides the user a pair of functions called foldr and foldl. Both functions perform a variety of the fold operations, which takes a list $L = [a_1, a_2, ..., a_n]$ and treats each element a_i as if it were a function; call this function F_{a_i}. When we apply a folding operation to L, we construct the function that is the composition for all the functions $F_{a_1}, F_{a_2}, ..., F_{a_n}$, that is, $F_{a_1} \circ F_{a_2} \circ ... \circ F_{a_n}$.
- Example: Many operations on lists can be specified by folding, using an appropriate definition of the functions F_{a_i} and also choosing the right constant to which the composition of functions is applied. For instance, suppose $L = [a_1, a_2, ..., a_n]$ is a list of integers, and the function F_{a_i} is the function that multiplies its argument by a_i. Then the function $F_{a_1} \circ F_{a_2} \circ ... \circ F_{a_n}$ multiplies its argument by the product of the element of the list L, that is, $a_1 \cdot a_2 \cdot ... \cdot a_n$. If we apply this function to 1, we can compute the product of the elements of a list.
- The function foldr, with given list $[a_1, a_2, ..., a_n]$ and initial value b, computes $F_{a_1}(F_{a_2}(...(F_{a_n}(b))...))$ while foldl computes $F_{a_n}(F_{a_{n-1}}(...(F_{a_1}(b))...))$.
- Definition of *foldr*:
  (1) *fun* foldr F b nil = b
  (2) | foldr F b (x :: xs) = F(x, foldr F b xs);
  (3) ⇒ *val foldr = fn: ('a * 'b) → 'b → 'b → 'a list → 'b*
- Example: Multiply the elements of a list
  (1) *val* L = [2, 3, 4];
  (2) foldr (*op* *) 1 L;
  (3) ⇒ *val it = 24: int*
- Because foldr is defined in Curried form, we can partially instantiate foldr with a function F and an initial value b and get another function that takes a list L and "folds" L according to F and b.
- Example: We can write a function that takes the product of the elements on any integer list by:
  (1) *val* prod = foldr (*op* *) 1;
  (2) ⇒ *val prod = fn: int list → int*
  (3) prod [2, 3, 4];
  (4) ⇒ *val it = 24: int*

Chapter 6: Defining Your Own Types

Definitions:

- There are two ways to make type extensions:
  (1) *Type definitions* are shorthand or macros for previously defined type expressions.
  (2) *Datatype definitions* are rules for constructing new types with new values that are not the values of previously defined types.

New Names for Old Types:

- *type* <*identifier*> = <*type expression*>
- Example :
  - We might define the type signal to be a list of reals by :
    (1) *type* signal = *real list*;
    (2) ⇒ *type signal = real list*
    (3) *val* v = [1.0, 2.0]: signal;
    (4) ⇒ *val v = [1.0, 2.0]: signal*
    - The *type* signal is nothing more than an abbreviation.

Parameterized Type Definitions:

- *type (<list of type parameters>) <identifiers> = <type expression>*
- Example:
  (1) *type* (‘d, ‘r) mapping = (‘d * ‘r) *list*;
  (2) *val* word = [(“in”, 6), (“a”,1)]: (*string*, *int*) mapping;
  (3) (* this would cause *)
  (4) ⇒ *val words = [(“in”, 6), (“a”,1)]: (string, int) mapping*
  (5) (* instead of *)
  (6) ⇒ *val word = [(“in”, 6), (“a”,1)]: (string * int) list*

Datatypes:

