Introduction to Standard ML

Robert Harper[1]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

[1]With exercises by Kevin Mitchell, Edinburgh University, Edinburgh, UK.

Contents

Acknowledgements

Several of the examples were cribbed from Luca Cardelli's introduction to his dialect of ML [3], from Robin Milner's core language definition [5], from Dave MacQueen's modules paper [6], and from Abelson and Sussman's book [1]. Joachim Parrow, Don Sannella, and David Walker made many helpful suggestions.

Chapter 1

Introduction

These notes are an introduction to the Standard ML programming language. Here are some of the highlights of Standard ML:

- ML is a *functional* programming language. Functions are first-class data objects: they may be passed as arguments, returned as results, and stored in variables. The principal control mechanism in ML is recursive function application.

- ML is an *interactive* language. Every phrase read is analyzed, compiled, and executed, and the value of the phrase is reported, together with its type.

- ML is *strongly typed.* Every legal expression has a type which is determined automatically by the compiler. Strong typing guarantees that no program can incur a type error at run time, a common source of bugs.

- ML has a *polymorphic* type system. Each legal phrase has a uniquely-determined most general typing that determines the set of contexts in which that phrase may be legally used.

- ML supports *abstract types.* Abstract types are a useful mechanism for program modularization. New types together with a set of functions on objects of that type may be defined. The details of the implementation are hidden from the user of the type, achieving a degree of isolation that is crucial to program maintenance.

- ML is *statically scoped.* ML resolves identifier references at compile time, leading to more modular and more efficient programs.

- ML has a type-safe *exception* mechanism. Exceptions are a useful means of handling unusual or deviant conditions arising at run-time.

- ML has a *modules facility* to support the incremental construction of large programs. An ML program is constructed as an interdependent collection of *structures* which are glued together using *functors.* Separate compilation is supported through the ability to export and import functors.

Standard ML is the newest member of a family of languages tracing its origins to the ML language developed at Edinburgh by Mike Gordon, Robin Milner, and Chris Wadsworth in the mid-seventies [4]. Since then numerous dialects and implementations have arisen, both at Edinburgh and elsewhere. Standard ML is a synthesis of many of the ideas that were explored in the variant languages, notably Luca Cardelli's dialect [3], and in the functional language HOPE developed by Rod Burstall, Dave MacQueen, and Don Sannella [2]. The most recent addition to the language is the modules system developed by Dave MacQueen [6].

These notes are intended as an informal introduction to the language and its use, and should not be regarded as a definitive description of Standard ML. They have evolved over a number of years, and are in need of revision both to reflect changes in the language, and the experience gained with it since its inception. Comments and suggestions from readers are welcome.

The definition of Standard ML is available from MIT Press [7]. A less formal, but in many ways obsolete, account is available as an Edinburgh University technical report [5]. The reader is encouraged to consult the definition for precise details about the language.

Chapter 2

The Core Language

2.1 Interacting with ML

Most implementations of ML are interactive, with the basic form of interaction being the "read-eval-print" dialogue (familiar to LISP users) in which an expression is entered, ML analyzes, compiles, and executes it, and the result is printed on the terminal.[1]

Here is a sample interaction:

```
- 3+2;
> 5 : int
```

ML prompts with "`- `," and precedes its output with "`> `." The user entered the phrase "`3+2`". ML evaluated this expression and printed the value, "`5`", of the phrase, together with its type, "`int`".

Various sorts of errors can arise during an interaction with ML. Most of these fall into three categories: syntax errors, type errors, and run-time faults. You are probably familiar with syntax errors and run-time errors from your experience with other programming languages. Here is an example of what happens when you enter a syntactically incorrect phrase:

```
- let x=3 in x end;
Parse error: Was expecting "in" in ... let <?> x ...
```

[1]The details of the interaction with the ML top level vary from one implementation to another, but the overall "feel" is similar in all systems known to the author. These notes were prepared using the Edinburgh compiler, *circa* 1988.

Run-time errors (such as dividing by zero) are a form of *exception*, about which we shall have more to say below. For now, we simply illustrate the sort of output that you can expect from a run-time error:

```
- 3 div 0;
Failure: Div
```

The notion of a type error is somewhat more unusual. We shall have more to say about types and type errors later. For now, it suffices to remark that a type error arises from the improper use of a value, such as trying to add `3` to `true`:

```
- 3+true;
Type clash  in: 3+true
Looking for  a: int
I have found a: bool
```

One particularly irksome form of error that ML cannot diagnose is the infinite loop. If you suspect that your program is looping, then it is often possible to break the loop by typing the interrupt character (typically Control-C). ML will respond with a message indicating that the exception `interrupt` has been raised, and return to the top level. Some implementations have a debugging facility that may be helpful in diagnosing the problem.

Other forms of errors do arise, but they are relatively less common, and are often difficult to explain outside of context. If you do get an error message that you cannot understand, then try to find someone with more experience with ML to help you.

The details of the user interface vary from one implementation to another, particularly with regard to output format and error messages. The examples that follow are based on the Edinburgh ML compiler; you should have no difficulty interpreting the output and relating it to that of other compilers.

2.2 Basic expressions, values, and types

We begin our introduction to ML by introducing a set of basic types. In ML a type is a collection of values. For example, the integers form a type, as do the character strings and the booleans. Given any two types σ and τ, the set of ordered pairs of values, with the left component a value of type σ and the right a value of type τ, is a type. More significantly, the set of functions mapping one type to another form a type. In addition to these and other basic types, ML allows for user-defined types. We shall return to this point later.

Expressions in ML denote values in the same way that numerals denote numbers. The type of an expression is determined by a set of rules that guarantee that if the expression has a value, then the value of the expression is a value of the type assigned to the expression (got that?) For example, every numeral has type `int` since the value of a numeral is an integer. We shall illustrate the typing system of ML by example.

2.2.1 Unit

The type `unit` consists of a single value, written `()`, sometimes pronounced "unit" as well. This type is used whenever an expression has no interesting value, or when a function is to have no arguments.

2.2.2 Booleans

The type `bool` consists of the values `true` and `false`. The ordinary boolean negation is available as `not`; the boolean functions `andalso` and `orelse` are also provided as primitive.

The conditional expression, `if` e `then` e_1 `else` e_2, is also considered here because its first argument, e, must be a boolean. Note that the `else` clause is *not* optional! The reason is that this "if" is a conditional *expression*, rather than a conditional *command*, such as in Pascal. If the `else` clause were omitted, and the test were false, then the expression would have no value! Note too that both the `then` expression and the `else` expression must have the same type. The expression

```
if true then true else ()
```

is *type incorrect*, or *ill-typed*, since the type of the `then clause` is `bool`, whereas the type of the `else` clause is `unit`.

```
- not true;
> false : bool
- false andalso true;
```

```
> false : bool
- false orelse true;
> true : bool
- if false then false else true;
> true : bool
- if true then false else true;
> false : bool
```

2.2.3 Integers

The type `int` is the set of (positive and negative) integers. Integers are written in the usual way, except that negative integers are written with the tilde character "~" rather than a minus sign.

```
- 75;
> 75 : int
- ~24;
> ~24 : int
- (3+2) div 2;
> 2 : int
- (3+2) mod 2;
> 1 : int
```

The usual arithmetic operators, `+`, `-`, `*`, `div`, and `mod`, are available, with `div` and `mod` being integer division and remainder, respectively. The usual relational operators, `<`, `<=`, `>`, `>=`, `=`, and `<>`, are provided as well. They each take two expressions of type `int` and return a boolean according to whether or not the relation holds.

```
- 3<2;
> false : bool
- 3*2 >= 12 div 6;
> true : bool
- if 4*5 mod 3 = 1 then 17 else 51;
> 51 : int
```

Notice that the relational operators, when applied to two integers, evaluate to either true or false, and therefore have type `bool`.

2.2.4 Strings

The type `string` consists of the set of finite sequences of characters. Strings are written in the conventional fashion as characters between double quotes. The double quote itself is written "\"".

```
- "Fish knuckles";
> "Fish knuckles" : string
- "\"";
> """ : string
```

Special characters may also appear in strings, but we shall have no need of them. Consult the ML language definition [7] for the details of how to build such strings.

The function `size` returns the length, in characters, of a string, and the function ^ is an infix append function.[2]

```
- "Rhinocerous " ^ "Party";
> "Rhinocerous Party"
- size "Walrus whistle";
> 14 : int
```

2.2.5 Real Numbers

The type of floating point numbers is known in ML as `real`. Real numbers are written in more or less the usual fashion for programming languages: an integer followed by a decimal point followed by one or more digits, followed by the exponent marker, `E`, followed by another integer. The exponent part is optional, provided that the decimal point is present, and the decimal part is optional provided that the exponent part is present. These conventions are needed to distinguish integer constants from real constants (ML does not support any form of type inclusion, so an integer must be explicitly coerced to a real.)

```
- 3.14159;
> 3.14159 : real
```

[2]By *infix* we mean a function of two arguments that is written between its arguments, just as addition is normally written.

```
- 3E2;
> 300.0 : real
- 3.14159E2;
> 314.159 : real
```

The usual complement of basic functions on the reals are provided. The arithmetic functions `~`, `+`, `-`, and `*` may be applied to real numbers, though one may not mix and match: a real can only be added to a real, and not to an integer. The relational operators `=`, `<>`, `<`, and so on, are also defined for the reals in the usual way. Neither `div` nor `mod` are defined for the reals, but the function `/` denotes ordinary real-valued division. In addition there are functions such as `sin`, `sqrt`, and `exp` for the usual mathematical functions. The function `real` takes an integer to the corresponding real number, and `floor` truncates a real to the greatest integer less than it.

```
- 3.0+2.0;
> 5.0 : real
- (3.0+2.0) = real(3+2);
> true : bool
- floor(3.2);
> 3 : real
- floor(~3.2);
> ~4 : real
- cos(0.0);
> 1.0 : real
- cos(0);
Type clash  in: (cos 0)
Looking for  a: real
I have found a: int
```

This completes the set of *atomic* types in ML. We now move on to the *compound types*, those that are built up from other types.

