CMSC 22100/32100: Programming Languages

An Overview of Standard ML

M. Blume **October 2, 2008**

Contents

1 What is SML

Standard ML is a strongly typed, impure, strict functional language.

Strongly typed: Every value, expression in the language has a *type* (int, real, bool *etc.*). The compiler rejects a program that does not conform to the type system of the language.

Functional: Each expression evaluates to a *value*. Some of these values are *functions*. In fact, every function in ML is a value. Like other values, functions can be bound to variables, passed as arguments to other functions, returned as values from function calls, and stored in data structures.

Impure: Unlike in other functional languages such as Haskell, the evaluation of expressions in ML can incur *side-effects*, *e.g.*, assignment to locations within mutable data structures or I/O.

Strict: Unlike "lazy" languages such as Haskell, arguments to ML functions are evaluated before the function call is performed. This means that if one of the arguments loops forever, then so will the entire program, regardless of whether or not the function actually needed that argument. Similarly, all side-effects caused by the evaluation of the arguments occur before any side-effects caused by the evaluation of the function body.

In this class, we will use the SML/NJ compiler. SML/NJ generates reasonably fast executable code, although some other compilers, *e.g.*, MLton, outperform it on a regular basis. However, SML/NJ can be used interactively and comes with a mature programming environment. (Some people use SML/NJ for development and use MLton to speed up the final version. However, in this class, runtime performance will not be an issue.)

2 Integers, reals, booleans

Among the built-in data types are the following: `int, real, bool, 'a option`, *etc.* Standard ML calls the type of floating points numbers `real`.[1] Here are some examples.

```
$ sml
Standard ML of New Jersey v110.68 [built: Mon Sep  8 13:47:59 2008]
- (* Integers: *)
```

[1]This is a misnomer; they are really finite precision numbers.

```
- 1;
val it = 1 : int
- 2;
val it = 2 : int
- 1+2;
val it = 3 : int
-
- (* Reals: *)
- 1.0;
val it = 1.0 : real
- 2.0;
val it = 2.0 : real
- Math.sqrt(3.0*3.0+4.0*4.0);
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
val it = 5.0 : real
- (* Booleans: *)
- false;
val it = false : bool
- true;
val it = true : bool
- if true then 1 else 2;
val it = 1 : int
```

Some notes:

- Comments in Standard ML are enclosed within *comment brackets*: `(* ... *)`. Comments are nestable.
- Many library functions such as `Math.sqrt` are automatically loaded by the SML/NJ interactive system on an on-demand basis. This is what accounts for the `[autoloading ... done]` sequence of messages.

3 Variables

Variables in Standard ML are identifiers that name *values*. Once a binding for a variable is established, the variable continues to name the *same* value until it goes out of scope. In other words, Standard ML variables are *immutable*.

Imperative programming with assignment, while possible using *reference* values (see below), it is not encouraged and cannot be done using variables alone.

Assignment to variables can usually be avoided by writing iterative algorithms in recursive style. In this style, instead of changing the value of a variable one simply establishes a new binding. The new binding can use the same identifier, in which case the old binding will be shadowed and go out of scope.

As an example, consider the following iterative version of the factorial function, written in tail-recursive style:

```
- fun fac_loop (n, f) = if n = 0 then f else fac_loop (n-1, f*n);
val fac_loop = fn : int * int -> int
- fun fac n = fac_loop (n, 1);
val fac = fn : int -> int
- fac 10;
val it = 3628800 : int
```

4 Constructing new types

ML has a variety of built-in *type constructors*, and programmers can add to that variety by defining their own. A type constructor maps a sequence of types to another type. The length of the argument sequence is called the *arity* of the type constructor. The names of “ordinary” types, *e.g.*, `int`, `bool`, `real`, and so on are actually type constructors of arity 0. When forming types, the type arguments *precede* the name of the constructor.[2]

4.1 Option type

One of the predefined type constructors in Standard ML is `option`. It gives rise to the family of *option* types.

Example: Values of type `int option` come in two varieties: There is the singleton value `NONE` or the set of values of the form `SOME i` where `i` an integer.

In general, types of the form t `option` are used to describe situations where there may or may not be a value of type t. For example, the library function `Int.fromString` takes a string argument and returns an `int option`, using `NONE` to indicate that the argument could not be parsed as an integer.

```
- NONE;
val it = NONE : 'a option
- SOME 1;
val it = SOME 1 : int option
- SOME 1.0;
val it = SOME 1.0 : real option
- SOME false;
val it = SOME false : bool option
```

The `option` type is really a datatype (see below) defined by:

```
datatype 'a option = NONE | SOME of 'a
```

4.2 List type

Another predefined type constructor in Standard ML is `list`. A type t `list` contains values that are finite lists whose elements are values of type t. The

[2]If there is more than one, the arguments are given as a comma-separated list, enclosed in parentheses.

empty list is called `nil`. The keyword `nil` is special in that it is interchangeable with the notation `[]`. A non-empty list consists of a *head* element and a *tail* list (of the same type). If h is the head and t is the tail, then the entire list is h `::` t. The infix constructor `::` is sometimes pronounced "cons." It associates to the right.