- Definition:
  (1) datatype (<list of type parameters>) <identifier> =
  (2) *<first constructor expression>* |
  (3) *<second constructor expression>* |
  (4) *…..*
  (5) *<last constructor expression>*
- Since the type declarations is limited to definitions of “abbreviations”, it is of limited power. Often, we want to create types whose values are new structures. For instance, with the types learned so far we cannot express the notion of a tree. ML has a very powerful mechanism for defining new types called *datatypes*.
- Example 1:
  - Let us define the *datatype* with name fruit to consist of the three values Apple, Pear and Grape.
  - Identifier fruit is the type constructor for the datatype. The name Apple, Pear and Grape are the data constructors for the datatype fruit.
    (1) *datatype* fruit = Apple | Pear | Grape;
    (2) *fun* isApple(x) = (x = Apple);
    (3) ⇒ *val isApple = fn: fruit ® bool*
- Example 2:
    (1) We want to deal with “elements” that may be pairs or singles.
    (1) *datatype* (‘a, ‘b) element =
    (2) P *of* ‘a * ‘b |
    (3) S *of* ‘a;
    (4) ⇒ *datatype (‘a, ‘b) element = P of ‘a * ‘b | S of ‘a*
  - Remember to separate the elements of the list of type parameters with commas.
  - Line (2) tells us about the data constructor P, which takes as data a pair consisting of an ‘a value and a ‘b value and “wraps” them in the symbol P.
  - The example continues:
    (1) *fun* sumElList(nil) = 0
    (2) | sumElList(S(x) :: L) = sumElList(L)
    (3) | sumElList(P(x, y) :: L) = y + sumElList(L);
    (4) ⇒ *val sumElList = fn: (‘a, int) element list ® int*
    (5) sumElList [P(“in”, 6), S(“function”), P(“as”, 2)];
    (6) ⇒ *val it = 8: int*
- Example 3: Binary tree definition
  (1) *datatype* (‘label) btree =
  (2) Empty |
  (3) Node *of* ‘label * ‘label btree * ‘label btree;
  (4) ⇒ *datatype ‘a btree = Empty | Node of ‘a * ‘a btree * ‘a btree*
  (5) *val* tree = Node(“as”, Node(“a”, Empty, Empty), Node(“in”, Empty, Empty);

- Example 4: Mutually Recursive Datatypes

```
(1) datatype
(2)     (‘label) evenTree = Empty  |
(3)           Enode of ‘label * ‘label oddTree * ‘label oddTree
(4) and
(5)     (‘label) oddTree =
(6)           Onode of ‘label * ‘label evenTree * ‘label evenTree;
```

  - And now some examples on this datatype.

```
(1) val t1 = Onode(1, Empty, Empty);
(2) ⇒ val t1 = Onode(1, Empty, Empty): int oddTree
(3) val t3 = Enode(3, t1, t2);
(4) ⇒ val t3 = Enode(3, Onode(1, Empty, Empty), Onode(2, Empty, Empty)): int evenTree
```

- Example 5: General Rooted Trees

```
(1) datatype (‘label) tree =
(2)         Node of ‘label * ‘label tree list;
(3) Node(3, [
(4)         Node(4, nil),
(5)         Node(5, [Node(7, nil)]),
(6)         Node(6, nil)
(7) ]);
```

  - Computing the sums using higher order functions:

```
(1) fun sum(Node(a, L)) = a + foldr (op + ) 0 (map sum L);
```

  - The function used in folding, (*op* +), is addition, in prefix form. The second argument, which is the initial value for the sum, is 0. As a result, the second line of sum adds the elements on the list that map produced, giving us the sum of all the labels of all the subtrees of the root.

Chapter 7: More About ML Data Structures

Record Structures:

- Let us design a record structure that can represent information about students. The fields will be:
  1. An integer ID, the “student ID number”.
  2. A string name, the students name.
  3. A string list that we call courses, indicating the courses in which the student is currently enrolled.
- Implementation of the structure described above:

```
(1) val NormsRecord = {
(2)         ID = 123,
(3)         name = “Norm dePlum”,
(4)         courses = [“CS106X”, “E40”, “M43”]
(5) };
(6) ⇒ val NormsRecord = {
(7)         ID = 123,
(8)         courses = [“CS106X”, “E40”, “M43”],
(9)         name = “Norm dePlum”
(10)    }: {ID: int, courses: string list, name: string}
```

- Extracting field values: *#<label> (<record>)*

```
(1) #name(NormsRecord);
(2) ⇒ val it = “Norm dePlum”: string
```

- Tuples as a special case of record structures:
  - *(<value1>, …, <valuen>)* is really a shorthand for the record
  - *{1 = <value1>, …, n = <valuen>}*
  - Example: (3, “four”) is shorthand for record: {1 = 3, 2 = “four”}

- Patterns that match records:
  - Example: Finding the ID of a student with a given name.

```
(1) exception NotFound;
(2) fun getID(person, nil) = raise NotFound
(3)     getID(person, (x as {name = p, …}) :: xs) =
(4)         if p = person then
(5)             #ID(x: {name: string, ID: int, courses: string list})
(6)         else
(7)             getID(person, xs);
```

  - Remember that order of fields is irrelevant, so this type will match records such as NormsRecord.