2.2.6 Tuples

The type σ`*`τ, where σ and τ are types, is the type of ordered pairs whose first component has type σ and whose second component has type τ. Ordered pairs are written `(`e_1`,`e_2`)`, where e_1 and e_2 are expressions. Actually, there's no need to restrict ourselves to pairs; we can build ordered *n-tuples*, where $n \geq 2$, by writing n comma-separated expressions between parentheses.

```
- ( true, () );
> (true,()) : bool * unit
- ( 1, false, 17, "Blubber" );
> (1,false,17,"Blubber") : int * bool * int * string
- ( if 3=5 then "Yes" else "No", 14 mod 3 );
> ("No",2) : string * int
```

Equality between tuples is *component-wise* equality — two n-tuples are equal if each of their corresponding components are equal. It is a type error to try to compare two tuples of different types: it makes no sense to ask whether, say `(true,7)` is equal to `("abc",())`, since their corresponding components are of different types.

```
- ( 14 mod 3, not false ) = ( 1+1, true );
> true : bool
- ( "abc", (5*4) div 2 ) = ( "a"^"bc", 11);
> false : bool
- ( true, 7 ) = ( "abc", () );
Type clash in: (true,7)=("abc",())
Looking for a: bool*int
I have found a: string*unit
```

2.2.7 Lists

The type τ `list` consists of finite sequences, or *lists*, of values of type τ. For instance, the type `int list` consists of lists of integers, and the type `bool list list` consists of lists of lists of booleans. There are two notations for lists, the basic one and a convenient abbreviation. The first is based on the following characterization of lists: a τ `list` is either empty, or it consists of a value of type τ followed by a τ list. This characterization is reflected in the following notation for lists: the empty list is written `nil` and a non-empty list is written `e::l`, where `e` is an expression of some type τ and l is some τ `list`. The operator `::` is pronounced "cons", after the LISP list-forming function by that name.

If you think about this definition for a while, you'll see that every non-empty list can be written in this form:

e_1`::`e_2`::`$\cdots$`::`e_n`::nil`

where each e_i is an expression of some type τ, and $n \geq 1$. This accords with the intuitive meaning of a list of values of a given type. The role of `nil` is to serve as the terminator for a list — every list has the form illustrated above.

This method of defining a type is called a *recursive* type definition. Such definitions characteristically have one or more *base cases*, or starting points, and one or more *recursion cases*. For lists, the base case is the empty list, `nil`, and the recursion case is `cons`, which takes a list and some other value and yields another list. Recursively-defined types occupy a central position in functional programming because the organization of a functional program is determined by the structure of the data objects on which it computes.

Here are some examples of using `nil` and `::` to build lists:

```
- nil;
> [] : 'a list
- 3 :: 4 :: nil;
> [3,4] : int list
- ( 3 :: nil ) :: ( 4 :: 5 :: nil ) :: nil;
> [[3],[4,5]] : int list list
- ["This", "is", "it"];
> ["This","is","it"] : string list
```

Notice that ML prints lists in a compressed format as a comma-separated list of the elements of the list between square brackets. This format is convenient for input as well, and you may use it freely. But always keep in mind that it is an abbreviation — the `nil` and `::` format is the primary one.

The type of `nil` (see the example in Section 2) is peculiar because it involves a *type variable*, printed as `'a`, and pronounced “alpha”. The reason for this is that there is nothing about an empty list that makes it a list of integers or a list of booleans, or any type of list at all. It would be silly to require that there be a distinct constant denoting the empty list for each type of list, and so ML treats `nil` as a *polymorphic* object, one that can inhabit a variety of structurally-related types. The constant `nil` is considered to be an `int list` or a `bool list` or a `int list list`, according to the context. Note however that `nil` inhabits only list types. This is expressed by assigning the type `'a list` to `nil`, where `'a` is a variable ranging over the collection of types. An *instance* of a type involving a type variable (called a *polytype*, for short) is obtained by replacing all occurrences of a given type variable by some type (perhaps another polytype). For example, `'a list` has `int list` and `(int * 'b) list` as instances. A type that does not involve any type variables is called a *monotype*.

Equality on lists is item-by-item: two lists are equal if they have the same length, and corresponding components are equal. As with tuples, it makes no sense to compare two lists with different types of elements, and so any attempt to do so is considered a type error.

```
- [1,2,3] = 1::2::3::nil;
> true : bool
- [ [1], [2,4] ] = [ [2 div 2], [1+1, 9 div 3] ];
> false : bool
```

2.2.8 Records

The last compound type that we shall consider in this section is the *record type*. Records are quite similar to Pascal records and to C structures (and to similar features in other programming languages). A record consists of a finite set of labelled fields, each with a value of any type (as with tuples, different fields may have different types). Record values are written by giving a set of equations of the form $l = e$, where l is a label and e is an expression, enclosed in curly braces. The equation $l = e$ sets the value of the field labelled l to the value of e. The type of such a value is a set of pairs of the form $l : t$ where l is a label and τ is a type, also enclosed in curly braces. The order of the equations and typings is completely immaterial — components of a record are identified by their label, rather than their position. Equality is component-wise: two records are equal if their corresponding fields (determined by label) are equal.

```
- {name="Foo",used=true};
> {name="Foo", used=true} : {name:string, used:bool}
- {name="Foo",used=true} = {used=not false,name="Foo"};
> true : bool
- {name="Bar",used=true} = {name="Foo",used=true};
> false : bool
```

Tuples are special cases of records. The tuple type σ `*` τ is actually short-hand for the record type `{ 1 :` σ`, 2 :` τ `}` with two fields labeled

"1" and "2". Thus the expressions `(3,4)` and {`1=3,2=4`} have precisely the same meaning.

This completes our introduction to the basic expressions, values, and types in ML. It is important to note the regularity in the ways of forming values of the various types. For each type there are basic expression forms for denoting values of that type. For the atomic types, these expressions are the *constants* of that type. For example, the constants of type `int` are the numerals, and the constants of type `string` are the character strings, enclosed in double quotes. For the compound types, values are built using *value constructors*, or just *constructors*, whose job is to build a member of a compound type out of the component values. For example, the *pairing* constructor, written `( , )`, takes two values and builds a member of a tuple type. Similarly, `nil` and `::` are constructors that build members of the list type, as do the square brackets. The record syntax can also be viewed as a (syntactically elaborate) constructor for record types. This view of data as being built up from constants by constructors is one of the fundamental principles underlying ML and will play a crucial role in much of the development below.

There is one more very important type in ML, the function type. Before we get to the function type, it is convenient to take a detour through the declaration forms of ML, and some of the basic forms of expressions. With that under our belt, we can more easily discuss functions and their types.

2.3 Identifiers, bindings, and declarations

In this section we introduce *declarations*, the means of introducing identifiers in ML. All identifiers must be declared before they are used (the names of the built-in functions such as `+` and `size` are pre-declared by the compiler). Identifiers may be used in several different ways in ML, and so there is a declaration form for each such use. In this section we will concern ourselves with *value identifiers*, or *variables*. A variable is introduced by *binding* it to a value as follows:

```
- val x = 4*5;
> val x = 20 : int
- val s = "Abc" ^ "def";
> val s = "Abcdef" : string
```

```
- val pair = ( x, s );
> val pair = (20,"Abcdef") : int * string
```

The phrase `val x = 4*5` is called a *value binding*. To evaluate a value binding, ML evaluates the right-hand side of the equation and sets the value of the variable on the left-hand side to this value. In the above example, `x` is bound to `20`, an integer. Thereafter, the identifier `x` always stands for the integer `20`, as can be seen from the third line above: the value of `( x, s )` is obtained from the values of `x` and `s`.

Notice that the output from ML is slightly different than in our examples above in that it prints "`x = `" before the value. The reason for that is that whenever an identifier is declared, ML prints its definition (the form of the definition depends on the sort of identifier; for now, we have only variables, for which the definition is the value of the variable). An expression e typed at the top level (in response to ML's prompt) is evaluated, and the value of e is printed, along with its type. ML implicitly binds this value to the identifier `it` so that it can be conveniently referred to in the next top-level phrase.

It is important to emphasize the distinction between ML's notion of a variable and that of most other programming languages. ML's variables are more like the **const** declarations than **var** declarations of Pascal; in particular, binding is *not* assignment. When an identifier is declared by a value binding, a *new* identifier is "created" — it has nothing whatever to do with any previously declared identifier of the same name. Furthermore, once an identifier is bound to a value, there is no way to change that value: its value is whatever we have bound to it when it was declared. If you are unfamiliar with functional programming, then this will seem rather odd, at least until we discuss some sample programs and show how this is used.

