Chapter 1

Introduction to Functional Programming

This chapter presents basic ideas underlying functional programming, or programming in functional languages. All examples are written in Standard ML (abbreviated as SML henceforth), but it should be straightforward to translate them in any other functional language because all functional languages share the same design principle at their core. The results of running example programs are all produced in the interactive mode of SML of New Jersey.

Since this chapter is devoted to the discussion of important concepts in functional programming, the reader is referred to other sources for a thorough introduction to SML.

1.1 Functional programming paradigm

In the history of programming languages, there have emerged a few different programming paradigms. Each programming paradigm focuses on different aspects of programming, showing strength in some application areas but weakness in others. Object-oriented programming, for example, exploits the mechanism of extending object classes to express the relationship between different objects. Functional programming, as its name suggests, is unique in its emphasis on the role of functions as the basic component of programs. Combined with proper support for modular development, functional programming proves to be an excellent choice for developing large-scale software.

Functional programming is often compared with imperative programming to highlight its characteristic features. In imperative programming, the basic component is commands. Typically a program consists of a sequence of commands which yield the desired result when executed. Thus a program written in the imperative style is usually a description of "how to compute" — how to sort an array, how to add an element to a linked list, how to traverse a tree, and so on. In contrast, functional programming naturally encourages programmers to concentrate on "what to compute" because every program fragment must have a value associated with it.

To clarify the difference, let us consider an `if-then-else` conditional construct. In an imperative language (*e.g.*, C, Java, Pascal), the following code looks innocent with nothing suspicious; in fact, such code is often inevitable in imperative programming:

```
if (x == 1) then
  x = x + 1;
```

The above code executes the command to increment variable `x` when it is equal to `1`; if it is not equal to `1`, no command is executed — nothing wrong here. Now consider the following code written in a hypothetical functional language:

```
if (x = 1) then x + 1
```

With the `else` branch missing, the above code does not make sense: every program fragment must have a value associated with it, but the above code does not have a value when `x` is different from `1`. For this reason, it is mandatory to provide the `else` branch in every functional language.

As with other programming paradigms, the power of functional programming can be truly appreciated only with substantial experience with it. Unlike other programming paradigms, however, functional programming is built on a lot of fascinating theoretical ideas, independently of the issue of its advantages and disadvantages in software development. Thus functional programming should be the first step in your study of programming language theory!

1.2 Expressions and values

In SML, programs consists of *expressions* which range from simple constants to complex functions. Each expression can have a *value* associated with it, and the process of reducing an expression to a value is called *evaluation*. We say that an expression *evaluates* to a value when such a process terminates. Note that a value is a special kind of expression.

Example: integers

An integer constant `1` is an expression which is already a value. An integer expression `1 + 1` is not a value in itself, but evaluates to an integer value `2`. We can try to find the value associated with an expression by typing it and appending a semicolon at the SML prompt:

```
- 1 + 1;
val it = 2 : int
```

The second line above says that the result of evaluating the given expression is a `value` `2`. (We ignore the type annotation `: int` until Section 1.5.)

All arithmetic operators in SML have names familiar from other programming languages (*e.g.*, `+`, `-`, `*`, `div`, `mod`). The only notable exception is the unary operator for negation, which is not - but `~`. For example, `~1` is a negative integer, but `-1` does not evaluate to an integer.

Example: boolean values

Boolean constants in SML are `true` and `false`, and a conditional construct has the form `if` e `then` e_1 `else` e_2 where e, e_1, and e_2 are all expressions and e must evaluate to a boolean value. For example:

```
- if 1 = ~1 then 10 else ~10;
val it = ~10 : int
```

Here `1 = ~1` is an expression which compares two subexpressions `1` and `~1` for equality. (In fact, = is also an expression — a binary function taking two arguments.) Since the two subexpressions are not equal, `1 = ~1` evaluates to `false`, which in turn causes the whole expression to evaluate to `~10`.

Logical operators available in SML include `andalso`, `orelse`, and `not`. The two binary operators implement short-circuiting internally, but short-circuiting makes no difference in pure functional programming because the evaluation of an expression never produces side effects.

Exercise 1.1. Can you simplify `if` e `then true else false`?

1.3 Variables

A *variable* is a container for a value. As an expression, it evaluates to the very value it stores. We use the keyword `val` to initialize a variable. For example, a variable `x` is initialized with an integer expression `1 + 1` as follows:

```
- val x = 1 + 1;
val x = 2 : int
```

Note that we must provide an expression to be used in computing the initial value for x because there is no default value for any variable in SML. (In fact, the use of default values for variables does not even conform to the philosophy of functional programming for the reason explained below.) After initializing x, we may use it in other expressions:

```
- val y = x + x;
val y = 4 : int
```

We say that a variable is *bound* to a given value when it is initialized.