A k-element list x_1 `::` $\cdots$ `::` x_k `::` `nil` can alternatively be written `[`x_1`,` $\ldots$`,` x_k`]`.

```
- nil;
val it = [] : 'a list
- 1 :: nil;
val it = [1] : int list
- true :: false :: nil;
val it = [true,false] : bool list
- [1.0,2.0,4.0];
val it = [1.0,2.0,4.0] : real list
```

The `list` type is essentially a datatype (see below) defined by:

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

However, to be precise, one must also declare `::` as an infix operator and establish the alternative syntax using square brackets. The former would be possible using a Standard ML `infix` declaration (actually, since we need right associativity, it would be an `infixr` declaration), but the latter is built-in syntax—which is why the `list` type itself must be built-in and cannot be defined from first principles.

4.3 Tuples

The built-in type constructor `*`, which is written in infix notation and may be repeated as in `int * real * bool` to form arbitrary n-ary tuple type constructors, produces types that correspond to cartesian products of other types. The values inhabiting these types are called *tuples* and are written as comma-separated sequences enclosed in round parentheses.

```
- (1, 2);
val it = (1,2) : int * int
- (1.0, 2.0);
val it = (1.0,2.0) : real * real
- (1, 2.0, true, NONE);
val it = (1,2.0,true,NONE) : int * real * bool * 'a option
```

The notation `#`i where i is an integer literal greater or equal than 1, specifying a position, can be used to project from tuples. However, it is often more convenient to get at the elements of a tuple by *pattern matching* (see below).

```
- #1 (1, 2.0);
val it = 1 : int
- #2 (1, 2.0);
val it = 2.0 : real
```

4.4 Records

Records generalize tuples by making it possible to give names (labels) to fields instead of relying on positional information. (In fact, Standard ML defines tuples to be special cases of records where the labels are numeric and form an uninterrupted sequence from 1 to some natural number n.) Projection from records uses the notation #l where l is the label name.

```
- val r = { a = 1.0, b = true, c = {a = 3.0}, d = fn x => x};
val r = {a=1.0,b=true,c={a=3.0},d=fn}
  : {a:real, b:bool, c:{a:real}, d:'a -> 'a}
- #a (#c r);
val it = 3.0 : real
- (#d r) 1;
val it = 1 : int
```

As shown in the examples above, record types have the form

$$\{l_1{:}t_1, \ldots, l_n{:}t_n\}$$

where the l_i are the labels and the t_i are the corresponding types of individual record fields. Record expressions have the form:

$$\{l_1{=}e_1, \ldots, l_n{=}e_n\}$$

4.5 Data Types

Datatype definitions in Standard ML serve several purposes at once:

- The introduce "brand new" types not equal to any other types defined elsewhere in the program (or in the library).
- The defined types can be *sum* types, which combine several different types $t_1, \ldots, t_k$ called *variants* into a single type. Some of the t_i can be identical. This does not cause confusion since variants are distinguished based on their respective *constructors* (see below).
- A datatype definition introduces constructors that are used to distinguish between variants. Constructors themselves serve a dual purpose:
  1. They act as functions that *inject* values v_i of their respective type t_i into the datatype. The value v_t resulting from such an injection remembers the original value v_i as well as the constructor c_i that was used to perform the injection.

2. When a constructor c_i is within a *pattern* (see below), it serves the purpose of determining whether or not a given value was formed by injecting some v_i into the datatype using the constructor c_i. If so, the pattern match also recovers v_i.

- Datatype definitions can define recursive types. That is, the name of the defined type may be used on the right-hand side of the definition.
- Some (or all) variants of a datatype can be constructors that do not carry a type. These constructors become *constants*, *i.e.*, brand-new values that inhabit the newly defined type. If all constructors are constants, then a datatype definition effectively becomes the definition of an *enumeration* type.
- Datatypes are type constructor, *i.e.*, they can have type arguments. In the definition, the formal parameter list consisting of type variables precedes the type constructor name. Type variables are identifiers that start with the '-character (apostrophe).