Since identifiers may be rebound, some convention about which binding to use must be provided. Consider the following sequence of bindings.

```
- val x = 17;
> val x = 17 : int
- val y = x;
> val y = 17 : int
- val x = true;
> val x = true : bool
- val z = x;
```

```
    > val z = true : bool
```

The second binding for `x` hides the previous binding, and does not affect the value of `y`. Whenever an identifier is used in an expression, it refers to the closest textually enclosing value binding for that identifier. Thus the occurrence of `x` in the right-hand side of the value binding for `z` refers to the second binding of `x`, and hence has value `true`, not `17`. This rule is no different than that used in other block-structured languages, but it is worth emphasizing that it is the same.

Multiple identifiers may be bound simultaneously, using the keyword "`and`" as a separator:

```
    - val x = 17;
    > val x = 17 : int
    - val x = true and y = x;
    > val x = true : bool
      val y = 17 : int
```

Notice that `y` receives the value `17`, not `true`! Multiple value bindings joined by `and` are evaluated in parallel — first all of the right-hand sides are evaluated, then the resulting values are all bound to their corresponding left-hand sides.

In order to facilitate the following explanation, we need to introduce some terminology. We said that the role of a declaration is to define an identifier for use in a program. There are several ways in which an identifier can be used, one of which is as a variable. To declare an identifier for a particular use, one uses the binding form associated with that use. For instance, to declare an identifier as a variable, one uses a value binding (which binds a value to the variable and establishes its type). Other binding forms will be introduced later on. In general, the role of a declaration is to build an *environment*, which keeps track of the meaning of the identifiers that have been declared. For instance, after the value bindings above are processed, the environment records the fact that the value of `x` is `true` and that the value of `y` is `17`. Evaluation of expressions is performed with respect to this environment, so that the value of the expression `x` can be determined to be `true`.

Just as expressions can be combined to form other expressions by using functions like addition and pairing, so too can declarations be combined with other declarations. The result of a compound declaration is an environment determined from the environments produced by the component declarations. The first combining form for declarations is one that we've already seen: the semicolon for *sequential composition* of environments.[3]

```
- val x = 17 ; val x = true and y = x;
> val x = 17 : int
> val x = true : bool
  val y = 17 : int
```

When two declarations are combined with semicolon, ML first evaluates the left-hand declaration, producing an environment E, and then evaluates the right-hand declaration (with respect to E), producing environment E'. The second declaration may hide the identifiers declared in the first, as indicated above.

It is also useful to be able to have *local* declarations whose role is to assist in the construction of some other declarations. This is accomplished as follows:

```
- local
    val x = 10
  in
    val u = x*x + x*x
    val v = 2*x + (x div 5)
  end;
> val u = 200 : int
  val v = 22 : int
```

The binding for `x` is local to the bindings for `u` and `v`, in the sense that `x` is available during the evaluation of the bindings for `u` and `v`, but not thereafter. This is reflected in the result of the declaration: only `u` and `v` are declared.

It is also possible to localize a declaration to an expression using `let`:

```
- let
    val x = 10
  in
```

[3]The semicolon is syntactically optional: two sequential bindings are considered to be separated by a semicolon.

```
      x*x + 2*x + 1
    end;
  - 121 : int
```

The declaration of `x` is local to the expression occurring after the `in`, and is not visible from the outside. The body of the `let` is evaluated with respect to the environment built by the declaration occurring before the `in`. In this example, the declaration binds `x` to the value `10`. With respect to this environment, the value of `x*x+2*x+1` is `121`, and this is the value of the whole expression.

Exercise 2.3.1 *What is the result printed by the ML system in response to the following declarations? Assume that there are no initial bindings for* `x`*,* `y` *or* `z`*.*

1. `val x = 2 and y = x+1;`
2. `val x = 1; local val x = 2 in val y = x+1 end; val z = x+1;`
3. `let val x = 1 in let val x = 2 and y = x in x + y end end;`

2.4 Patterns

You may have noticed that there is no means of obtaining, say, the first component of a tuple, given only the expressions defined so far. Compound values are decomposed via pattern *matching*. Values of compound types are themselves compound, built up from their component values by the use of value constructors. It is natural to use this structure to guide the decomposition of compound values into their component parts.

Suppose that `x` has type `int*bool`. Then `x` must be some pair, with the left component an integer and the right component a boolean. We can obtain the value of the left and right components using the following generalization of a value binding.

```
  - val x = ( 17, true );
  > val x = (17,true) : int*bool
  - val ( left, right ) = x;
  > val left = 17 : int
    val right = true : bool
```

The left-hand side of the second value binding is a *pattern*, which is built up from variables and constants using value constructors. That is, a pattern is just an expression, possibly involving variables. The difference is that the variables in a pattern are not references to previously-bound variables, but rather variables that are about to be bound by pattern-matching. In the above example, `left` and `right` are two new value identifiers that become bound by the value binding. The pattern matching process proceeds by traversing the value of `x` in parallel with the pattern, matching corresponding components. A variable matches any value, and that value is bound to that identifier. Otherwise (*i.e.*, when the pattern is a constant) the pattern and the value must be identical. In the above example, since `x` is an ordered pair, the pattern match succeeds by assigning the left component of `x` to `left`, and the right component to `right`.

Notice that the simplest case of a pattern is a variable. This is the form of value binding that we introduced in the previous section.

It does not make sense to pattern match, say, an integer against an ordered pair, nor a list against a record. Any such attempt results in a type error at compile time. However, it is also possible for pattern matching to fail at run time:

```
- val x=(false,17);
> val x = (false,17) : bool*int
- val (false,w) = x;
> val w = 17 : int
- val (true,w) = x;
Failure: match
```

Notice that in the second and third value bindings, the pattern has a constant in the left component of the pair. Only a pair with this value as left component can match this pattern successfully. In the case of the second binding, `x` in fact has `false` as left component, and therefore the match succeeds, binding `17` to `w`. But in the third binding, the match fails because `true` does not match `false`. The message `Failure: match` indicates that a run-time matching failure has occurred.

Pattern matching may be performed against values of any of the types that we have introduced so far. For example, we can get at the components of a three element list as follows:

```
- val l = ["Lo", "and", "behold"];
> val l = ["Lo","and","behold"] : string list
- val [x1,x2,x3] = l;
> val x1 = "Lo" : string
  val x2 = "and" : string
  val x3 = "behold" : string
```

This works fine as long as we know the length of `l` in advance. But what if `l` can be any non-empty list? Clearly we cannot hope to write a single pattern to bind all of the components of `l`, but we can decompose `l` in accordance with the inductive definition of a list as follows:

```
- val l = ["Lo", "and", "behold"];
> val l = ["Lo","and","behold"] : string list
- val hd::tl = l;
> val hd = "Lo" : string
  val tl = ["and","behold"] : string list
```

Here `hd` is bound to the first element of the list `l` (called the *head* of `l`), and `tl` is bound to the list resulting from deleting the first element (called the *tail* of the list). The type of `hd` is `string` and the type of `tl` is `string list`. The reason is that `::` constructs lists out of a component (the left argument) and another list.

Exercise 2.4.1 *What would happen if we wrote* `val [hd,tl] = l;` *instead of the above. (Hint: expand the abbreviated notation into its true form, then match the result against* `l`*).*

Suppose that all we are interested in is the head of a list, and are not interested in its tail. Then it is inconvenient to have to make up a name for the tail, only to be ignored. In order to accommodate this "don't care" case, ML has a *wildcard* pattern that matches any value whatsoever, without creating a binding.

```
- val l = ["Lo", "and", "behold"];
> val l = ["Lo","and","behold"] : string list
- val hd::_ = l;
> val hd = "Lo" : string
```

Pattern matching may also be performed against records, and, as you may have guessed, it is done on the basis of labelled fields. An example will illustrate record pattern matching:

```
- val r = { name="Foo", used=true };
> val r = {name="Foo",used=true} : {name:string,used:bool}
- val { used=u, name=n } = r;
> val n = "Foo" : string
  val u = true : bool
```

It is sometimes convenient to be able to match against a partial record pattern. This can be done using the *record wildcard*, as the following example illustrates:

```
- val { used=u, ... } = r ;
> val u = true : bool
```

There is an important restriction on the use of record wildcards: it must be possible to determine at compile time the type of the entire record pattern (*i.e.*, all the fields and their types must be inferrable from the context of the match).

Since single-field selection is such a common operation, ML provides a short-hand notation for it: the `name` field of `r` may be designated by the application `#name r`. Actually, `#name` is bound to the function `fn {name=n,...} => n`, which selects the `name` field from a record, and thus it must be possible to determine from context the entire record type whenever a selection function is used. In particular, `fn x => #name x` will be rejected since the full record type of x is not fixed by the context of occurrence. You will recall that n-tuples are special forms of records whose labels are natural numbers i such that $1 \leq i \leq n$. The ith component of a tuple may therefore be selected using the function `#`i.

Patterns need not be flat, in the following sense:

```
- val x = ( ( "foo", true ), 17 ) ;
> val x = (("foo",true),17) : (string*bool)*int
- val ((ll,lr),r) = x ;
> val ll = "foo" : string
  val lr = true : bool
  val r = 17 : int
```

Sometimes it is desirable to bind "intermediate" pattern variables. For instance, we may want to bind the pair `(ll,lr)` to an identifier `l` so that we can refer to it easily. This is accomplished by using a *layered pattern*. A layered pattern is built by attaching a pattern to a variable within another pattern as follows:

```
- val x = ( ( "foo", true ), 17 );
> val x = (("foo",true),17) : (string*bool)*int
- val ( l as (ll,lr), r ) = x;
> val l = ("foo",true) : string*bool
  val ll = "foo" : string
  val lr = true : bool
  val r = 17 : int
```

Pattern matching proceeds as before, binding `l` and `r` to the left and right components of `x`, but in addition the binding of `l` is further matched against the pattern `(ll,lr)`, binding `ll` and `lr` to the left and right components of `l`. The results are printed as usual.