Unlike variables in imperative languages, a variable in SML is immutable in that its contents never change. In other words, a variable is bound to a single value for life. (This is the reason why it makes no sense to declare a variable without initializing it or by initializing it with a default value.) In this sense, "variable" is a misnomer because the contents of a variable is not really "variable." Despite their immutability, however, variables are useful in functional programming. Consider an example of a *local declaration* of SML in which zero or more *local variables* are declared before evaluating a final expression:

```
let
  val x = 1
  val y = x + x
in
  y + y
end
```

Here we declare two local variables x and y before evaluating `y + y`. Since y is added twice in the final expression, it saves computation time to declare y as a local variable instead of expanding both instances of y into `x + x`. The use of the local variable y also improves code readability.

While it may come as a surprise to you (especially if you still believe that executing a sequence of commands is the only way to complete a computation, as is typical in imperative programming), immutability of variables is in fact a feature that differentiates functional programming from imperative programming — without such a restriction on variables, there would be little difference between functional programming and imperative programming because commands (*e.g.*, for updating the contents of a variable) become available in both programming paradigms.

1.4 Functions

In the context of functional programming, a function can be thought of as equivalent to a mathematical function, *i.e.*, a black box mapping a given input to a unique output. Thus declaring a function in SML is indeed tantamount to defining a mathematical function, which in turn implies that the definition of a mathematical function is easily transcribed into an SML function. Interesting examples are given in Section 1.6 when we discuss recursion, and the present section focuses on the concept of function and its syntax in SML.

We use the keyword fun to declare a function. For example, we declare a function incr that returns a given integer incremented by one as follows:

```
- fun incr x = x + 1;
val incr = fn : int -> int
```

Here x is called a *formal argument/parameter* because it serves only as a placeholder for an *actual argument/parameter*. `x + 1` is called a *function body*. We can also create a nameless function with the keyword fn. The following code creates the same function as incr and stores it in a variable incr:

```
- val incr = fn x => x + 1;
val incr = fn : int -> int
```

The two declarations above are equivalent to each other.

A *function application* proceeds by substituting an actual argument for a formal argument in a function body and then evaluating the resultant expression. For example, a function application `incr 0` (applying function `incr` to `0`) evaluates to integer `1` via the following steps:

```
    incr 0
↦   (fn x => x + 1) 0
↦   0 + 1
↦   1
```

As an expression, a function is already a value. Intuitively a function is a black box whose internal working is hidden from the outside, and thus cannot be further reduced. As a result, a function body is evaluated only when a function application occurs. For example, a nameless function `fn x => 0 + 1` does *not* evaluate to `fn x => 1`; only when applied to an actual argument (which is ignored in this case) does it evaluate its body.

An important feature of functional programming is that functions are treated no differently from primitive values such as boolean values and integers. For example, a function can be stored in a variable (as shown above), passed as an actual argument to another function, and even returned as a return value of another function. Such values are often called *first-class objects* in programming language jargon because they are the most basic element comprising a program. Hence functions are first-class objects in functional languages. In fact, it turns out that a program in a functional language can be thought of as consisting entirely of functions and nothing else, since primitive values can also be encoded in terms of functions (which will be discussed in Chapter **??**).

Interesting examples exploiting the status of functions as first-class objects are found in Section 1.10. For now, we will content ourselves with an example illustrating that a function can be a return value of another function. Consider the following code which declares a function `add` taking two integers to calculate their sum:

```
- fun add x y = x + y;
val add = fn : int -> int -> int
```

Then what is a nameless function corresponding to `add`? A naive attempt does not even satisfy the syntax rules:

```
- val add = fn x y => x + y;
<some syntax error message>
```

The reason why the above attempt fails is that every function in SML can have only a single argument! The function `add` above (declared with the keyword `fun`) appears to have two arguments, but it is just a disguised form of a function that take an integer and returns another function:

```
- val add = fn x => (fn y => x + y);
val add = fn : int -> int -> int
```

That is, when applied to an argument `x`, it returns a new function `fn y => x + y` which returns `x + y` when applied to an argument `y`. Thus it is legitimate to apply `add` to a single integer to instantiate a new function as demonstrated below:

```
- val incr = add 1;
val incr = fn : int -> int
- incr 0;
val it = 1 : int
- incr 1;
val it = 2 : int
```

Now it should be clear how the evaluation of `add 1 1` proceeds:

```
     add 1 1
↦   (fn x => (fn y => x + y)) 1 1
↦   (fn y => 1 + y) 1
↦   1 + 1
↦   2
```

1.5 Types

Documentation is an integral part of good programming — without proper documentation, no code is easy to read unless it is self-explanatory. The importance of documentation (which is sometimes overemphasized in an introductory programming language course), however, often misleads students into thinking that long documentations are always better than concise ones. This is certainly untrue! For example, an overly long documentation on the function `add` can be more distracting than helpful to the reader:

```
(* Takes two arguments and returns their sum.
 * Both arguments must be integers.
 * If not, the result is unpredictable.
 * If their sum is too large, an overflow may occur.
 * ...
 *)
```

The problem here is that what is stated in the documentation cannot be formally verified by the compiler and we have to trust whoever wrote it. As an unintended consequence, any mistake in the documentation can leave the reader puzzled about the meaning of the code rather than helping her understand it. (For the simple case of `add`, it is not impossible to formally prove that the result is the sum of the two arguments, but then how can you express this property of `add` as part of the documentation?)