Examples:

```
$ sml
Standard ML of New Jersey v110.68 [built: Mon Sep  8 13:47:59 2008]
- (* enumerations *)
- datatype color = Red | Green | Blue;
datatype color = Blue | Green | Red
-
- (* integer trees with values on leaves *)
- datatype itree = Leaf of int | Node of itree * itree;
datatype itree = Leaf of int | Node of itree * itree
-
- (* real trees with values on internal nodes *)
- datatype rtree = RLeaf | RNode of real * rtree * rtree;
datatype rtree = RLeaf | RNode of real * rtree * rtree
-
- (* trees with integer values on leaves and real values on internal
=    nodes *)
- datatype irtree = IRLeaf of int | IRNode of real * irtree * irtree;
datatype irtree = IRLeaf of int | IRNode of real * irtree * irtree
-
- (* integer lists *)
- datatype ilist = INil | ICons of int * ilist;
datatype ilist = ICons of int * ilist | INil
-
- (* our own list type constructor; 'a is the formal argument *)
- datatype 'a mylist = MyNil | MyCons of 'a * 'a mylist;
datatype 'a mylist = MyCons of 'a * 'a mylist | MyNil
-
```

```
- (* or own option type constructor *)
- datatype 'a myoption = MyNONE | MySOME of 'a;
datatype 'a myoption = MyNONE | MySOME of 'a
```

The `option` constructor that we have seen is just a (built-in) datatype. The built-in `list` type constructor is essentially isomorpmic to `mylist`.[3] Similarly, the `bool` type constructor is also a built-in datatype consisting precisely of the two constants `false` and `true`. The type is built-in because Standard ML provides syntactic conveniences in form of `if`-expressions.[4]

We use `case` to dispatch on the constructor of a datatype value:

```
- val x = Red;
val x = Red : color
- case x of
=   Red => "rouge"
= | Green => "vert"
= | Blue => "bleu";
val it = "rouge" : string
```

An often-used trick is to define a datatype that has only a single variant to distinguish between two isomorphic types that have different purposes and should not be confused within the program:

```
- datatype money = Dollars of real;
datatype money = Dollars of real
- datatype time = Seconds of real;
datatype time = Seconds of real
```

The same trick also works well in conjunction with the use of record types. In Standard ML, projecting from a record—or, more generally, the use of *flexible record patterns*—requires that the compiler knows the "shape" of the record type, *i.e.*, the set of all its labels, including those not being selected. The constructor of a single-variant datatype can serve as a light-weight annotation that provides precisely this information:

```
- datatype person = P of { name: string, age: int, favorite: color };
datatype person = P of {age:int, favorite:color, name:string}
- fun howold (P { name, age, ... }) =
=             name ^ " is " ^ Int.toString age ^ " years old.";
val howold = fn : person -> string
- howold (P { name = "Adam", age = 23, favorite = Blue });
val it = "Adam is 23 years old." : string
```

[3]The main difference is that Standard ML provides additional syntactic conveniences for `list`.

[4]There is more, in particular: `andalso` and `orelse`, which are short-circuiting forms logical conjunction and disjunction.

5 Functions

5.1 Function values and function definitions

Functions in Standard ML take a single argument and produce a single result. Functions that we think of as taking multiple arguments can be defined in a number of ways, namely

- as taking a tuple argument,
- or, more generally, as taking a record of values as their argument,
- or via *currying*, *i.e.*, by defining functions that return functions. In particular, if conceptually we have k arguments $x_1, \ldots, k_k$ to function `f`, then `f` really takes only x_1 and returns a new function that takes the remaining arguments $x_2, \ldots, k_k$. If $k > 2$, then any of the aforementioned techniques can be applied recursively to that returned function.

A function that maps arguments of type t_1 to results of type t_2 itself has type t_1 `->` t_2. Functions do not need to have names; they can be written anonymously using the `fn`/`=>` syntax:

```
- fn x => x*2.0;
val it = fn : real -> real
```

Functions (like all other values) can be bound to names using the `val` keyword:

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

If a function is meant to be *recursive*, it cannot be declared using plain `val`, since in this case its name would not be bound within the right-hand side of the definition. To circumvent this problem, one must add the `rec` keyword:

```
- val fac = fn n => if n = 0 then 1 else n * fac (n-1);
stdIn:37.44-37.47 Error: unbound variable or constructor: fac
- val rec fac = fn n => if n = 0 then 1 else n * fac (n-1);
val fac = fn : int -> int
```

A convenient way of combining the definition of a named recursive function (*i.e.*, something one could define using `val rec`) with a `case` analysis of the arguments is to use the `fun` keyword. In fact, `fun` is the most common way of defining functions, even if they are not recursive or do not perform a case analysis:

```
- fun frenchColor Red = "red"
=   | frenchColor Blue = "bleu"
=   | frenchColor Green = "vert";
val frenchColor = fn : color -> string
```

```
- fun fac n = if n = 0 then 1 else n * fac (n-1);
val fac = fn : int -> int
- fun append (nil, ys) = ys
=   | append (x :: xs, ys) = x :: append (xs, ys);
val append = fn : 'a list * 'a list -> 'a list
```

5.2 Function application

In Standard ML, to apply a function f to an argument a we simply write f next to a. While the language does not require parentheses, we often see them anyway—either because the argument a is a tuple literal (and tuple literals themselves use parentheses), or out of habit (and because any expression whatsoever may be freely parenthesized, even where parentheses are redundant).