Before you get too carried away with pattern matching, you should realize that there is one significant limitation: patterns must be *linear*: a given pattern variable may occur only once in a pattern. This precludes the possibility of writing a pattern `(x,x)` which matches only symmetric pairs, those for which the left and right components have the same value. This restriction causes no difficulties in practice, but it is worth pointing out that there are limitations.

Exercise 2.4.2 *Bind the variable* `x` *to the value* `0` *by constructing patterns to match against the following expressions.*
For example, given the expression `(true,"hello",0)`*, the required pattern is* `(_,_,x)`.

1. `{ a=1, b=0, c=true }`

2. `[ ~2, ~1, 0, 1, 2 ]`

3. `[ (1,2), (0,1) ]`

2.5 Defining functions

So far we have been using some of the pre-defined functions of ML, such as the arithmetic functions and the relational operations. In this section we introduce *function bindings*, the means by which functions are defined in ML.

We begin with some general points about functions in ML. Functions are used by *applying* them to an argument. Syntactically, this is indicated by writing two expressions next to one another, as in `size "abc"` to invoke the function `size` with argument `"abc"`. All functions take a single argument; multiple arguments are passed by using tuples. So if, for example, there were a function `append` which takes two lists as arguments, and returns a list, then an application of `append` would have the form `append(l1,l2)`: it has single argument which is an ordered pair `(l1,l2)`. There is a special syntax for some functions (usually just the built-in's) that take a pair as argument, called *infix application*, in which the function is placed between the two arguments. For example, the expression `e1 + e2` really means "apply the function `+` to the pair `(e1,e2)`. It is possible for user-defined functions to be infix, but we shall not go into that here.

Function application can take a syntactically more complex form in ML than in many common programming languages. The reason is that in most of the common languages, functions can be designated only by an identifer, and so function application always has the form $f(e_1, \ldots, e_n)$, where f is an identifier. ML has no such restriction. Functions are perfectly good values, and so may be designated by arbitrarily complex expressions. Therefore the general form of an application is $e\ e'$, which is evaluated by first evaluating e, obtaining some function f, then evaluating e', obtaining some value v, and applying f to v. In the simple case that e is an identifier, such as `size`, then the evaluation of e is quite simple — simply retrieve the value of `size`, which had better be a function. But in general, e can be quite complex and require any amount of computation before returning a function as value. Notice that this rule for evaluation of function application uses the *call-by-value* parameter passing mechanism since the argument to a function is evaluated before the function is applied.

How can we guarantee that in an application $e\ e'$, e will in fact evaluate to a function and not, say, a boolean? The answer, of course, is in the type of e. Functions are values, and all values in ML are divided up into types. A *function type* is a compound type that has functions as members.

A function type has the form σ`->`τ, pronounced "σ to τ," where σ and τ are types. An expression of this type has as value a function that whenever it is applied to a value of type σ, returns a value of type τ, provided that it terminates (unfortunately, there is no practical means of ensuring that all functions terminate for all arguments). The type σ is called the *domain type* of the function, and τ is called its *range type*. An application $e\ e'$ is legal only if e has type σ`->`τ and e' has type σ, that is, only if the type of the argument matches the domain type of the function. The type of the whole expression is then τ, which follows from the definition of the type σ`->`τ.

For example,

```
- size;
  size = fn : string -> int
- not;
  not = fn : bool -> bool
- not 3;
Type clash  in: not 3
Looking  for a: bool
I have found a: int
```

The type of `size` indicates that it takes a string as argument and returns an integer, just as we might expect. Similarly, `not` is a function that takes a boolean and returns a boolean. Functions have no visible structure, and so print as "`fn`". The application of `not` to `3` fails because the domain type of `not` is `bool`, whereas the type of `3` is `int`.

Since functions are values, we can bind them to identifiers using the value binding mechanism introduced in the last section. For example,

```
- val len = size;
> val len = fn : string -> int
- len "abc";
> 3 : int
```

The identifier `size` is bound to some (internally-defined) function with type `string->int`. The value binding above retrieves the value of `size`, some function, and binds it to the identifier `len`. The application `len "abc"` is processed by evaluating `len` to obtain some function, evaluating `"abc"` to obtain a string (itself), and applying that function to that string. The result is `3` because the function bound to `size` in ML returns the length of a string in characters.

Functions are complex objects, but they are not built up from other objects in the same way that ordered pairs are built from their components. Therefore their structure is not available to the programmer, and pattern matching may not be performed on functions. Furthermore, it is not possible to test the equality of two functions (due to a strong theoretical result which says that this cannot be done, even in principle). Of all the types we have introduced so far, every one except the function type has an equality defined on values of that type. Any type for which we may test equality of values of that type is said to *admit equality*. No function type admits equality, and every atomic type admits equality. What about the other compound types? Recall that equality of ordered pairs is defined "component-wise": two ordered pairs are equal iff their left components are equal and their right components are equal. Thus the type $\sigma * \tau$ admits equality iff both σ and τ admit equality. The same pattern of reasoning is used to determine whether an arbitrary type admits equality. The rough-and-ready rule is that if the values of a type involve functions, then it probably doesn't admit equality (this rule can be deceptive, so once you get more familiar with ML, you are encouraged to look at the official definition in the ML report [7]).

With these preliminaries out of the way, we can now go on to consider user-defined functions. The syntax is quite similar to that used in other languages. Here are some examples.

```
- fun twice x = 2*x;
> val twice = fn : int->int
- twice 4;
> 8 : int
- fun fact x = if x=0 then 1 else x*fact(x-1);
> val fact = fn : int->int
- fact 5;
> 120 : int
- fun plus(x,y):int=x+y;
> val plus = fn : int*int->int
- plus(4,5);
> 9 : int
```

Functions are defined using *function bindings* that are introduced by the keyword `fun`. The function name is followed by its parameter, which is a pattern. In the first two examples the parameter is a simple pattern, consisting of a single identifier; in the third example, the pattern is a pair whose left component is `x` and right component is `y`. When a user-defined function is applied, the value of the argument is matched against the parameter of the function in exactly the same way as for value bindings, and the body of the function is evaluated in the resulting environment. For example, in the case of `twice`, the argument (which must be an integer, since the type of `twice` is `int->int`) is bound to `x` and the body of `twice`, `2*x` is evaluated, yielding the value `8`. For `plus` the pattern matching is slightly more complex since the argument is a pair, but it is no different from the value bindings of the previous section: the value of the argument is matched against the pattern `(x,y)`, obtaining bindings for `x` and `y`. The body is then evaluated in this environment, and the result is determined by the same evaluation rules. The "`:int`" in the definition of `plus` is called a *type constraint*; its purpose here is to disambiguate between integer addition and real addition. We shall have more to say about this, and related issues, later on.

Exercise 2.5.1 *Define the functions* `circumference` *and* `area` *to compute these properties of a circle given its radius.*

Exercise 2.5.2 *Define a function to compute the absolute value of a real number.*

The definition of the function `fact` illustrates an important point about function definitions in ML: functions defined by `fun` are *recursive*, in the sense that the occurrence of `fact` in the right-hand side of the definition of `fact` refers to the very function being defined (as opposed to some other binding for `fact` which may happen to be in the environment). Thus `fact` "calls itself" in the process of evaluating its body. Notice that on each recursive call, the argument gets smaller (provided that it was greater than zero to begin with), and therefore `fact` will eventually terminate. Non-terminating definitions are certainly possible, and are the bane of the ML novice. For a trivial example, consider the function

```
- fun f(x)=f(x);
> val f = fn: 'a->'b
```

Any call to `f` will loop forever, calling itself over and over.

Exercise 2.5.3 *An alternative syntax for conditional statements might be defined by*

```
        fun new_if(A,B,C) = if A then B else C
```

Explain what goes wrong if the definition for `fact` *is altered to use this new definition.*

Now we can go on to define some interesting functions and illustrate how real programs are written in ML. Recursion is the key to functional programming, so if you're not very comfortable with it, you're advised to go slowly and practice evaluating recursive functions like `fact` by hand.

So far we have defined functions with patterns consisting only of a single variable, or an ordered pair of variables. Consider what happens if we attempt to define a function on lists, say `is_nil` which determines whether or not its argument is the empty list. The list types have two value constructors: `nil` and `::`. A function defined on lists must work regardless of whether the list is empty or not, and so must be defined by cases, one case for `nil` and one case for `::`. Here is the definition of `is_nil`:

```
     - fun is_nil( nil ) = true
         | is_nil( _::_ ) = false ;
     > is_nil = fn : 'a list -> bool
     - is_nil nil ;
     > true : bool
     - is_nil [2,3] ;
     > false : bool
```

The definition of `is_nil` reflects the structure of lists: it is defined by cases, one for `nil` and one for `h::t`, separated from one another by a vertical bar.

In general if a function is defined on a type with more than one value constructor, then that function must have one case for each constructor. This guarantees that the function can accept an arbitrary value of that type without failure. Functions defined in this way are called *clausal function definitions* because they contain one clause for each form of value of the argument type.