On the other hand, short documentations that can be formally verified by the compiler are often useless. For example, we could extend the syntax of SML so as to annotate each function with the number of its arguments:

```
argnum add 2          (* NOT valid SML syntax! *)
fun add x y = x + y;
```

Here `argnum add 2`, *as part of the code*, states that the function `add` has two arguments. The compiler can certainly verify that `add` has two arguments, but this property of `add` does not seem to be useful.

Types are a good compromise between expressiveness and simplicity: they convey *useful* information on the code (expressiveness) and can be *formally* verified by the compiler (simplicity). Informally a type is a collection of values of the same kind. For example, an expression that evaluates to an integer or a boolean constant has type `int` or `bool`, respectively. The function `add` has a *function type* `int -> (int -> int)` because given an integer of type `int`, it returns another function of type `int -> int` which takes an integer and returns another integer.[1] To exploit types as a means of documentation, we can explicitly annotate any formal argument with its type; the return type of a function and the type of any subexpression in its body can also be explicitly specified:

```
- fun add (x:int) (y:int) : int = (x + y) : int;
val add = fn : int -> int -> int
```

The SML compiler checks if types provided by programmers are valid; if not, it spits out an error message:

```
- fun add (x:int) (y:bool) = x + y;
stdIn:2.23-2.28 Error: <some error message>
```

[1] `->` is right associative and thus `int -> int -> int` is equal to `int -> (int -> int)`.

Perhaps add is too simple an example to exemplify the power of types, but there are countless examples in which the type of a function explains what it does, as we will see in subsequent chapters.

Another important use of types is as a debugging aid. In imperative programming, successful compilation seldom guarantees absence of errors. Usually we compile a program, run the executable code, and *then* start debugging by examining the result of the execution (be it a segmentation fault or a number different than expected). In functional programming with a rich type system, the story is different: we start debugging a program *before running the executable code* by examining the result of the compilation which is usually a bunch of *type errors*. Of course, neither in functional programming does successful compilation guarantee absence of errors, but programs that successfully compile run correctly *in most cases!* (You will encounter numerous such examples in doing assignments.)

Types are such a powerful tool in software engineering. We refer the reader to Chapter 1 of Pierce for an in-depth discussion of all the benefits of types in software engineering.

1.6 Recursion

Many problems in computer science require iterative procedures to reach a solution – adding integers from 1 to 100, sorting an array, searching for an entry in a B-tree, and so on. Because of its prominent role in programming, iterative computation is supported by built-in constructs in all programming languages. The C language, for example, provides constructs for directly implementing iterative computation such as the for loop construct:

```
for (i = 1; i <= 10; i++)
  sum += i;
```

The example above uses an index variable i which changes its value from 1 to 10. Surprisingly we cannot translate the above code in the pure fragment of SML (*i.e.*, without mutable references) because no variable in SML is allowed to change its value! Does this mean that SML is inherently inferior to C in its expressive power? The answer is "of course not:" SML supports *recursive computations* which are equally expressive to iterative computations.

Typically a recursive computation proceeds by decomposing a given problem into smaller problems, solving these smaller problems separately, and then combining their individual answers to produce a solution to the original problem. (Thus it is reminiscent of the divide-and-conquer algorithm which is in fact a particular instance of recursion.) It is important that these smaller problems are also solved recursively using the same method, perhaps by spawning another group of smaller problems to be solved recursively using the same method, and so on. Since such a sequence of decomposition cannot continue indefinitely (causing nontermination), a recursive computation starts to backtrack when it encounters a situation in which no such decomposition is necessary (*e.g.*, when the problem at hand is immediately solvable). Hence a typical form of recursion consists of *base cases* to specify the termination condition and *inductive cases* to specify how to decompose a problem into smaller ones.

As an example, here is a recursive function sum adding integers from 1 to a given argument n; note that we cannot use the keyword fn because we need to call the same function in its function body:

```
- fun sum n =
    if n = 1 then 1            (* base case *)
    else n + sum (n - 1);      (* inductive case *)
val sum = fn : int -> int
```

The evaluation of sum 10 proceeds as follows:

```
     sum 10
↦    if 10 = 1 then 1 else 10 + sum (10 - 1)
↦    if false then 1 else 10 + sum (10 - 1)
↦    10 + sum (10 - 1)
↦    10 + sum 9
↦*   10 + 9 + ··· 2 + sum 1
↦    10 + ··· 2 + (if 1 = 1 then 1 else 1 + sum (1 - 1))
↦    10 + ··· 2 + (if true then 1 else 1 + sum (1 - 1))
↦    10 + ··· 2 + 1
```

As with iterative computations, recursive computations may fall into infinite loops (*i.e.*, non-terminating computations), which occur if the base condition is never reached. For example, if the function `sum` is invoked with a negative integer as its argument, it goes into an infinite loop (usually ending up with a stack flow). In most cases, however, an infinite loop is due to a design flaw in the function body rather than an invocation with an inappropriate argument. Therefore it is a good practice to design a recursive function *before* writing code. A good way to design a recursive function is by formulating a mathematical equation. For example, a mathematical equation for `sum` would be given as follows:

$$\begin{array}{lcll} sum(1) & = & 1 & \\ sum(n) & = & 1 + sum(n-1) & \quad \textit{if } n > 1 \end{array}$$

Once such a mathematical equation is formulated, it should take little time to transcribe it into an SML function. (So think a lot before you write code!)