Function application associates to the left, which means that $f\ a\ b$ is equivalent to $(f\ a)\ b$. This is the application of a curried multi-argument function whose type is t_a `->` t_b `->` t_r where t_a is the type of a, t_b is the type of b, and t_r is the result type. Notice that `->` (which is a binary type constructor written in infix style) associates to the *right*, so that t_a `->` t_b `->` t_r is the same as t_a `->` `(`t_b `->` t_r`)`.

5.3 Curried function definitions

A curried function can be defined conveniently using the `fun` syntax:

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

This is equivalent to

```
- val rec plus = fn x => fn y => x+y;
val plus = fn : int -> int -> int
```

And since this definition does not actually use recursion, it is also the same as:

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

Notice that as far as the inner `fn`-expression is concerned, variable `x` is free (*i.e.*, bound in an outer context). The value obtained by evaluating this inner `fn`-expression *captures* the value of `x`, so it will still be available at the time(s) of its invocation(s).[5]

```
- val p1 = plus 1;
val p1 = fn : int -> int
- val p2 = plus 2;
```

[5]The data structure used internally to capture such free variables is called a *closure*, referring to the fact that it is used to "close over" free variables.

```
val p2 = fn : int -> int
- p1 3;
val it = 4 : int
- p2 3;
val it = 5 : int
```

The function values bound to `p1` and `p2` share a common underlying implementation but differ in their closures: in `p1` the value of `x` is `1` while in `p2` the value of `x` is `2`.

6 Other types

6.1 The `unit` type

The built-in type `unit` has precisely one value, written `()`. The `unit` type is used for computations that do not naturally have a meaningful result and are performed for effect only.

6.2 References—the `ref` type

The built-in type constructor `'a ref` is used to give types to *reference* values.

A reference value of tpe t `ref` is an abstract pointer (without pointer arithmetic!) to a mutable location holding a value of type t. Reference values are freshly created by an application of the *reference constructor* (whose name is also `ref`). Its argument initializes the new location. To read the location we use a function whose name is `!` (exclamation point), and to update the value we use the infix operator `:=`.

```
- val r = ref 10;
val r = ref 10 : int ref
- !r;
val it = 10 : int
- r := !r + 1;
val it = () : unit
- !r;
val it = 11 : int
```

6.3 Text—the types `char` and `string`

The type `string` is inhabited by possibly empty sequences of characters. String literals simply spell out the sequence enclosed in double quotes (`"`). If the double quote character appears within a string, it must be protected using the *escape* character \. The same is true for the escape character itself. Moreover, a number of special escape sequences is available for specifying non-printable characters within strings. In particular, \`n` stands for *newline* and \`t` stands for *tabulator*. Strings can be concatenated using the ^ infix operator. To send a string to standard output, use `print`.

White space within strings that is enclosed by escape characters is ignored. Since white space includes newlines, this can be used to conveniently spread long literals across multiple source lines (even without losing proper indentation).

```
- print "hello world";
hello worldval it = () : unit
- print "a string with\ntwo lines\n";
a string with
two lines
val it = () : unit
- print ("a string containing a \" character" ^ " along with a \\ character\n");
a string containing a " character along with a \ character
val it = () : unit
- "a string spanning \
=
=
= \multiple lines\n";
val it = "a string spanning multiple lines\n" : string
- (* let's use multi-line strings with indentation: *)
- print "this string\
=       \ also spans multiple\
=       \ lines\n";
this string also spans multiple lines
val it = () : unit
```

Characters can also be handled individually. They have type `char`. Character literals look like one-element string literals with a preceding `#`.

```
- String.sub ("abc", 1);
val it = #"b" : char
```

7 Block structure

We have seen a number of ways of *declaring* things at the interactive toplevel: value bindings (including functions) using `val`, `val rec`, and `fun`, algebraic datatypes using `datatype`, and type abbreviations using `type`. There are a few more that have not been explained in detail: *exceptions* using `exception` as well as infix status and operator precedence using `infix` and `infixr`.

These are the declaration forms of the Standard ML *core language*, and all of them can be used at arbitrary nesting levels. New nested blocks are established by `let`-expressions.

The general form of a `let`-expression is

```
let
```

    decl$_1$
    $\vdots$
    decl$_n$
`in`
    exp$_1$;
    $\vdots$
    exp$_m$
`end`

where $n \geq 0$ and $m \geq 1$. The bindings established by the declarations come into effect *incrementally*, *i.e.*, a binding established by *decl*$_i$ is visible in the right-hand side of every *decl*$_j$ where $j > i$. Declarations of the form `datatype`, `val rec` and `fun` are recursive, meaning that their right-hand sides can see the bindings established on their own left-hand sides.

There can be more than one `;`-separated expression between `in` and `end`. All of them see all the bindings established by the list of declarations. All of them except the last are evaluated for effect only. Their types should be `unit` (although, unfortunately, the language does not enforce this convention). The value *exp*$_m$ becomes the value of the entire `let`-expression.