Of course, clausal definitions are appropriate for recursively-defined functions as well. Suppose that we wish to define a function `append` that, given two lists, returns the list obtained by tacking the second onto the end of the first. Here is a definition of such a function:

```
- fun append(nil,l) = l
    | append(hd::tl,l) = hd :: append(tl,l);
> val append = fn : ( 'a list * 'a list ) -> 'a list
```

There are two cases to consider, one for the empty list and one for a non-empty list, in accordance with the inductive structure of lists. It is trivial to append a list `l` to the empty list: the result is just `l`. For non-empty lists, we can append `l` to `hd::tl` by cons'ing `hd` onto the result of appending `l` to `tl`.

Exercise 2.5.4 *Evaluate the expression* `append([1,2],[3])` *by hand to convince yourself that this definition of* `append` *is correct.*

Exercise 2.5.5 *What function does the following definition compute?*

```
fun r [] = [] | r(h::t) = append(r(t),[h])
```

The type of `append` is a polytype; that is, it is a type that involves the type variable `'a`. The reason is that `append` obviously works no matter what the type of the elements of the list are — the type variable `'a` stands for the type of the elements of the list, and the type of `append` ensures that both lists to be appended have the same type of elements (which is the type of the elements of the resulting list). This is an example of a *polymorphic function*; it can be applied to a variety of lists, each with a different element type. Here are some examples of the use of `append`:

```
- append([],[1,2,3]);
> [1,2,3] : int list
- append([1,2,3],[4,5,6]);
> [1,2,3,4,5,6] : int list
- append(["Bowl","of"],["soup"]);
> ["Bowl", "of", "soup"] : string list
```

Notice that we used `append` for objects of type `int list` and of type `string list`.

In general ML assigns the most general type that it can to an expression. By "most general", we mean that the type reflects only the commitments that are made by the internal structure of the expression. For example, in the definition of the function `append`, the first argument is used as the target of a pattern match against `nil` and `::`, forcing it to be of some list type. The type of the second argument must be a list of the same type since it is potentially cons'd with an element of the first list. These two constraints imply that the result is a list of the same type as the two arguments, and hence `append` has type `('a list * 'a list) -> 'a list`.

Returning to the example above of a function `f(x)` defined to be `f(x)`, we see that the type is `'a->'b` because, aside from being a function, the body of `f` makes no commitment to the type of `x`, and hence it is assigned the type 'a, standing for any type at all. The result type is similarly uncommitted, and so is taken to be `'b`, an arbitrary type. You should convince yourself that no type error can arise from any use of `f`, even though it has the very general type `'a->'b`.

Function bindings are just another form of declaration, analogous to the value bindings of the previous section (in fact, function bindings are just a special form of value binding). Thus we now have two methods for building declarations: value bindings and function bindings. This implies that a function may be defined anywhere that a value may be declared; in particular, local function definitions are possible. Here is the definition of an efficient list reversal function:

```
- fun reverse l =
    let fun rev(nil,y) = y
          | rev(hd::tl,y) = rev(tl,hd::y)
    in
        rev(l,nil)
    end;
> val reverse = fn : 'a list -> 'a list
```

The function `rev` is a local function binding that may be used only within the `let`. Notice that `rev` is defined by recursion on its first argument, and `reverse` simply calls `rev`, and hence does not need to decompose its argument `l`.

Functions are not restricted to using parameters and local variables — they may freely refer to variables that are available when the function is defined. Consider the following definition:

```
- fun pairwith(x,l) =
    let fun p y = (x,y)
```

```
        in  map p l
        end;
    > val pairwith = fn : 'a * 'b list -> ('a*'b) list
    - val l=[1,2,3];
    > val l = [1,2,3] : int list
    - pairwith("a",l);
    > [("a",1),("a",2),("a",3)] : ( string * int ) list
```

The local function `p` has a non-local reference to the identifier `x`, the parameter of the function `pairwith`. The same rule applies here as with other non-local references: the nearest enclosing binding is used. This is exactly the same rule that is used in other block structured languages such as Pascal (but differs from the one used in most implementations of LISP).

Exercise 2.5.6 *A "perfect number" is one that is equal to the sum of all its factors (including 1 but not including itself). For example, 6 is a perfect number because* $6 = 3 + 2 + 1$. *Define the predicate* `isperfect` *to test for perfect numbers.*

It was emphasized above that in ML functions are values; they have the same rights and privileges as any other value. In particular, this means that functions may be passed as arguments to other functions, and applications may evaluate to functions. Functions that use functions in either of these ways are called *higher order* functions. The origin of this terminology is somewhat obscure, but the idea is essentially that functions are often taken to be more complex data items than, say, integers (which are called "first order" objects). The distinction is not absolute, and we shall not have need to make much of it, though you should be aware of roughly what is meant by the term.

First consider the case of a function returning a function as result. Suppose that `f` is such a function. What must its type look like? Let's suppose that it takes a single argument of type τ. Then if it is to return a function as result, say a function of type σ`->`ρ, then the type of `f` must be τ`->(`σ`->`ρ`)` This reflects the fact that `f` takes an object of type τ, and returns a function whose type is σ`->`ρ. The result of any such application of `f` may itself be applied to a value of type σ, resulting in a value of type ρ. Such a successive application is written `f(e1)(e2)`, or just `f e1 e2`; this is *not* the same as `f(e1,e2)`! Remember that `(e1,e2)` is a *single* object, consisting of an ordered pair of values. Writing `f e1 e2` means "apply `f` to `e1`, obtaining a function, then apply that function to `e2`". This is why we went to such trouble above to explain function application in terms of obtaining a function value and applying it to the value of the argument: functions can be denoted by expressions other than identifiers.

Here are some examples to help clarify this:

```
- fun times (x:int) (y:int) = x*y;
> val times = fn : int->(int->int)
- val twice = times 2;
> val twice = fn : int -> int
- twice 4;
> 8 : int
- times 3 4;
> 12 : int
```

The function `times` is defined to be a function that, when given an integer, returns a function which, when given an integer returns an integer.[4] The identifer `twice` is bound to `times 2`. Since `2` is an object of type `int`, the result of applying `times` to `2` is an object of type `int->int`, as can be seen by inspecting the type of `times`. Since `twice` is a function, it may be applied to an argument to obtain a value, in this case `twice 4` returns `8` (of course!). Finally `times` is successively applied to `3`, then the result is applied to `4`, yielding `12`. This last application might have been parenthesized to `(times 3) 4` for clarity.

It is also possible for functions to take other functions as arguments. Such functions are often called *functionals* or *operators*, but, once again, we shall not concern ourselves terribly much with this terminology. The classical example of such a function is the `map` function which works as follows: `map` takes a function and a list as arguments, and returns the list resulting from applying the function to each element of the list in turn. Obviously the function must have domain type the same as the type of the elements of the list, but its range type is arbitrary. Here is a definition for `map`:

```
- fun map f nil = nil
    | map f (hd::tl) = f(hd) :: map f tl ;
> val map = fn : ('a->'b) -> ('a list) -> ('b list)
```

[4]The need for "`:int`" on `x` and `y` will be explained in Section 6 below.

Notice how the type of `map` reflects the correlation between the type of the list elements and the domain type of the function, and between the range type of the function and the result type.

Here are some examples of using `map`:

```
- val l = [1,2,3,4,5];
> val l = [1,2,3,4,5] : int list
- map twice l;
> [2,4,6,8,10] : int list
- fun listify x = [x];
> val listify = fn : 'a -> 'a list
- map listify l;
> [[1],[2],[3],[4],[5]] : int list list
```

Exercise 2.5.7 *Define a function* `powerset` *that given a set (represented as a list) will return the set of all its subsets.*

Combining the ability to take functions as values and to return functions as results, we now define the composition function. It takes two functions as argument, and returns their composition:

```
- fun compose(f,g)(x) = f(g(x));
> val compose = fn : ('a->'b * 'c->'a) -> ('c->'b)
- val fourtimes = compose(twice,twice);
> val fourtimes = fn : int->int
- fourtimes 5;
> 20 : int
```

Let's walk through this carefully. The function `compose` takes a pair of functions as argument and returns a function; this function, when applied to `x` returns `f(g(x))`. Since the result is `f(g(x))`, the type of `x` must be the domain type of `g`; since `f` is applied to the result of `g(x)`, the domain type of `f` must be the range type of `g`. Hence we get the type printed above. The function `fourtimes` is obtained by applying `compose` to the pair `(twice,twice)` of functions. The result is a function that, when applied to `x`, returns `twice(twice(x))`; in this case, `x` is `5`, so the result is `20`.

Now that you've gained some familiarity with ML, you may feel that it is a bit peculiar that declarations and function values are intermixed. So far there is no primitive expression form for functions: the only way to designate a function is to use a `fun` binding to bind it to an identifier, and then to refer to it by name. But why should we insist that *all* functions have names? There is a good reason for naming functions in certain circumstances, as we shall see below, but it also makes sense to have *anonymous* functions, or *lambda*'s (the latter terminology comes from LISP and the λ-calculus.)