SML also supports mutually recursive functions. The keyword `and` is used to declare two or more mutually recursive functions. The following code declares two mutually recursive functions `even` and `odd` which determine if a given natural number is even or odd:

```
fun even n =
  if n = 0 then true
  else odd (n - 1)
and odd n =
  if n = 0 then false
  else even (n - 1)
```

Recursion may appear at first to be an awkward device not suited to iterative computations. This may be because iterative approaches, which are in fact intuitively easier to comprehend than recursive approaches, come first to mind (after being indoctrinated with mindless imperative programming!). Once you get used to functional programming, however, you will find that recursion is not an awkward device at all, but the most elegant device you can use in programming. (Note that *elegant* is synonymous with *easy-to-use* in the context of programming.) So the bottom line is: always think recursively!

1.7 Polymorphic types

In a software development process, we often write the same pattern of code repeatedly only with minor differences. As such, it is desirable to write a single common piece of code and then instantiate it with a different parameter whenever a copy of it is needed, thereby achieving a certain degree of code reuse. The utility of such a scheme is obvious. For example, if a bug is discovered in the program, we do not have to visit all the different places only to make the same change.

The question of how to realize code reuse in a safe way is quite subtle, however. For example, the C language provides macros to facilitate code reuse, but macros are notoriously prone to unexpected errors. Templates in the C++ language are safer because their parameters are types, but they are still nothing more than complex macros. In contrast, SML provides a code reuse mechanism, which not only is safe but also has a solid theoretical foundation, called *parametric polymorphism*.[2]

[2] A bit similar to, but not to be confused with the *polymorph* spell in the Warcraft series!

As a simple example, consider an identity function `id`:

```
val id = fn x = x;
```

Since we do not specify the type of `x`, it may accept an argument of any type. Semantically such an invocation of `id` poses no problem because its body does not need to know what type of value `x` is bound to. This observation suggests that a single declaration of `id` is conceptually equivalent to an infinite number of declarations all of which share the same function body:

```
val id_int = fn (x:int) = x;
val id_bool = fn (x:bool) = x;
val id_int→int = fn (x:int -> int) = x;
...
```

When `id` is applied to an argument of type A, the SML compiler automatically chooses the right declaration of `id` for type A.

The type system of SML compactly represents all these declarations of the same structure with a single declaration by exploiting a *type variable*:

```
- val id = fn (x:'a) => x;
val id = fn : 'a -> 'a
```

Here type variable `'a` may read *"any type α"*.[3] Then the type of `id` means that given an argument of any type α, it returns a value of type α. We may also explicitly specify type variables that may appear in a variable declaration by listing them before the variable:

```
- val 'a id = fn (x:'a) => x;
val id = fn : 'a -> 'a
```

We refer to types with no type variables as *monotypes* (or *monomorphic types*), and types with some type variables as *polytypes* (or *polymorphic types*). The type system of SML allows a type variable to be replaced by any monotype (but not a polytype). There are a few more restrictions on the use of type variables, which will be discussed in detail in Chapter **??**.

1.8 Datatypes

We have briefly discussed a few primitive types in SML. Here we give a comprehensive summary of basic types available in SML for future reference:

- `bool`: boolean values `true` and `false`.
- `int`: integers.
  E.g., `0`, `1`, `~1`, $\cdots$.
- `real`: floating pointer numbers.
  E.g., `0.0`, `1.0`, `~1.0`.
- `char`: characters.
  E.g., `#"a"`, `#"b"`, `#" "`.
- `string`: character strings.
  E.g., `"hello"`, `"newline\n"`, `"quote\""`, `"backslash\\"`.
- A `->` B: functions from type A to type B.

[3]Conventionally type variables are read as Greek letters (*e.g.*, `'a` as alpha, `'b` as beta, and so on).

- A `*` B: pairs of types A and B.
  If e_1 has type A and e_2 has type B, then `(`e_1`,` e_2`)` has type A `*` B.
  E.g., `(0, true) : int * bool`.
  A `*` B is called a *product type*.

- A_1 `*` A_2 `*` $\cdots$ `*` A_n: tuples of types A_1 through A_n.
  E.g., `(0, true, 1.0) : int * bool * real`.
  Tuple types are a generalized form of product types.

- `unit`: unit value `()`.
  The only value belonging to type `unit` is `()`. It is useful when declaring a function taking no interesting argument (*e.g.*, of type `unit -> int`).