As an example, here is an alternative definition of the iterative factorial function that does not expose the binding of the "helper" function than implements the iterative (*i.e.*, tail-recursive) loop:

```
- val fac =
=   let fun loop f n = if n = 0 then f else loop (f*n) (n-1)
=   in loop 1
=   end;
val fac = fn : int -> int
- fac 10;
val it = 3628800 : int
```

This example also shows a use of currying and a *partial application* of the curried `loop` function.

7.1 Simultaneous bindings and mutual recursion

A single declaration (one of the *decl*$_i$ forms above) can bind multiple names by stringing several individual declarations of the same kind together using the `and` keyword.

Examples:

```
- val x = 1 and y = 2;
val x = 1 : int
val y = 2 : int
- fun even n = n = 0 orelse odd (n-1)
= and odd n = n <> 0 orelse even (n-1);
```

```
val even = fn : int -> bool
val odd = fn : int -> bool
- datatype t = A of u
= and      u = B of t | C;
datatype t = A of u
datatype u = B of t | C
- type t = int and u = real;
type t = int
type u = real
```

Notice that `and` replaces the usual declaration keyword in all but the first individual declaration.

If a declaration form is not recursive (*e.g.*, `val`, or `type`, or `exception`), then using `and` causes *simultaneous* binding, meaning that *none* of the right-hand sides can see *any* of the new bindings.

If a declaration form is recursive (*e.g.*, `datatype`, or `fun`, or `val rec`), then using `and` causes *mutual recursion*, meaning that *all* of the right-hand sides can see *all* of the new bindings.

8 Patterns and pattern matching

Wherever a variable can be bound in Standard ML, we can also make use of more general *patterns*. Patterns are in some sense the dual to expression forms. They essentially form "templates" with "holes" that are named by variables. Patterns are matched against runtime values. The matching process implicitly deconstructs the values and binds their constituents to the corresponding variables.

Like expressions, patterns have types. If a pattern has type t, then only values of type t can be matched against it. A pattern of type t that matches all values of type t is called *irrefutable*.

If there is more than one rule that matches a given value, the leftmost rule takes precedence. (One can think of this as follows: Each pattern is tried in turn, beginning from the left and stopping at the first where the matching process succeeds.) A match where some of the rules are unreachable because all possible values can be matched by other rules to the left is flagged as an error by the compiler.

A *match* is a sequence of *rules* p `=>` e, each rule consisting of a pattern p on the left-hand side and an expression e on the right-hand side. Rules in a match are separated by a vertical bar `|`. All patterns in a match must have the same type t. Expressions on the right-hand sides can refer to the variables bound by their respective left-hand side patterns. Given these bindings, the expressions must all have the same type u.

If the union of the values matched by the patterns in a match covers all values of type t, then the match is said to be *exhaustive*. A non-exhaustive match can fail at runtime, causing an implicit `Match` or `Bind` exception to be raised. The compiler issues a warning when it detects a non-exhaustive match.

Matches are used in `fn`-expressions, where they define an anonymous function of type t `->` u and in `case`-expressions, where they are used to scrutinize a value of type t and dispatch into one ore more distinct branches of control, each returning a value of type u.
The syntax

$$\texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{=>}\ e_n$$

is actually syntactic sugar for

$$\texttt{(fn}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{=>}\ e_n\texttt{)}\ e$$

The pattern language mirrors (to some extend) the expression language; patterns can be made from parts that contain other patterns:

wildcard The pattern _ (underscore) has an arbitrary type t and matches all values of that type. The actual runtime value is discarded, *i.e.*, it is not bound to any variable.

variable Any variable name x (except those currently bound to datatype constructors) can be used as a pattern. Such a pattern has an arbitrary type t (which becomes the type of x within its scope) and matches any value of that type. Within each *branch* of a pattern (see *or*-patterns below), there can be at most one occurrence of each variable. If a value v matches the pattern, then x becomes bound to the part of v that corresponds to the position of x within the overall pattern.

tuple If $p_1, \ldots, p_n$ are patterns of types $t_1, \ldots, t_n$, then $(p_1, \ldots, p_n)$ is a tuple pattern of type t_1 `*` $\cdots$ `*` t_n. It matches a value $(v_1, \ldots, v_n)$ provided that for all i we have that p_i matches v_i. All variable bindings from all the sub-patterns become effective simultaneously. In function definitions, tuple patterns are often used to give the illusion of having multiple arguments.

record Similarly, under the same assumption as above, the pattern $\{\ l_1 = p_1, \ldots, l_n = p_n\ \}$ is a record pattern of type $\{\ l_1 : t_1, \ldots, l_n : t_n\ \}$ and matches record values of that type. As a syntactic shortcut, if some p_i happens to be a variable that is lexically equal to the corresponding l_i, then $l_i = p_i$ can be abbreviated simply as l_i. Record patterns can be used in function definitions to give the illusion of the function taking several *named* arguments.

flexible record pattern Under the same assumptions, the pattern $\{\ l_1 = p_1, \ldots, l_n = p_n, \ldots\ \}$ is equivalent to some pattern $\{\ l_1 = p_1, \ldots, l_n = p_n, l_{n+1} = \_, \ldots, l_{n+m} = \_\ \}$ for some $m \leq 0$ and some labels $l_{n+1}, \ldots, l_{n+m}$. An additional restriction is that the compiler must be able to determine m and $l_{n+1}, \ldots, l_{n+m}$ from the context in which the pattern is used.