Here are some examples of the use of function constants and their relationship to clausal function definitions:

```
- fun listify x = [x];
> val listify = fn : 'a->'a list
- val listify2 = fn x=>[x];
> listify2 = fn : 'a->'a list
- listify 7;
> [7] : int list
- listify2 7;
> [7] : int list
- (fn x=>[x])(7);
> [7] : int list
- val l=[1,2,3];
> val l = [1,2,3] : int list
- map(fn x=>[x],l);
> [[1],[2],[3]] : int list list
```

We begin by giving the definition of a very simple function called `listify` that makes a single element list out of its argument. The function `listify2` is exactly equivalent, except that it makes use of a function constant. The expression `fn x=>[x]` evaluates to a function that, when given an object `x`, returns `[x]`, just as `listify` does. In fact, we can apply this function "directly" to the argument `7`, obtaining `[7]`. In the last example, we pass the function denoted by `fn x=>[x]` to `map` (defined above), and obtain the same result as we did from `map listify l`.

Just as the `fun` binding provides a way of defining a function by pattern matching, so may anonymous functions use pattern-matching in their definitions. For example,

```
- (fn nil => nil | hd::tl => tl)([1,2,3]);
> [2,3] : int list
```

```
- (fn nil => nil | hd::tl => tl)([]);
> nil : int list
```

The clauses that make up the definition of the anonymous function are collectively called a *match.*

The very anonymity of anonymous functions prevents us from writing down an anonymous function that calls itself recursively. This is the reason why functions are so closely tied up with declarations in ML: the purpose of the `fun` binding is to arrange that a function have a name for itself while it is being defined.

Exercise 2.5.8 *Consider the problem of deciding how many different ways there are of changing £1 into 1, 2, 5, 10, 20 and 50 pence coins. Suppose that we impose some order on the types of coins. Then it is clear that the following relation holds*

Number of ways to change amount a using n kinds of coins
= Number of ways to change amount a using all but the first kind of coin
+ Number of ways to change amount a-d using all n kinds of coins,
where d is the denomination of the first kind of coin.

This relation can be transformed into a recursive function if we specify the degenerate cases that terminate the recursion. If $a = 0$, we will count this as one way to make change. If $a < 0$, or $n = 0$, then there is no way to make change. This leads to the following recursive definition to count the number of ways of changing a given amount of money.

```
fun first_denom 1 = 1
  | first_denom 2 = 2
  | first_denom 3 = 5
  | first_denom 4 = 10
  | first_denom 5 = 20
  | first_denom 6 = 50;

fun cc(0,_) = 1
  | cc(_,0) = 0
  | cc(amount, kinds) =
      if amount < 0 then 0
```

```
        else
          cc(amount-(first_denom kinds), kinds)
        + cc(amount, (kinds-1));

fun count_change amount = cc(amount, 6);
```

Alter this example so that it accepts a list of denominations of coins to be used for making change.

Exercise 2.5.9 *The solution given above is a terrible way to count change because it does so much redundant computation. Can you design a better algorithm for computing the result (this is hard, and you might like to skip this exercise on first reading).*

Exercise 2.5.10 (The Towers of Hanoi) *Suppose you are given three rods and n disks of different sizes. The disks can be stacked up on the rods, thereby forming "towers". Let the n disks initially be placed on rod A in order of decreasing size. The task is to move the n disks from rod A to rod C such that they are ordered in the original way. This has to be achieved under the constraints that*

1. *In each step exactly one disk is moved from one rod to another rod*
2. *A disk may never be placed on top of a smaller disk*
3. *Rod B may be used as an auxiliary store.*

Define a function to perform this task.

2.6 Polymorphism and Overloading

There is a subtle, but important, distinction that must be made in order for you to have a proper grasp of polymorphic typing in ML. Recall that we defined a polytype as a type that involved a type variable; those that do not are called monotypes. In the last section we defined a polymorphic function as one that works for a large class of types in a uniform way. The key idea is that if a function "doesn't care" about the type of a value (or component of a value), then it works regardless of what that value is, and therefore works for a wide class of types. For example, the type of `append` was seen to be `'a list * 'a list -> 'a list`, reflecting the fact that `append` does not care what the component values of the list are, only that the two arguments are both lists having elements of the same type. The type of a polymorphic function is always a polytype, and the collection of types for which it is defined is the infinite collection determined by the instances of the polytype. For example, `append` works for `int list`'s and `bool list`'s and `int*bool list`'s, and so on *ad infinitum*. Note that polymorphism is not limited to functions: the empty list `nil` is a list of every type, and thus has type `'a list`.

This phenomenon is to be contrasted with another notion, known as *overloading*. Overloading is a much more *ad hoc* notion than polymorphism because it is more closely tied up with notation than it is with the structure of a function's definition. A fine example of overloading is the addition function, `+`. Recall that we write `3+2` to denote the sum of two integers, `3` and `2`, and that we also write `3.0+2.0` for the addition of the two real numbers `3.0` and `2.0`. This may seem like the same phenomenon as the appending of two integer lists and the appending of two real lists, but the similarity is *only* apparent: the *same* append function is used to append lists of any type, but *the algorithm for addition of integers is different from that for addition for real numbers.* (If you are familiar with typical machine representations of integers and floating point numbers, this point is fairly obvious.) Thus the single symbol `+` is used to denote two different functions, and not a single polymorphic function. The choice of which function to use in any given instance is determined by the type of the arguments.

This explains why it is not possible to write `fun plus(x,y)=x+y` in ML: the compiler must know the types of `x` and `y` in order to determine which addition function to use, and therefore is unable to accept this definition. The way around this problem is to explicitly specify the type of the argument to `plus` by writing `fun plus(x:int,y:int)=x+y` so that the compiler knows that integer addition is intended. It it an interesting fact that in the absence of overloaded identifiers such as `+`, it is never necessary to include explicit type information.[5] But in order to support overloading and to allow you to explicitly write down the intended type of an expression as a double-checking measure, ML allows you to qualify a phrase with a type expression. Here are

[5]Except occasionally when using partial patterns, as in `fun f {x,...} = x`

some examples:

```
- fun plus(x,y) = x+y;
Unresolvable overloaded identifier: +
- fun plus(x:int,y:int) = x+y;
> val plus = fn : int*int->int
- 3 : bool;
Type clash in:  3 : bool
Looking for a:  bool
I have found a: int
- (plus,true): (int*int->int) * bool;
> (fn, true) : (int*int->int) * bool
- fun id(x:'a) = x;
> val id = fn : 'a -> 'a
```

Note that one can write polytypes just as they are printed by ML: type variables are identifiers preceded by a single quote.

Equality is an interesting "in-between" case. It is not a polymorphic function in the same sense that `append` is, yet, unlike `+`, it is defined for arguments of (nearly) every type. As discussed above, not every type admits equality, but for every type that does admit equality, there is a function `=` that tests whether or not two values of that type are equal, returning `true` or `false`, as the case may be. Now since ML can tell whether or not a given type admits equality, it provides a means of using equality in a "quasi-polymorphic" way. The trick is to introduce a new kind of type variable, written `''a`, which may be instantiated to any type that admits equality (an "equality type", for short). The ML type checker then keeps track of whether a type is required to admit equality, and reflects this in the inferred type of a function by using these new type variables. For example,

```
- fun member( x, nil ) = false
    | member( x, h::t ) = if x=h then true else member(x,t);
> val member = fn : ''a * ''a list -> bool
```

The occurrences of `''a` in the type of `member` limit the use of `member` to those types that admit equality.

2.7 Defining types

The type system of ML is extensible. Three forms of type bindings are available, each serving to introduce an identifier as a type constructor.

The simplest form of type binding is the *transparent type binding*, or *type abbreviation*. A type constructor is defined, perhaps with parameters, as an abbreviation for a (presumably complex) type expression. There is no semantic significance to such a binding — all uses of the type constructor are equivalent to the defining type.

```
- type intpair = int * int ;
> type intpair = int * int
- fun f(x:intpair) = let val (l,r)=x in l end ;
> val f = fn : intpair -> int
- f(3,2);
> 3 : int
- type 'a pair = 'a * 'a
> type 'a pair = 'a * 'a
- type boolpair = bool pair
> type boolpair = bool pair
```

Notice that there is no difference between `int*int` and `intpair` because `intpair` is defined to be equal to `int*int`. The only reason to qualify `x` with `:intpair` in the definition of `f` is so that its type prints as `intpair->int`.

The type system of ML may be extended by defining new compound types using a `datatype` binding. A data type is specified by giving it a name (and perhaps some type parameters) and a set of value constructors for building objects of that type. Here is a simple example of a `datatype` declaration:

```
- datatype color = Red | Blue | Yellow ;
> type color
  con Red : color
  con Blue : color
  con Yellow : color
- Red;
> Red : color
```

This declaration declares the identifier `color` to be a new data type, with constructors `Red`, `Blue`, and `Yellow`.[6] This example is reminiscent of the enumeration type of Pascal.

Notice that ML prints `type color`, without any equation attached, to reflect the fact that `color` is a new data type. It is not equal to any other type previously declared, and therefore no equation is appropriate. In addition to defining a new type, the `datatype` declaration above also defines three new value constructors. These constructors are printed with the keyword `con`, rather than `val`, in order to emphasize that they are constructors, and may therefore be used to build up patterns for clausal function definitions. Thus a `datatype` declaration is a relatively complex construct in ML: it simultaneously creates a new type constructor and defines a set of value constructors for that type.