Arrays:

- In this section we shall see the first ML construct that changes a value binging. Recall that normally, when we appear to be assigning a new value to a variable, in an "assignment" such as: *val* x = 1; we are really creating a binging for a new variable named x. This binging goes above any other binging for x in the current environment, say a binging in which x was bound to the value 2. However, other function calls may still have access to the old binging and that bindings still exists in the environment. If we were to change the binding for the "old" variable x, then functions that hat access to the old x would see the value of x change form 2 to 1.
- The guarantee that a value binging, once created, is available forever to functions that use it, is an important feature of ML. However, there are some situations where programs cannot be made adequately efficient unless we are allowed to change some value bindings. Thus, in this section we shall first motivate the need for arrays, with their attendant ability to change value bindings, and then show how ML allows us to create and use arrays.
- We need arrays to increase performance. This increase in efficiency is an important reason why ML provides arrays, even though the array violated the functional style of ML.
- The array operations are not available in the top – level – environment, so if we want to use arrays we may open the structure Array by:
  - *open* Array;
- The most important operations on arrays are the ones on page. 239.
- Example: p.240, figure 7.5

References:

- Standard ML has a mechanism for associating new values with names of any type, not just with array elements. This feature, called the reference, is the second way to violate the functional, side – effect – free style. We advocate its use only in situations where it simplifies the programming.
- For details see p.242.

The while – do Statement:

- *while* <*expression*> *do* <*expression*>
- Example:

```
(1) val i = ref 1;  (* for that we can assign new values to i *)
(2) ⇒ val i = ref 1: int ref
(3) while !i <= 10 do (
(4)     print(Int.toString(!i));
(5)     print(" ");
(6)     i := !i + 1;
(7) );
(8) ⇒ 1 2 3 4 5 6 7 8 9 10 val it = ( ): unit
```

Chapter 8: Encapsulation and the ML Module System

Definitions:

- One of the great themes of modern programming language design is facilitating the encapsulation of information, that is, the grouping of concepts such as types and functions on those types in a cluster that can be used only in limited ways. The limitation on use is not intended to make programming difficult, but rather:
  - To prevent data from being used in unexpected ways that result in hard – to – discover bugs.
  - To encourage reuse of code by allowing the definitions supporting a common idea to be packaged with a simple and precisely defined interface.
- In this chapter, we shall learn about the features of ML that support encapsulation: structures, signatures, and functors. Together, these concepts are called the *ML module system*.
- The *ML module system* has three major building blocks:
  1. *Structures* are collections of types, datatypes, functions, exceptions, and other elements that we wish to encapsulate. The definitions of these elements appear in the structure. We have actually met certain structures that appear in the ML standard basis, such as *Int* or *TextIO*.
  2. *Signatures* are collections of information describing the types and other specifications for some of the elements of a structure.
  3. Functors are operations that takes as arguments one or more elements such as structures and produce a structure that combines the functors arguments in some way.

Structures:

- *structure <identifier> = struct <elements of the structure> end*
- There are a number of different kinds of elements that may appear in a structure. The ones we shall use principally are:
  1. Function definitions
  2. Exceptions
  3. Constants
  4. Types
- Example:

```
(1) structure Mapping = struct
(2)       exception NotFound;
(3)       val create = nil;
(4)       fun lookup(d, nil) = …;
(5)       fun insert(d, r, nil) = ….;
(6) end;
```

(1) This causes the following type inference:

```
(1) ⇒ structure Mapping:
(2)       sig
(3)          exception NotFound
(4)          val create: 'a list
(5)          val insert: ''a * 'b * (''a * 'b) list ® (''a * 'b) list
(6)          val lookup: ''a * (''a * 'b) list ® 'b
(7)       end
```

- An identifier denoting a structure, such as identifier Mapping (see above) has a signature that serves as the "type" of the structure.
- Signature definitions:
  - *sig <specification> end*

- One can see an ML structure as a generalization for the idea of a class. Typically, a structure involves a single type, like the class, and the structure offers some functions (methods) applicable to that class. However, structures may define many different types or datatypes, and may provide functions that apply to any of these types or to different types altogether (analogous to "friend" classes in C++). Similarly, an ML functors generalizes the idea of a template in C++. Templates are parameterized classes that can be instantiated by designating particular types for the parameters of the template. Functors can do the same, but also can transform or combine the types or structures that are its arguments in ways that are limited only by the programmers imagination.

