IT University
of Copenhagen

Tips for Computer Scientists

on

Standard ML (Revised)

Mads Tofte

April 5, 2008

This document and all examples in it are available from `http://www.itu.dk/people/tofte`

Preface

This note is inspired by a brilliant piece of writing, entitled *Tips for Danes on Punctuation in English*, by John Dienhart, Department of English, Odense University (1980). In a mere 11 pages, Dienhart's lucid writing gives the reader the impression that punctuation in English is pretty easy and that any Dane can get it right in an afternoon or so.

In the same spirit, this note is written for colleagues and mature students who would like to get to know Standard ML without spending too much time on it. It is intended to be a relaxed stroll through the structure of Standard ML, with plenty of small examples, without falling into the trap of being just a phrase book.

I present enough of the grammar that the reader can start programming in Standard ML, should the urge arise.

The full grammar and a formal definition of the semantics can be found in the 1997 language definition[2]. Some of the existing textbooks also contain a BNF for the language, e.g., [3]. I have tried to use the same terminology and notation as the language definition, for ease of reference.

Contents

1 Numbers

Standard ML has two types of numbers: integers (`int`) and reals (`real`).

Example 1.1 These are integer constants: `5`, `0`, `~37` □

Example 1.2 These are real constants: `0.7`, `3.1415`, `~3.32E~7` □

2 Overloaded Arithmetic Operators

Each of the binary operators +, ⋆, −, <, >, <= and >= can be applied either to a pair of integers or to a pair of reals. The function `real` coerces from `int` to `real`. Any value produced by +, ⋆ or − is of the same type as the arguments.

Example 2.1 The expression `2+3` has type `int` and the expression `2.0+real(5)` has type `real`. □

It is sometimes necessary to impose a type constraint "`:int`" or "`:real`" to disambiguate overloaded operators.

Example 2.2 The squaring function on integers can be declared by

```
fun square(x:int) = x*x
```

or, equivalently, by

```
fun square(x) = (x:int) * x
```

□

Unary minus (~) either maps an integer to an integer or a real to a real.

3 Strings

String constants are written like this: `"hello world"`. There is special syntax for putting line breaks, tabs and other control characters inside string constants. The empty string is `""`.

4 Lists

All elements of a list must have the same type, but different lists can contain elements of different types.

Example 4.1 These are lists: `[2, 3, 5]`, `["ape", "man"]`. □

The empty list is written `[]` or `nil`. The operation for adding an element to the front (i.e., the left) of a list is the right-associative, infix operator `::` , pronounced "cons." Hence, the expression `[2, 3, 5]` is short for `2::3::5::nil`.

5 Expressions

Expressions denote values. The preceding sections gave examples of constant expressions, function application expressions and list expressions. Other forms will be introduced below. We use *exp* to range over expressions.

6 Declarations

Standard ML has a wide variety of declarations, e.g., value declarations, type declarations and exception declarations. Common to them all is that a declaration binds identifiers. We use *dec* to range over declarations.

A common form of expression is

`let` *dec* `in` *exp* `end`

which makes the bindings produced by dec available locally within the expression exp.

Example 6.1 The expression

```
let val pi = 3.1415
in  pi * pi
end
```

is equivalent to `3.1415 * 3.1415`. □

Value Bindings

Value declarations bind values to value identifiers. A common form of value declaration is

val vid = exp

We use vid to range over value identifiers. Declarations can be sequenced (with or without semicolons); furthermore, a declaration can be made local to another declaration.

Example 6.2

```
val x = 3
```

Example 6.3

```
val x = 3
val y = x+x
```

Example 6.4

```
local
  val x = 3;
  val y = 5
in
  val z = x+y
end
```

In a sequential declaration $dec_1 dec_2$ (or dec_1; dec_2), the declaration dec_2 may refer to bindings made by dec_1, in addition to the bindings already in force. Declarations are (as all phrases are) evaluated left-to-right. Later declarations can shadow over earlier declarations, but they cannot undo them.

Example 6.5

```
val x = 3
val y = x
val x = 4
```

At the end of the above declaration, `x` is bound to 4 and `y` is bound to 3. □

Function-value Bindings

Function-value bindings bind functions (which are values in Standard ML) to value identifiers. A common form is

fun vid (vid_1) = exp

where vid is the name of the function, vid_1 is the formal parameter and exp is the function body. Parentheses can often be omitted. When in doubt, put them in.

Example 6.6

```
fun fac(n) =
  if n=0 then 1
  else n*fac(n-1)
val x = fac(5)   (* or fac 5,
             if one prefers *)
```

By the way, note that comments are enclosed between `(*` and `*)`; comments may be nested, which makes it possible to comment out large program fragments. □

7 Function Values

The expression `fn` vid `=>` exp denotes the function with formal parameter vid and body exp. The `fn` is pronounced "lambda".