The notation `#`l is a shorthand for `fn` $\{\ l, \ldots\ \}$ `=>` l.

list If $p_1, \ldots, p_n$ are patterns of type t, then `[`p_1`,`...`,`p_n`]` is a pattern of type t `list`. It matches list values `[`v_1`,`...`,`v_n`]` that are precisely n elements long provided that each element v_i matches its corresponding sub-pattern p_i.

integer Any integer literal i is a pattern of type `int` that matches precisely the value i.

string Any string literal s is a pattern of type `string` that matches precisely the value s.

data constructor Let c_i be a datatype constructor of type t_i `->` t, and let p_i be a pattern of type t_i. Then c_i p_i is a pattern of type t. It will match values of the form c_i v_i where v_i matches p_i. (These are the values of type t than were formed by applying the constructor c_i to v_i.)

If c_i has infix status, then it must be written as an infix operator: p_x c_i p_y instead of $c_i(p_x, p_y)$. A frequently occurring example of this is the "cons" constructor `::` of type `list`.

or pattern (This is a conservative extension to Standard ML implemented by SML/NJ.) If p_1 and p_2 are patterns of the same type t, then p_1 `|` p_2 is also a pattern of type t. It matches the *union* of the values matched by p_1 and p_2. The sub-patterns p_1 and p_2 must agree precisely on the set of variable that are bound by them (same set of variables, same types). Matching proceeds from left to right, meaning that if a value matches both p_1 and p_2 at the same time, then the bindings from p_1 will go into effect.

as-pattern If p is a pattern of type t and x is a variable pattern, then x `as` p is also a pattern of type t. It matches the same values v that p alone would match. In this case, in addition to the variable bindings that are established within p, the variable x is bound to the respective v itself.

reference If p is a pattern of type t, then `ref` p is a pattern of type t `ref`. It matches a value v if v denotes a location that *at the time the match is performed* contains a value v' matching p. Reference patterns are unusual in that they incur a side-effect (reading of a mutable location).

8.1 Examples

As mentioned above, patterns are used in `case`-expressions and `fn`-expressions. They are also used in function definitions in clause form (using keyword `fun`).

Examples:

```
- fun fac n =
=    case n of
=      0 => 1
=    | _ => n * fac (n-1);
val fac = fn : int -> int
```

```
-
- (* several equivalent ways of defining "reverse-and-append": *)
- fun revappend (xs, ys) =
=   case xs of
=     [] => ys
=   | x :: xs => revappend (xs, x :: ys);
val revappend = fn : 'a list * 'a list -> 'a list
-
- fun revappend2 ([], ys) = ys
=   | revappend2 (x :: xs, ys) = revappend2 (xs, x :: ys);
val revappend2 = fn : 'a list * 'a list -> 'a list
-
- val rec revappend3 =
=    fn ([], ys) => ys | (x :: xs, ys) => revappend3 (xs, x :: ys);
val revappend3 = fn : 'a list * 'a list -> 'a list
```

Notice that in `case`- and `fn`-expressions the syntax requires the use of the double-arrow `=>`, while function definitions in clause form (using `fun`) use an equal sign `=`.
The definition of a curried function in clause form

```
fun $f$ $p_{11}$ ... $p_{1n}$ = $b_1$
  | $f$ $p_{21}$ ... $p_{2n}$ = $b_2$
  ⋮
  | $f$ $p_{m1}$ ... $p_{mn}$ = $b_m$
```

de-sugars as follows:

```
fun $f$ $x_1$ ... $x_n$ =
  case ($x_1$,...,$x_n$) of
      ($p_{11}$,...,$p_{1n}$) => $b_1$
    | ($p_{21}$,...,$p_{2n}$) => $b_2$
    ⋮
    | ($p_{m1}$,...,$p_{mn}$) => $b_m$
```

assuming $x_1, \ldots, x_n$ to be fresh variables.
Here is an example of how one would use an **as**-pattern:

```
- fun f (x as (y, z)) = (x, y+z);
val f = fn : int * int -> (int * int) * int
- f (2, 3);
val it = ((2,3),5) : (int * int) * int
```

9 Other language features

9.1 Polymorphism

9.2 Modules

One of the most distinguishing features of Standard ML is its module system. It deserves a much more complete introduction. However, we will use only relatively little of its full power, so the following overview should suffice:

Structures

(First-order) Modules in Standard ML are called *structures*. Structures can be bound to module names much like values can be bound to variables. A module binding is established by a declaration that starts with the keyword `structure`.

The right-hand side of a `structure` declaration is often a structure expression, *i.e.*, a sequence of declarations enclosed within `struct` and `end`. Any declaration that can appear at top level can also appear within a structure.