These types suffice for most problems in which only numerical computations are involved. There are, however, a variety of problems for which *symbolic computations* are more suitable than numerical computations. As an example, consider the problem of classifying images into three categories: circles, squares, and triangles. We can assign integers `1`, `2`, `3` to these three shapes and use a function of type `image -> int` to classify images (where `image` is the type for images). A drawback of this approach is that it severely reduces the maintainability of the code: there is no direct connection between shapes and type `int`, and programmers should keep track of which variable of type `int` denotes shapes and which denotes integers.

A better approach is to represent each shape with a symbolic constant. In SML, we can use a *datatype* declaration to define a new type `shape` for three symbolic constants:

```
datatype shape = Circle | Square | Triangle
```

Each symbolic constant here has type `shape`. For example:

```
- Circle;
val it = Circle : shape
```

Note that datatypes are a special way of defining new types; hence they form only a subset of types. For example, `int -> int` is a (function) type but not a datatype whereas every datatype is also a type.

A datatype declaration is similar to an enumeration type in the C language, but with an important difference: symbolic constants are not compatible with integers and cannot be substituted for integers. Hence the new approach based on datatypes does not suffer from the same disadvantage of the previous approach. We refer to such symbolic constants as *data constructors*, or simply *constructors* in SML.

There is another feature of SML datatypes that sets themselves apart from C enumeration types: constructors may have arguments. For example, we can augment the above datatype declaration with arguments for constructors:

```
datatype shape =
  Circle of real
| Square of real
| Triangle of real * real * real
```

To create values of type `shape`, then, we have to provide appropriate arguments for constructors:

```
- Circle 1.0;
val it = Circle 1.0 : shape
- Square 1.0;
val it = Square 1.0 : shape
- Triangle (1.0, 1.0, 1.0);
val it = Triangle (1.0,1.0,1.0) : shape
```

Note that each constructor may be seen as a function from its argument type to type `shape`. For example, `Circle` is a function of type `real -> shape`:

```
- Circle;
val it = fn : real -> shape
```

Now we will discuss two important extensions of the datatype declaration mechanism. To motivate the first extension, consider a datatype `pair_bool` for pairing boolean values and another datatype `pair_int` for pairing integers:

```
datatype pair_bool = Pair of bool * bool
datatype pair_int = Pair of int * int
```

The two declarations are identical in structure except their argument types. We have previously seen how declarations of functions with the same structure but with different argument types can be coalesced into a single declaration by exploiting type variables. The situation here is no different: for any type A, a new datatype `pair_`A can be declared exactly in the same way:

```
datatype pair_A = Pair of A * A
```

The SML syntax for parameterizing a datatype declaration with type variables is to place type variables before the datatype name to indicate which type variables are local to the datatype declaration. Here are a couple of examples:

```
datatype 'a pair = Pair of 'a * 'a
datatype ('a, 'b) hetero = Hetero of 'a * 'b
```

The second extension of the datatype declaration mechanism allows a datatype to be used as the type of arguments for its own constructors. As with recursive functions, there must be at least one constructor (corresponding to base cases for recursive functions) that does not use the same datatype for its arguments — without such a constructor, it would be impossible to build values belonging to the datatype. Such datatypes are commonly referred to as *recursive datatypes*, which enable us to implement recursive data structures. As an example, consider a datatype `itree` for binary trees whose nodes store integers:

```
datatype itree =
  Leaf of int
| Node of itree * int * itree
```

The constructor `Leaf` represents a leaf node storing an integer of type `int`; the constructor `Node` represents an internal node which contains two subtrees as well as an integer. For example,

```
Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4)
```

represents the following binary tree:

```
      7
    /   \
   3     4
  / \
 1   2
```

Note that without using `Leaf`, it is impossible to build a value of type `itree`.

The two extensions of the datatype declaration mechanism may coexist to define *recursive datatypes with type variables*. For example, the datatype `itree` above can be generalized to a datatype `tree` for binary trees of values of *any* type:

```
datatype 'a tree =
  Leaf of 'a
| Node of 'a tree * 'a * 'a tree
```

Now arguments to constructors `Leaf` and `Node` automatically determine the actual type to be substituted for type variable `'a`:

```
- Node (Leaf ~1, 0, Leaf 1);
val it = Node (Leaf ~1,0,Leaf 1) : int tree
- Node (Leaf "L", "C", Leaf "R");
val it = Node (Leaf "L","C",Leaf "R") : string tree
```

Note that once type variable `'a` is instantiated to a specific type (say `int`), values of different types (say `string`) cannot be used. In other words, `tree` can be used only for homogeneous binary trees. The following expression does not compile because it does not determine a unique type for `'a`:

```
- Node (Leaf ~1, "C", Leaf 1);
stdIn:35.1-35.28 Error: <some error message>
```

Recursive data structures are common in both functional and imperative programming. In conjunction with type parameters, recursive datatypes in SML (and any other functional language) enable programmers to implement most of the common recursive data structures in a concise and elegant way. More importantly, a recursive data structure implemented in SML can have the compiler recognize some of its invariants, *i.e.*, certain conditions that must hold for all instances of the data structure. For example, given the definition of the datatype `tree` above, the SML compiler is aware of the invariant that every internal node must have two child nodes. This invariant would not be trivial to enforce in imperative programming. (Can you implement in the C or C++ language a datatype for binary trees in which every internal node has exactly two child nodes?)