Function-value bindings allow convenient syntax for Curried functions. Hence

```
fun f x y = (x+y):int
```

is short for

```
val f = fn x=>fn y=>(x+y):int
```

No legal function-value binding begins `fun` *vid* `=` . If one wants to bind a function value to an identifier, *vid*, without introducing formal parameters, one can write `val` *vid* `=` *exp* `.`

Infix identifiers denoting functions are sometimes called infix operators (for example in the case of +). When an infix operator is to be regarded as a function by itself, precede it by the keyword `op`.

Example 7.1 The expression

```
map op + [(1,2),(2,3),(4,5)]
```

evaluates to the list [3, 5, 9]. □

Standard ML is *statically scoped*. In particular, the values of any free value identifiers a function value may have are determined when the function value is created, not when the function is applied.

Example 7.2 Assume we have already declared a function `length` which, when applied to a list l, returns the length of l. Then the declarations below bind `y` to 18.

```
local val l = 15
in
  fun f(r) = l + r
end;
val y =
  let val l = [7,9,12]
  in f(length l)
  end
```

The two bindings involving value identifier `l` have nothing with to do with each other. □

8 Constructed Values

Standard ML has several ways of constructing values out of existing values. One way is *record formation*, which includes *pairing* and *tupling*. Another way is application of a value constructor (such as `::`). The characteristic property of a constructed value is that it contains the values out of which it is built. For example `(3,5)` evaluates to the pair $(3, 5)$ which contains 3 and 5; by contrast, `3+5` evaluates to 8, which is not a constructed value.

Pairing and Tupling

Expressions for constructing pairs and tuples are written as in Mathematics. Examples: `(2,3)`, `(x,y)`, `(x, 3+y, "ape")` . The function `#`i ($i \geq 1$) can be applied to any pair or tuple which has at least i elements; it returns the i'th element.

Records

Record expressions take the form

$$\{\,lab_1{=}exp_1,\cdots,lab_n{=}exp_n\,\} \quad (n \geq 0)$$

We use lab to range over *record labels*.

Example 8.1

```
{make = "Ford", built = 1904}
```

□

Record expressions are evaluated left-to-right; apart from that, the order of the fields in the record expression does not matter.

When lab is a label, `#`lab is the function which selects the value associated with lab from a record.

Pairs and tuples are special records, whose labels are `1`, `2` etc.

The type `unit`

There is a built-in type, `unit`, which is an alias for the tuple type `{}`. This type contains just one element, namely the 0-tuple `{}`, which is also written `()`. With a slight abuse of terminology, this one value is often pronounced "unit".

Datatype Constructors

Applying a datatype constructor *con* to a value v constructs a new value, which can be thought of as the value v fused with the "tag" *con*. (Nullary datatype constructors can be thought of as standing for just a tag.)

Example 8.2 The expression `[1]` is short for `1 :: nil`, which in turn means the same thing as `op ::(1, nil)`. In principle, the evaluation of `[1]` creates four values, namely 1, `nil`, the pair (1,`nil`) and the value :: (1,`nil`). □

9 Patterns

For every way of constructing values (see Sec. 8) there is a way of decomposing values. The phrase form for decomposition is the *pattern.* A pattern commonly occurs in a value binding or in a function-value binding:

```
val pat = exp
fun vid(pat) = exp
```

We use *pat* to range over patterns. A value identifier can be used as a pattern.

Example 9.1

```
val x = 3;
fun f(y) = x+y
```

Patterns for Pairs and Tuples

Example 9.2

```
val pair = (3,4)
val (x,y) = pair
val z = x+y
```

Here we have a pair pattern, namely `(x,y)`. □

Example 9.3 Here is an example of a function-value binding which uses a tuple pattern.

```
val mycar = {make = "Ford",
             built = 1904}
fun evolve{make = m,
           built = year}=
        {make = m ,
         built = year+1}
```

In the above tuple pattern, `make` and `built` are labels, whereas `m` and `year` are value identifiers. The same holds true of the tuple-building expressions in the example. □

There is no convenient syntax for producing from a record r a new record r' which only differs from r at one label. However, there is syntax for the implicit introduction of a value identifiers with the same name as a label: in a record pattern, *lab* = *vid* can be abbreviated *lab*, if *vid* and *lab* are the same identifier.

Example 9.4 The `evolve` function could have been declared by just:

```
fun evolve{make, built} =
  {make = make,
   built = built+1}
```

□

The *wildcard record pattern*, written `...`, can be used to extract a selection of fields from a record:

```
  val {make, built, ...} =
      {built = 1904,
       colour = "black",
       make = "Ford"}
```

The empty tuple `{}` (or `()`) can be used in patterns.