One of the roles that structures play is that of *name space management*. Declarations that appear within a module are not visible directly. Any reference to a name bound within a module, when used from elsewhere, must be *qualified* with the module's name.

Structures are bound

Example:

```
- structure A = struct
=   type t = int
=   type u = t * t
=   fun f (x : t) = (x, x+1) : u
= end;
structure A :
  sig
    type t = int
    type u = t * t
    val f : t -> u
  end
- A.f 10;
val it = (10,11) : A.u
```

Signatures

Much like values that are classified by types, structures are classified by *signatures*. Programmers can explicitly write signatures, bind them to signature names, and *ascribe* them to structures.

For every structure (even in the absence of a signature ascription) the compiler infers a *principal* signature, which in some sense is the "default" signature of the structure. An ascribed signature may elide some elements and give more specific types for some values, but it must otherwise match the principal signature.

An *opaque* signature ascription can also hide the identity of certain types, thus rendering them abstract. This is Standard ML's primary way of defining abstract types. An opaque signature ascription uses the symbol `:>`, while its counterpart, *transparent* signature ascription uses `:`.

Examples:
First define the signature and bind it to `S`:

```
- signature S = sig
=     type t
=     val x : t
=     val f : t -> int
= end;
signature S =
  sig
    type t
    val x : t
    val f : t -> int
  end
```

Now define a matching structure (but still without ascription):

```
- structure M = struct
=     type t = int
=     val x = 10
=     fun f y = y  (* still polymorphic *)
      val a = "hello"
= end;
structure M :
  sig
    type t = int
    val x : int
    val f : 'a -> 'a
    val a : string
  end
- M.x;
val it = 10 : int
- M.f M.x;
val it = 10 : int
- M.f 3;
val it = 3 : int
- M.f true;
val it = true : bool
```

The transparent ascription of `S` to `M` hides `a` and makes the type of `f` less general:

```
- structure MT = M : S;
structure MT : S
- MT.x;
val it = 10 : M.t
- MT.f MT.x;  (* MT.t is the same as int *)
val it = 10 : int
- MT.f 3;
val it = 3 : int
- MT.f true;   (* MT.f is specialize to int now *)
stdIn:1.1-1.10 Error: operator and operand don't agree [tycon mismatch]
  operator domain: M.t
  operand:         bool
  in expression:
    MT.f true
```

If we use opaque ascription instead, the identity of type `t` (*i.e.*, the fact that it is defined to be `int`) becomes hidden:

```
- structure MO = M :> S;
structure MO : S
- MO.x;
val it = - : MO.t
- MO.f MO.x;  (* MO.f and MO.x still have matching types *)
val it = 10 : int
- MO.f 3;   (* but MO.t is not the same as int anymore *)
stdIn:36.1-36.7 Error: operator and operand don't agree [literal]
  operator domain: MO.t
  operand:         int
  in expression:
    MO.f 3
- MO.f true;  (* and, of course, it is not bool either *)
stdIn:1.1-1.10 Error: operator and operand don't agree [tycon mismatch]
  operator domain: MO.t
  operand:         bool
  in expression:
    MO.f true
```

Functors

In Standard ML, a *functor* is a "function" from structures to structures. Functors make it possible to write *generic* code that is parametrized not only over other values but also over other types. In this course we will have little need for writing our own functors, but we will sometimes make use of functors defined in libraries.

9.3 Exceptions

9.4 Infix declarations

10 Using Files

When you start SML/NJ using the `sml` command, you find yourself in the *interactive toplevel loop*. Code you type here is compiled to machine code on the fly and then executed right away. A pretty-printing mechanism displays results or gives a summary of the definitions that were entered.

Code entered at the interactive level is ephemeral, since the system does not save it for you. Therefore, any program longer than a few lines of code should first be saved to a text file and then read into the system. While it would be possible to use your favorite GUI's cut&paste feature to achieve the latter, SML/NJ provides a number of facilities to make the process easier.

10.1 Function `use`

Function `use` takes an argument that must be the name of a file containing Standard ML code. It opens the file and invokes the compiler on its contents. The code may refer to bindings created earlier at top level. If compilation succeeds, the code is executed. Variable bindings created at execution time are later available at the top level. In effect, the code in the file is treated in almost exactly the same way as code entered manually at the interactive prompt.