We close this section with a brief introduction to the most frequently used datatype in functional programming: lists. They are provided as a built-in datatype `list` with two constructors: a unary constructor `nil` and a binary constructor `::`:

```
datatype 'a list = nil | :: of 'a * 'a list
```

`nil` denotes an empty list of any type (because it has no argument). `::`, called *cons*, is a right-associative infix operator and builds a list of type `'a list` by concatenating its *head* of type `'a` and *tail* of type `'a list`. For example, the following expression denotes a list consisting of `1`, `2`, and `3` in that order:

```
1 :: 2 :: 3 :: nil
```

Another way to create a list in SML is by enumerating its elements separated by commas `,` within brackets (`[` and `]`). For example, `[1, 2, 3]` is an abbreviation of the list given above. The two notations may also appear simultaneously within any list expression. For example, the following expressions are all equivalent:

```
[1, 2, 3]
1 :: [2, 3]
1 :: 2 :: [3]
1 :: 2 :: 3 :: []
```

1.9 Pattern matching

So far we have investigated how to create expressions of various types in SML. Equally important is the question of how to inspect those values that such expressions evaluate to. For simple types such as integers and tuples, the question is easy to answer: we only need to invoke operators already available in SML. For example, we use arithmetic and comparison operators on integers to test if a given integer belongs to a certain interval; for tuple types (including product types), we use the projection operator `#`n to retrieve the n-th element of a given tuple (*e.g*, `#2(1, 2, 3)` evaluates to `2`). In order to answer the question for datatypes, however, we need a means of testing which constructor has been applied in creating values of a given datatype. What makes this possible in SML is *pattern matching*.

As an example, let us write a function `length` that calculates the length of a given list. The way that `length` works is by simple recursion:

- If the argument is `nil`, return 0.
- If the argument has the form `<head> :: <tail>`, then invoke `length` on `<tail>` and return the result incremented by 1 (to account for `<head>`).

Thus `length` tries to match a given list with `nil` and `::` in either order. Moreover, when the list is matched with `::`, it also needs to retrieve arguments to `::` so as to invoke itself once more. The above definition of `length` is translated into the following code using pattern matching (which remotely resembles the `switch` construct in the C language):

```
fun length l =
  case l of
    nil => 0
  | head :: tail => 1 + length tail
```

Note that `nil` here is *not* a value; rather it is a *pattern* to be compared with `l` (or whatever follows the keyword `case`). Likewise `head :: tail` is a pattern which, when matched, binds `head` to the head of `l` and `tail` to the tail of `l`. If a pattern match occurs, the whole `case` expression reduces to the expression to the right of the corresponding =>. We call `nil` and `head :: tail` *constructor patterns* because of their use of datatype constructors.

Exercise 1.2. What is the type of `length`?

In the case of `length`, we do not need `head` in the second pattern. A *wildcard pattern* _ may be used if no binding is necessary:

```
fun length l =
  case l of
    nil => 0
  | _ :: tail => 1 + length tail
```

_ alone is also a valid pattern, which comes in handy when not every constructor needs to be considered. For example, a function testing if a given list is nil can be implemented as follows:

```
fun testNil l =
  case l of
    nil => true        (* if l is nil *)
  | _ => false        (* for ALL other cases *)
```

Regardless of constructors associated with the datatype `list`, the case analysis above is exhaustive because _ matches with any value.

Pattern matching in SML can be thought of as a generalized version of the `if-then-else` conditional construct. In fact, pattern matching is applicable to *any* type (not just datatypes with constructors). For example, `if` e `then` e_1 `else` e_2 may be expanded into:

```
case e of                case e of
  true => e1      or       true => e1
| false => e2            | _ => e2
```

It turns out that all variables in SML are also a special form of patterns, which in turn implies that any variable may be replaced by another pattern. We have seen two ways of introducing variables in SML: using the keyword `val` and in function declarations. Thus what immediately follows `val` can be not just a single variable but also any form of pattern; similarly formal arguments in a function declaration can be any form of pattern. As a first example, an easy way to retrieve all individual elements of a tuple is by exploiting a tuple pattern (instead of repeatedly using the projection operator $\#n$):

```
val (x, y, z) = <some tuple expression>
```

You can even use a constructor pattern `head :: tail` to match with a list:

```
val (head :: tail) = [1, 2, 3];
```

Here `head` becomes bound to `1` and `tail` to `[2, 3]`. Note, however, that the pattern is not exhaustive. For example, if `nil` is given as the right hand side, there is no way to match `head :: tail` with `nil`. (We will see in Section 1.11 how to handle such abnormal cases.) As a second example, we can rewrite the mutually recursive functions `even` and `odd` using pattern matching:

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

1.10 Higher-order functions

We have seen in Section 1.4 that every function in SML is a first-class object — it can be passed as an argument to another function and also returned as the result of a function application. Then a function that takes another function as an argument or returns another function as the result has a function type A `->` B in which A and B themselves may contain function types. We refer to such a function as a *higher-order* function. For example, functions of type `(int -> int) -> int` or `int -> (int -> int)` are all higher-order functions. We may also use type variables in higher-order function types. For example, `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)` is a higher-order function type. (Can you guess what a function of this type is supposed to do?)