The idea of a data type is pervasive in ML. For example, the built-in type `bool` can be thought of as having been pre-declared by the compiler as

```
- datatype bool = true | false ;
> type bool
  con true : bool
  con false : bool
```

Functions may be defined over a user-defined data type by pattern matching, just as for the primitive types. The value constructors for that data type determine the overall form of the function definition, just as `nil` and `::` are used to build up patterns for functions defined over lists. For example,

```
- fun favorite Red = true
    | favorite Blue = false
    | favorite Yellow = false ;
> val favorite = fn : color->bool
- val color = Red;
> val color = Red : color
- favorite color;
> true : bool
```

This example also illustrates the use of the same identifier in two different ways. The identifier `color` is used as the name of the type defined above, and as a variable bound to `Red`. This mixing is always harmless (though

[6]Nullary constructors (those with no arguments) are sometimes called constants.

perhaps confusing) since the compiler can always tell from context whether the type name or the variable name is intended.

Not all user-defined value constructors need be nullary:

```
- datatype money = nomoney | coin of int | note of int |
                   check of string*int ;
> type money
  con nomoney : money
  con coin : int->money
  con note : int->money
  con check : string*int->money
- fun amount(nomoney) = 0
    | amount(coin(pence)) = pence
    | amount(note(pounds)) = 100*pounds
    | amount(check(bank,pence)) = pence ;
> val amount = fn : money->int
```

The type `money` has four constructors, one a constant, and three with arguments. The function `amount` is defined by pattern-matching using these constructors, and returns the amount in pence represented by an object of type `money`.

What about equality for user-defined data types? Recall the definition of equality of lists: two lists are equal iff either they are both `nil`, or they are of the form `h::t` and `h'::t'`, with `h` equal to `h'` and `t` equal to `t'`. In general, two values of a given data type are equal iff they are "built the same way" (*i.e.*, they have the same constructor at the outside), and corresponding components are equal. As a consequence of this definition of equality for data types, we say that a user-defined data type admits equality iff each of the domain types of each of the value constructors admits equality. Continuing with the `money` example, we see that the type `money` admits equality because both `int` and `string` do.

```
- nomoney = nomoney;
> true : bool
- nomoney = coin(5);
> false : bool
- coin(5) = coin(3+2);
> true : bool
```

```
- check("TSB",500) <> check("Clydesdale",500);
> true : bool
```

Data types may be recursive. For example, suppose that we wish to define a type of binary trees. A binary tree is either a leaf or it is a node with two binary trees as children. The definition of this type in ML is as follows:

```
- datatype btree = empty | leaf | node of btree * btree ;
> type btree
  con empty : btree
  con leaf : btree
  con node : btree*btree->btree
- fun countleaves( empty ) = 0
    | countleaves( leaf ) = 1
    | countleaves( node(tree1,tree2) ) =
      countleaves(tree1)+countleaves(tree2) ;
> val countleaves = fn : btree->int
```

Notice how the definition parallels the informal description of a binary tree. The function `countleaves` is defined recursively on `btree`'s, returning the number of leaves in that tree.

There is an important pattern to be observed here: functions on recursively-defined data values are defined recursively. We have seen this pattern before in the case of functions such as `append` which is defined over lists. The built-in type τ `list` can be considered to have been defined as follows:[7]

```
- datatype 'a list = nil | :: of 'a * 'a list ;
> type 'a list
  con nil : 'a list
  con :: : ('a * ('a list)) -> ('a list)
```

This example illustrates the use of a *parametric* data type declaration: the type `list` takes another type as argument, defining the type of the members of the list. This type is represented using a type variable, `'a` in this case, as argument to the type constructor `list`. We use the phrase "type constructor" because `list` builds a type from other types, much as value constructors build values from other values.

[7]This example does not account for the fact that `::` is an infix operator, but we will neglect that for now.

Here is another example of a recursively-defined, parametric data type.

```
- datatype 'a tree = empty | leaf of 'a |
                     node of 'a tree * 'a tree ;
> type 'a tree
  con empty : 'a tree
  con leaf : 'a->'a tree
  con node : 'a tree*'a tree->'a tree
- fun frontier( empty ) = []
    | frontier( leaf(x) ) = [x]
    | frontier( node(t1,t2) ) =
        append(frontier(t1),frontier(t2));
> val frontier = fn : 'a tree -> 'a list
- val tree = node(leaf("a"),node(leaf("b"),leaf("c"))) ;
> val tree = node(leaf("a"),node(leaf("b"),leaf("c")))
           : string tree
- frontier tree;
> ["a","b","c"] : string list
```

The function `frontier` takes a `tree` as argument and returns a list consisting of the values attached to the leaves of the tree.

Exercise 2.7.1 *Design a function* `samefrontier(x,y)` *which returns true if the same elements occur in the same order, regardless of the internal structure of* `x` *and* `y`*, and returns false otherwise. A correct, but unsatisfactory definition is*

```
fun samefrontier(x,y) = (frontier x) = (frontier y)
```

This is a difficult exercise, the problem being to avoid flattening a huge tree when it is frontier unequal to the one with which it is being compared.

ML also provides a mechanism for defining *abstract types* using an `abstype` binding.[8] An abstract type is a data type with a set of functions defined on it. The data type itself is called the *implementation type* of the abstract type, and the functions are called its *interface*. The type defined by an abstype binding is abstract because the constructors of the implementation type are

[8]Abstract types in this form are, for the most part, superseded by the modules system described in the next chapter.

hidden from any program that uses the type (called a *client*): only the interface is available. Since programs written to use the type cannot tell what the implementation type is, they are restricted to using the functions provided by the interface of the type. Therefore the implementation can be changed at will, without affecting the programs that use it. This is an important mechanism for structuring programs so as to prevent interference between components.

Here is an example of an abstract type declaration.

```
- abstype color = blend of int*int*int
  with val white = blend(0,0,0)
       and red = blend(15,0,0)
       and blue = blend(0,15,0)
       and yellow = blend(0,0,15)
       fun mix(parts:int, blend(r,b,y),
               parts':int, blend(r',b',y')) =
           if parts<0 orelse parts'<0 then white
           else let val tp=parts+parts'
                    and rp = (parts*r+parts'*r') div tp
                    and bp = (parts*b+parts'*b') div tp
                    and yp = (parts*y+parts'*y') div tp
                in  blend(rp,bp,yp)
                end
  end;
> type color
  val white = - : color
  val red = - : color
  val blue = - : color
  val yellow = - : color
  val mix = fn : int*color*int*color->color
- val green = mix(2, yellow, 1, blue);
> val green = - : color
- val black = mix(1, red, 2, mix(1, blue, 1, yellow));
> val black = - : color
```

There are several things to note about this declaration. First of all, the type equation occurring right after `abstype` is a data type declaration: exactly the same syntax applies, as the above example may suggest. Following the definition of the implementation type is the interface declaration, between `with` and `end`. Examining ML's output for this declaration, we see that ML reports `type color` without an equation, reflecting the fact that it is a new type, unequal to any others. Furthermore, note that no constructors are declared as a result of the `abstype` declaration (unlike the case of data type definitions). This prevents the client from building an object of type `color` by any means other than using one of the values provided by the interface of the type. These two facts guarantee that the client is insulated from the implementation details of the abstract type, and therefore allows for a greater degree of separation between client and implementor. Among other things, this allows for more flexibility in program maintenance, as the implementation of `color` is free to be changed without affecting the client. Note, however, that the functions defined within the `with` clause *do* have access to the implementation type and its constructors, for otherwise the type would be quite useless!

Note that the insulation of the client from the implementation of the abstract type prevents the client from defining functions over that type by pattern matching. It also means that abstract types do not admit equality. If an abstract type is to support an equality test, then the implementor must define an equality function for it.

Thus there are three ways to define type constructors in ML. Transparent type bindings are used to abbreviate complex type expressions, primarily for the sake of readability, rather than to introduce a new type. Data type bindings are used to extend the type system of ML. A data type is specified by declaring a new type constructor and providing a set of value constructors for that type. Data type definitions are appropriate for specifying data that is described structurally (such as a tree), for then it is natural that the underlying structure be visible to the client. For data structures that are defined behaviorally (such as a stack or a priority queue), an abstract type definition is appropriate: the structural realization is not part of the definition of the type, only the functions that realize the defined behavior are relevant to the client.

Exercise 2.7.2 *An abstract type* `set` *might be implemented by*

```
abstype 'a set = set of 'a list
   with val emptyset: 'a set = ...
```

```
        fun singleton (e: 'a): 'a set = ...
        fun union(s1: 'a set, s2: 'a set): 'a set = ...
        fun member(e: 'a, s: 'a set): bool = ...
          | member(e, set (h::t)) = (e = h)
                                   orelse member(e, set t)
        fun intersection(s1: 'a set, s2: 'a set): 'a set = ...
    end;
```

Complete the definition of this abstract type.

Exercise 2.7.3 *Modify your solution so that the elements of the set are stored in an ordered list. [Hint: One approach would be to pass the ordering relation as an additional parameter to each function. Alternatively, the ordering relation could be supplied to those functions that create a set from scratch, and embedded in the representation of a set. The union function could then access the ordering relation from the representation of one of its arguments, and propagate it to the union set. We will return to this problem later, when a more elegant mechanism for performing this parameterization will be discussed]*

2.8 Exceptions

Suppose that we wish to define a function `head` that returns the head of a list. The head of a non-empty list is easy to obtain by pattern-matching, but what about the head of `nil`? Clearly something must be done to ensure that `head` is defined on `nil`, but it is not clear what to do. Returning some default value is undesirable, both because it is not at all evident what value this might be, and furthermore it limits the usability of the function (if `head(nil)` were defined to be, say, `nil`, then `head` would apply only to lists of lists).