Example:

Recall our earlier definition of the `mylist` type constructor:

```
- datatype 'a mylist = MyNil | MyCons of 'a * 'a mylist;
datatype 'a mylist = MyCons of 'a * 'a mylist | MyNil
```

Now suppose the contest of file `mylistlength.sml` is:

```
fun myLength MyNil = 0
  | myLength (MyCons (x, xs)) = 1 + myLength xs
```

Invoking `use` on this file loads it into the system and makes—as indicated by the message—the binding to `myLength` available.

```
- use "mylistlength.sml";
[opening mylistlength.sml]
val myLength = fn : 'a mylist -> int
val it = () : unit
```

Notice that the result of `use` itself is `()` (the `unit` value), indicating that `use` is used *for effect*, namely the effect of creating additional toplevel bindings. Since `myLength` is now available, we are able to invoke it:

```
- myLength (MyCons (1, MyCons (2, MyCons (3, MyNil))));
val it = 3 : int
```

10.2 A note on polymorphism and type inference

Notice that the type of `myLength` is `'a mylist -> int`. Implicitly, the *type variable* `'a` is universally quantified. This means that `myLength` can be used with *any* instantiation of `mylist`. (Indeed, the implementation of `myLength` completely ignores the contents carried by the `MyCons` nodes of the list.)

The type of `myLength` (or any other *polymorphic* value) is automatically instantiated appropriately as needed wherever it is used. This process is part of *type inference*, a mechanism implemented by the compiler designed to relieve the programmer from having to write most of the type annotations that are necessary in many other statically typed programming languages.

10.3 Generativity of `datatype` definitions

Previously we put the code of `myLength` into its own file but left the definition of the corresponding type out. This means that once we exit SML/NJ the type definition is gone and has to be re-entered by hand before one can `use` the file again.

One possibility is to put the type definition into the same file as the code of the function. But perhaps we want to write many functions that operate on `mylist`s, and having all of them in the same file could be awkward. Thus, let us put the type definition into *its own* source file `mylist.sml`:

```
datatype 'a mylist = MyNil | MyCons of 'a * 'a mylist
```

We can now easily recover the type definition whenever we need it:

```
- use "mylist.sml";
[opening mylist.sml]
datatype 'a mylist = MyCons of 'a * 'a mylist | MyNil
val it = () : unit
```

Once the type is in place, we can also load the code of the function:

```
- use "mylistlength.sml";
[opening mylistlength.sml]
val myLength = fn : 'a mylist -> int
val it = () : unit
```

Finally, we can test it out:

```
- myLength (MyCons (1, MyCons (2, MyNil)));
val it = 2 : int
```

Now, suppose for some reason we *reload* the type definition:

```
- use "mylist.sml";
[opening mylist.sml]
datatype 'a mylist = MyCons of 'a * 'a mylist | MyNil
val it = () : unit
```

Apparently this works without problem. But now, if we want to run another test of `myLength`, we find trouble:

```
- myLength (MyCons (1, MyNil));
stdIn:6.1-6.29 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z ?.mylist
  operand:         int mylist
  in expression:
    myLength (MyCons (1,MyNil))
```

This rather cryptic error message indicates that the expected argument type of `myLength` does not match the type of the actual argument! Why is that?

The reason for this behavior is that reloading the definition of `mylist` has the effect of creating a brand-new type with brand-new constructors. But `myLength`, which had not been reloaded, still has the *old* type! In fact, as indicated by the question marks `?` in the error message, it now has a type that cannot even be named anymore, since the new but identically named definition of `mylist` shadows it.

10.4 CM — the SML/NJ compilation manager

The solution, of course, is to reload `mylistlength.sml` whenever we reload `mylist.sml`. More generally, to be consistent we must reload any file that *depends* on another file that has been reloaded.[6] While the compiler will always check and make sure that there are no fundamental inconsistencies (*i.e.*, those that would compromise type safety), it is a cumbersome task to manually track down all dependencies. Moreover, some inconsistencies merely result in the wrong version of some code getting executed, which is something the compiler will not tell us about.

The SML/NJ compilation manager (CM for short) is designed to help with compilation dependencies. It lets programmers define projects as collection of source files and it automatically determines their inter-dependencies. It then monitors file modification time stamps and provides a service called *smart recompilation*, *i.e.*, after a modification it determines the smallest set of files that need to be recompiled and reloaded.

Since CM inherently relies on programs being written in a style that makes use of the module system, we will come back to this topic later after we discussed the basics of ML modules.

[6]The details here are actually somewhat more complicated, but we will stick to this first approximation of an explanation.

A Reserved words

The following identifiers are *reserved words* in Standard ML. They cannot be used as names of user-defined variables, constructors, types, or modules:

`abstype and andalso as case datatype do else end eqtype exception fn fun functor handle if in include infix infixr let local nonfix of op open orelse raise rec sharing sig signature struct structure then type val where with withtype while ( ) [ ] { } , : ; ... _ | = => -> # :>`

The SML/NJ implementation of Standard ML adds the following two additional reserved words to the above list:

`abstraction funsig`

B Pre-defined infix operators

The following operators are pre-defined and have infix status. Although in principle it would be possible to re-define these names in user code, it is recommended not to do so:

`before o * / mod div ^ + - := > < >= <= = <> :: @`

C Other restricted identifiers

Although they are not classified as “reserved” (*i.e.*, they are not *keywords* that are used in parsing Standard ML), the following pre-defined identifiers are restricted in the sense that it is illegal for user code to re-bind them:

`true false nil :: ref it`