Higher-order functions can significantly simplify many programming tasks when properly used. As an example, consider a function `map` of type `('a -> 'b) -> 'a list -> 'b list`.

Exercise 1.3. Make an educated guess of what a function of the above type is supposed to do. If you make a correct guess, it typifies the use of types as a means of documentation!

As you might have guessed, `map` takes a function f of type `'a -> 'b` and a list l of type `'a list`, and applies f to each element of l to create another list of type `'b list`. Here is an example of using `map` with the two functions `even` and `odd` defined above:

```
- map even [1, 2, 3, 4];
val it = [false,true,false,true] : bool list
- map odd [1, 2, 3, 4];
val it = [true,false,true,false] : bool list
```

The behavior of `map` can be formally written as follows:

$$\texttt{map}\ f\ \texttt{[}l_1\texttt{,}l_2\texttt{,}\cdots\texttt{,}l_n\texttt{]} \quad = \quad \texttt{[}f\ l_1\texttt{,}f\ l_2\texttt{,}\cdots\texttt{,}f\ l_n\texttt{]} \qquad (n \geq 0) \tag{1.1}$$

In order to implement `map`, we rewrite the equation (1.1) inductively by splitting it into two cases: a base case $n = 0$ and an inductive case $n > 0$. The base case is easy because the right side is an empty list:

$$\texttt{map}\ f\ \texttt{[]} \quad = \quad \texttt{[]} \tag{1.2}$$

The inductive case exploits the observation that $\texttt{[}f\ l_2\texttt{,}\cdots\texttt{,}f\ l_n\texttt{]}$ results from another application of `map`:[4]

$$\texttt{map}\ f\ \texttt{[}l_1\texttt{,}l_2\texttt{,}\cdots\texttt{,}l_n\texttt{]} \quad = \quad f\ l_1\ \texttt{::}\ \texttt{map}\ f\ \texttt{[}l_2\texttt{,}\cdots\texttt{,}l_n\texttt{]} \quad (n > 0) \tag{1.3}$$

The two equations (1.2) and (1.3) derive the following definition of `map`:

```
fun map f [] = []
  | map f (head :: tail) = f head :: map f tail
```

[4] `::` has a lower operator precedence than function applications, so we do not need parentheses.

1.11 Exceptions

Exceptions in SML provide a convenient mechanism for handling erroneous conditions that may arise during a computation. An exception is generated, or *raised*, either by the runtime system when an erroneous condition is encountered, or explicitly by programmers to transfer control to a different part of the program. For example, the runtime system raises an exception when a division by zero occurs, or when no pattern in a `case` expression matches with a given value; programmers may choose to raise an exception when an argument to a function does not satisfy the invariant of the function. An exception can be *caught* by an *exception handler*, which analyzes the exception to decide whether to raise another exception or to resume the computation. Thus not every exception results in aborting the computation.

An exception is a data constructor belonging to a special built-in datatype `exn` whose set of constructors can be freely extended by programmers. An exception declaration consists of a data constructor declaration preceded by the keyword `exception`. For example, we declare an exception `Error` with a string argument as follows:

```
exception Error of string
```

To raise `Error`, we use the keyword `raise`:

```
raise Error "Message for Error"
```

As exceptions are constructors for a special datatype `exn`, the syntax for exception handlers also uses pattern matching:

```
e handle
  <pattern1> => e1
|  ...
| <patternn> => en
```

If an exception is raised during the evaluation of expression e, <*pattern*$_1$> through <*pattern*$_n$> are tested in that order for a pattern match. If <*pattern*$_i$> matches with the exception, e_i becomes a new expression to be evaluated; if no pattern matches, the exception is propagated to the next exception handler, if any.

As a contrived example, consider the following code:

```
exception BadBoy of int;
exception BadGirl of int;
1 + (raise BadGirl ~1) handle
  BadBoy s => (s * s)
| BadGirl s => (s + s)
```

Upon attempting to evaluate the second operand of `+`, an exception `BadGirl` with argument `~1` is raised. Then the whole evaluation is aborted and the exception is propagated to the enclosing exception handler. As the second pattern matches with the exception being propagated, the expression `s + s` to the right of `=>` becomes a new expression to be evaluated. With `s` replaced by the argument `~1` to `BadGirl`, the whole expression evaluates to `~2`.