In order to handle cases like this, ML has an *exception mechanism.* The purpose of the exception mechanism is to provide the means for a function to "give up" in a graceful and type-safe way whenever it is unable or unwilling to return a value in a certain situation. The graceful way to write `head` is as follows:

```
- exception Head;
> exception Head
```

```
- fun head(nil) = raise Head
    | head(x::l) = x;
> val head = fn : 'a list->'a
- head [1,2,3];
> 1 : int
- head nil;
> Failure: Head
```

The first line is an *exception binding* that declares `head` to be an exception. The function `head` is defined in the usual way by pattern-matching on the constructors of the `list` type. In the case of a non-empty list, the value of `head` is simply the first element. But for `nil`, the function `head` is unable to return a value, and instead *raises* an exception. The effect of this is seen in the examples following the declaration of `head`: applying `head` to `nil` causes the message `Failure: Head` to be printed, indicating that the expression `head(nil)` caused the exception `Head` to be raised. Recall that attempts to divide by zero result in a similar message; the internally-defined function `div` raises the exception `Div` if the divisor is 0.

With `exception` and `raise` we can define functions that flag undesirable conditions by raising an exception. But to be complete, there ought to be a way of doing something about an error, and indeed there is such a mechanism in ML, called an *exception handler*, or simply a *handler*. We illustrate its use by a simple example:

```
- fun head2 l = head(l) handle Head => 0;
> val head2 = fn : int list->int
- head2([1,2,3]);
> 1 : int;
- head2(nil);
> 0 : int
```

The expression e `handle` exn `=>` e' is evaluated as follows: first, evaluate e; if it returns a value v, then the value of the whole expression is v; if it raises the exception exn, then return the value of e'; if it raises any other exception, then raise that exception. Notice that the type of e and the type of e' must be the same; otherwise, the entire expression would have a different type depending on whether or not the left-hand expression raised an exception. This explains why the type of `head2` is `int list->int`, even though `l` does not appear to be constrained to be an integer list. Continuing the above example, head2 applies head to l; if it returns a value, then that is the value of head2; if it raises exception Head, then head2 returns 0.

Since a given expression may potentially raise one of several different exceptions, several exceptions can be handled by a single handler as follows:

```
- exception Odd;
> exception Odd
- fun foo n = if n mod 2 <> 0 then
                 raise Odd
              else
                 17 div n;
> val foo = fn : int->int
- fun bar m = foo(m) handle   Odd => 0
                            | Div => 9999 ;
> val bar = fn : int->int
- foo 0;
> Failure: Div
- bar 0;
> 9999 : int
- foo 3;
> Failure: Odd
- bar 3;
> 0 : int
- foo 20;
> 1 : int
- bar 20;
> 1 : int
```

The function foo may fail in one of two ways: by dividing by zero, causing the exception Div to be raised, or by having an odd argument, raising the exception Odd. The function bar is defined so as to handle either of these contingencies: if foo(m) raises the exception Odd, then bar(m) returns 0; if it raises Div, it returns 9999; otherwise it returns the value of foo(m).

Notice that the syntax of a multiple-exception handler is quite like the syntax used for a pattern-matching definition of a lambda. In fact, one can think of an exception handler as an anonymous function whose domain type is exn, the type of exceptions, and whose range type is the type of the expression appearing to the left of `handle`. From the point of view of type checking, exceptions are nothing more than constructors for the type `exn`, just as `nil` and `cons` are constructors for types of the form `'a list`.

It follows that exceptions can carry values, simply by declaring them to take an argument of the appropriate type. The attached value of an exception can be used by the handler of the exception. An example will illustrate the point.

```
- exception oddlist of int list and oddstring of string;
> exception oddlist of int list
  exception oddstring of string
- ... handle   oddlist(nil) => 0
             | oddlist(h::t) => 17
             | oddstring("") => 0
             | oddstring(s) => size(s)-1
```

The `exception` declaration introduces two exceptions, `oddlist`, which takes a list of integers as argument, and `oddstring`, which takes a string. The handler performs a case analysis, both on the exception, and on its argument, just as we might defined a function by pattern matching against a data type.

What happens if the elided expression in the previous example raises an exception other than *oddstring* or *oddlist*? Here the similarity to functions ends. For in the case of functions, if the match is not exhaustive, and the function is applied to an argument that fails to match any pattern, then the exception `Match` is raised. But in the case of exception handlers, the exception is *re-raised* in the hope that an outer handler will catch the exception. For example,

```
- exception Theirs and Mine;
> exception Theirs
  exception Mine
- fun f(x) = if x=0 then raise Mine else raise Theirs;
> val f = fn : int -> 'a
- f(0) handle Mine => 7;
> 7 : int
- f(1) handle Mine => 7;
Failure: Theirs
- (f(1) handle Mine => 7) handle Theirs => 8;
```

```
> 8 : int
```

Since exceptions are really values of type `exn`, the argument to a raise expression need not be simply an identifier. For example, the function `f` above might have been defined by

```
- fun f(x) = raise (if x=0 then Mine else Theirs);
> val f = fn : int -> 'a
```

Furthermore, the wild-card pattern matches any exception whatsoever, so that we may define a handler that handles all possible exceptions simply be including a "default" case, as in:

```
- ... handle _ => 0;
```

An exception binding is a form of declaration, and so may have limited scope. The handler for an exception must lie within the scope of its declaration, regardless of the name. This can sometimes lead to peculiar error messages. For example,

```
- exception Exc;
> exception Exc
- (let exception Exc in raise Exc end) handle Exc => 0;
> Failure: Exc
```

Despite appearances, the outer handler *cannot* handle the exception raised by the `raise` expression in the body of the *let*, for the inner `Exc` is a *distinct* exception that cannot be caught outside of the scope of its declaration other than by a wild-card handler.

Exercise 2.8.1 *Explain what is wrong with the following two programs.*

1.
```
exception exn: bool;
fun f x =
    let exception exn: int
     in if x > 100 then raise exn with x else x+1
    end;
f(200) handle exn with true => 500 | false => 1000;
```

2.
```
fun f x =
    let exception exn
     in if p x then a x
        else if q x then f(b x) handle exn => c x
        else raise exn with d x
    end;
f v;
```

Exercise 2.8.2 *Write a program to place n queens on an $n * n$ chess board so that they do not threaten each other.*

Exercise 2.8.3 *Modify your program so that it returns all solutions to the problem.*

2.9 Imperative features

ML supports references and assignments. References are a type-safe form of pointer to the heap. Assignment provides a way to change the object to which the pointer refers. The type τ `ref` is the type of references to values of type τ.[9] The function `ref:'a->'a ref` allocates space in the heap for the value passed as argument, and returns a reference to that location. The function `!:'a ref->'a` is the "contents of" function, returning the contents of the location given by the reference value, and the function `:= : 'a ref*'a->unit` is the assignment function.

```
- val x = ref 0;
> val x = ref(0) : int ref;
- !x;
> 0 : int
- x := 3;
> () : unit;
- !x;
> 3 : int
```

[9]At present τ must be a monotype, though it is expected that one of several proposed methods of handling polymorphic references will soon be adopted.

All reference types admit equality. Objects of type τ `ref` are heap addresses, and two such objects are equal iff they are identical. Note that this implies that they have the same contents, but the converse doesn't hold: we can have two unequal references to the same value.

```
- val x = ref 0 ;
> val x = ref 0 : int ref
- val y = ref 0 ;
> val y = ref 0 : int ref
- x=y ;
> false : bool
- !x = !y ;
> true : bool
```

This corresponds in a language like Pascal to having two different variables with the same value assigned to them: they are distinct variables even though they have the same value (at the moment). For those of you familiar with LISP, the equality of references in ML corresponds to LISP's `eq` function, rather than to `equal`.

Along with references comes the usual imperative language constructs such as sequential composition and iterative execution of statements. In ML statements are expressions of type `unit`, expressing the idea that they are evaluated for their side effects to the store, rather than their value. The infix operator "`;`" implements sequencing, and the construct `while e do e'` provides iteration.

Exercise 2.9.1 *The following abstract type may be used to create an infinite* stream *of values.*

```
abstype 'a stream = stream of unit -> ('a * 'a stream)
  with fun next(stream f) = f()
       val mkstream = stream
   end;
```

Given a stream `s`*,* `next s` *returns the first value in the stream, and a stream that produces the rest of the values. This is illustrated by the following example:*

```
- fun natural n = mkstream(fn () => (n, natural(n+1)));
> val natural = fn : int -> int stream
- val s = natural 0;
> val s = - : int stream
- val (first,rest) = next s;
> val first = 0 : int
  val rest = - : int stream
- val (next, _) = next rest;
> val next = 1 : int
```

Write a function that returns the infinite list of prime numbers in the form of a stream.

Exercise 2.9.2 *The implementation of the stream abstract type given above can be very inefficient if the elements of the stream are examined more than once. This is because the* `next` *function computes the next element of the stream each time it is called. This is wasteful for an applicative stream (such as the prime numbers example), as the value returned will always be the same. Modify the abstract type so that this inefficiency is removed by using references.*

Exercise 2.9.3 *Modify your stream abstract type so that streams can be finite or infinite, with a predicate* `endofstream` *to test whether the stream has finished.*