Functors:

- Having written a structure, we might wonder whether it is possible to create a large number of similar structures, perhaps each providing the same operations on a different type, in a manner that is easier than rewriting the structure many times. The functors provides this capability in a surprising way. A functors is actually a higher – level function that takes structures or certain other kinds of elements as arguments and returns a structure as a result.

ML Techniques for Hiding Information:

- In this section we shall discuss techniques for information hiding, that is, making all, or certain parts of, a package of definitions inaccessible to the user. Information hiding improves the effectiveness of encapsulation, since it limits the ways in which the elements of the package can be used.
- Abstract types:
  (1) abstype <datatype definition>
  (2) with
  (3) <declarations using the constructors>
  (4) *end*
  - An abstract type is essentially a datatype that hides its data constructors. We use the keyword *abstype* to define an abstract type with almost the same syntax as we use to define a datatype. However, for the abstract type we must follow it by the keyword *with* to introduced the only functions, variables, exceptions, and types that have access to the data constructors.
  - Example: p.290 for more information
    (1) *abstype* abtree = Empty |
    (2) Node *of string* * abtree * abtree
    (3) with
    (4) *fun* lookup(x, T) = …;
    (5) …
    (6) *end*;
- Local definitions:
  (1) local
  (2) <definitions>
  (3) in
  (4) <definitions>
  (5) *end*
  - *local* versus *let*: Do not confuse the *local – in – end* form with the *let – in – end* form. The former has a list of definitions between the *in* and *end*, while the latter has a list of expressions there. For example, a *val* – declaration may not follow the keyword *in* in a *let* – expression, while it may in a *local* – declaration.

Chapter 9: Summary of the ML Standard Basis

Definition:

- All the features of ML are grouped into structures, and these structures form the standard basis. Some of these features are present in the top – level environment and some need to be accessed through their structure, either by giving their long name (including the structure) or by opening the structure. We have seen almost all of the types, datatypes, exceptions, operators, and functions in the top – level environment.

The Infix Operators:

- Precedence levels in ML:

-2	orelse	Left assoziative
-1	andalso	Left assoziative
3	o, :=	Left assoziative
4	=, <>, <, >, <=, >=	Left assoziative
5	::, @	Right assoziative
6	+, -, ^	Left assoziative
7	*, /, div, mod	Left associative

- *infix <level> <identifier list>*
- If we wish an operator to be right – associative, we use the keyword *infixr* in place of *infix*.
- To remove the infix property from an identifier, use the keyword *nonfix*.
- Example:
  (1) *infix* 2 comb;
  (2) 5 comb 2 comb 4; (* (5 comb 2) comb 4 *)
  (3) ⇒ *val it = 210: int*
  (4) *nonfix* comb;
  (5) comb(comb(5, 2), 4);
  (6) ⇒ *val it = 210: int*

Functions in the Top – Level Environment:

- Only functions are presented which have not explained so far in this summary.
- *abs*(x): absolute value operator.
  - *abs* ~3; ⇒ *val it = 3: int*
- *size*(s): returns the integer that is the length of string s.
  - *size*("abc"); ⇒ *val it = 3: int*
- *substring*(s, i, j): Is the string of length j formed by taking the j position of s starting at position i.
  - *substring*("abcdefg", 2, 3); ⇒ *val it = "cde": string*
- *length*(L): returns the length of a list.
  - *length*(["a", "b", "c"]); ⇒ *val it 3: int*
- *rev*(L): returns the reverse of a list.
  - *rev* [1, 2, 3]; ⇒ *val it = [3, 2, 1]: int list*
- *toLower*(x): upper – case letters to lower – case letters.
  - *toLower*("#A"); ⇒ *val it = #"a": char*
- *toUpper*(x): lower – case letters to upper – case letters.
  - *toUpper*(#"a"); ⇒ *val it = #"A": char*
- *isSpace*(x): Predicate that checks if x is a space or new line.
  - *isSpace*(#" ");⇒ *val it = true: bool*

Primitive Datatypes:

- The datatypes *bool* has the definition:
  - *datatype bool = true | false;*
- The datatype *list* is defined by:
  - *datatype 'element list = nil | :: of 'element * 'element list;*