Exceptions are useful in a variety of situations. Even fully developed programs often exploit the exception mechanism to deal with exceptional cases. For example, when a time consuming computation is interrupted with a division by zero, the exception mechanism comes to rescue to save the partial result accumulated by the time of interruption. Here are a couple of other examples of exploiting exceptions in functional programming:

- You are designing a program in which a function `f` must never be called with a negative integer (which is an invariant of `f`). You raise an exception at the entry point of `f` if its argument is found to be a negative integer.

- All SML programs that you hand in for programming assignments should compile; otherwise you will receive no credit for your hard work. Now you have finished implementing a function `funEasy` but not another function `funHard`, both of which are part of the assignment. Instead of forfeiting points for `funEasy`, you submit the following code for `funHard`, which instantly makes the whole program compile:

  ```
  exception NotImplemented
  fun funHard _ = raise NotImplemented
  ```

  This trick works because `raise NotImplemented` has type `'a`.

1.12 Modules

Modular programming is a methodology for software development in which a large program is partitioned into independent smaller units. Each unit contains a set of related functions, types, *etc.* that can be readily reused in different programming tasks. SML provides a strong support for modular programming with *structures* and *signatures*. A structure, the unit of modular programming in SML, is a collection of declarations satisfying the specification given in a signature. SML also provides an innovative feature called *functors* which can be thought of as functions on structures. In this section, we restrict ourselves to the basic use of structures and signatures, which should be enough for all programming assignments in the early part of this course.

A structure is a collection of functions, types, exceptions, and other elements enclosed within a `struct` — `end` construct; a signature is a collection of specifications on these declarations enclosed within a `sig` — `end` construct. For example, the structure in the left conforms to the signature in the right:

```
struct                              sig
  type 'a set = 'a list               type 'a set
  val emptySet : 'a set = nil         val emptySet : 'a set
  fun singleton x = [x]               val singleton : 'a -> 'a set
  fun union s1 s2 = s1 @ s2           val union :
                                        'a set -> 'a set -> 'a set
end                                 end
```

The first line in the signature states that a type declaration of `'a set` must be given in a structure matching it; any type declaration resulting in a new type `'a set` is acceptable. In the example above, we use a type declaration using the keyword *type*, but a datatype declaration like

```
datatype 'a set = Set of 'a list
```

is also fine (if other elements in the same structure are redefined accordingly.) The second line in the signature states that a variable `emptySet` of type `'a set` must be defined in a structure matching it. The structure defines a variable `emptySet` of type `'a list`, which coincides with `'a set` under the definition of `'a set`. The third line in the signature states that a variable `singleton` of type `'a -> 'a set`, or equivalently a function `singleton` of type `'a -> 'a set`, must be defined in a structure matching it. Again `singleton` in the structure has type `'a -> 'a list` which is equal to `'a -> 'a set` under the definition of `'a set`. The case for `union` is similar.[5]

Like ordinary values, structures and signatures can be given names. We use the keywords `structure` and `signature` as illustrated below:

```
structure Set =             signature SET =
struct                      sig
  ...                         ...
end                         end
```

[5] `@` is an infix operator concatenating two lists.

Elements of the structure `Set` can then be accessed using the `.` notation familiar from the C language (*e.g.*, `Set.set`, `Set.emptySet`, $\cdots$).

Now how can we specify that the structure `Set` conforms to the signature `SET`? One way to do this is to impose a *transparent constraint* between `Set` and `SET` using a colon (`:`):

```
structure Set : SET = ...
```

The constraint by `:` says that `Set` conforms to `SET`; the program does not compile if `Set` fails to implement any specification in `SET`. Another way is to impose an *opaque constraint* between `Set` and `SET` using the symbol `:>`:

```
structure Set :> SET = ...
```

The constraint by `:>` says not only that `Set` conforms to `SET` but also that only those type declarations explicitly mentioned in `SET` are visible to the outside. To clarify their difference, consider the following code:

```
signature S = sig
  type t
end
```

```
structure Transparent : S =
struct
  type t = int
  val x = 1
end
structure Opaque :> S =
struct
  type t = int
  val x = 1
end
```

First note that both structures `Transparent` and `Opaque` conform to signature `S`. Since `S` does not declare variable `x`, there is no way to access `Transparent.x` and `Opaque.x`. The difference between `Transparent` and `Opaque` lies in the visibility of the definition of type `t`. In the case of `Transparent`, the definition of `t` as `int` is exported to the outside. Thus the following declaration is accepted because it is known that `Transparent.t` is indeed `int`:

```
- val y : Transparent.t = 1;
val y = 1 : Transparent.t
```

In the case of `Opaque`, however, the definition of `t` remains unknown to the outside, which causes the following declaration to be rejected:

```
- val z : Opaque.t = 1;
stdIn:3.5-3.21 Error: <some error message>
```

An opaque constraint in SML allows programmers to achieve data abstraction by hiding details of the implementation of a structure. In order to use structures given opaque constraints (*e.g.*, those included in the SML basis library or written by other programmers), therefore, you only need to read their signatures to see what values are exported. Often times you will see detailed documentation in signatures but no documentation in structures, for which there is a good reason.

Funtors