Example 9.5 This is the function, which when applied to unit returns the constant 1:

```
  fun one() = 1
```

Constructed Patterns

The syntax for patterns with value constructors resembles that of function application.

Example 9.6

```
   val mylist = [1,2,3]
   val first::rest = mylist
```

Here `first` will be bound to `1` and `rest` to `[2,3]`. Incidentally, the pattern `[first,rest]` would be matched by lists of length 2 only. □

The Wildcard Pattern

The wildcard pattern, written _ , matches any value. It relieves one from having to invent an identifier for a value in a pattern, when no identifier is needed.

Constants in Patterns

Constants of type `int`, `real` and `string` are also allowed in patterns. So are nullary value constructors (such as `nil`).

10 Pattern Matching

A *match rule* takes the form

$$pat \ \texttt{=>} \ exp$$

Matching a value v against pat will either succeed or fail. If it succeeds, the match rule binds any value identifier of pat to the corresponding value components of v. Then exp is evaluated, using these new bindings (in addition to the bindings already in force). We use $mrule$ to range over match rules.

A *match* takes the form

$$mrule_1 \ \texttt{|} \ \cdots \ \texttt{|} \ mrule_n \quad (n \geq 1)$$

One can *apply* a match to a value, v. This is done as follows. Searching from left to right, one looks for the first match rule whose pattern matches v. If one is found, the other match rules are ignored and the match rule is evaluated, as described above. If none is found, the match raises exception `Match`. Most compilers produce code which performs the search for a matching match rule very effectively in most cases.

Two common forms of expression that contain matches are the case expression and the function expression:

```
case exp of match
fn match
```

In both cases, the compiler will check that the match is exhaustive and irredundant. By exhaustive is meant that every value of the right type is matched by some match rule; by irredundant is meant that every match rule can be selected, for some value.

Example 10.1

```
fun length l =
  case l of
    [] => 0
  | _ ::rest=>1+length rest
```

This function also illustrates a use of the wildcard pattern. □

11 Function-value Bindings (revisited)

A common form of function-value binding is:

```
fun  vid  pat_1  =  exp_1
|    vid  pat_2  =  exp_2
...
|    vid  pat_n  =  exp_n
```

Example 11.1 The length function can also be written thus

```
fun length [] = 0
  | length (_::rest) =
        1 + length rest
```

□

Notice that this form of value binding uses = where the match used =>. The above form generalises to the case where vid is a Curried function of m arguments ($m \geq 2$); in this case vid must be followed by exactly m patterns in each of the n clauses.

The reserved word `and` in connection with function-value bindings achieves mutual recursion:

Example 11.2

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

□

Layered Patterns

A useful form of pattern is

vid `as` *pat*

which is called a *layered* pattern. A value v matches this pattern precisely if it matches *pat*; when this is the case, the matching yields a binding of *vid* to v in addition to any bindings which *pat* may produce.

Example 11.3 A finite map f can be represented by an association list, i.e., a list of pairs (d, r), where d belongs to the domain of f and r is the value of f at d. The function below takes arguments f, d and r and produces the represenation of a map f' which coincides with f except that $f'(d) = r$.

```
fun update f d r  =
case f of
  [] => [(d,r)]
| ((p as (d',_)):: rest)=>
    if d=d' then (d,r)::rest
    else p::update rest d r
```

□

12 Function Application

Standard ML is call-by-value (or “strict”, as it is sometimes called). The evaluation of an application expression $exp_1\ exp_2$ proceeds as follows. Assume exp_1 evaluates to value v_1 and that exp_2 evaluates to value v_2. Now v_1 can take different forms. If v_1 is a value constructor, then the constructed value obtained by tagging v_2 by v_1 is produced. Otherwise, if v_1 is a function value `fn` $match$ then $match$ is applied to v_2, bearing in mind that the values of any free value identifiers of $match$ were determined when the function value was first created; if this evaluation yields value v then v is the result of the application.

13 Type Expressions

The identifiers by which one refers to types are called *type constructors*. Type constructors can be nullary (such as `int` or `string`) or they can take one or more type arguments. An example of the latter is the type constructor `list`, which takes one type argument, namely the type of the list elements. Application of a type constructor to an argument is written postfix. For example

```
int list
```

is the type of integer lists. We use *tycon* to range over type constructors.

Type variables start with a prime (e.g. `'a`, which sometimes is pronounced "alpha").

Other type constructors are `*` (product) and `->` (function space). Here `*` binds more tightly than `->` and `->` associates to the right. Also, there are record types.

Example 13.1 Here are some of the value identifiers introduced in the previous sections together with their types:

```
x:  int
fac:  int -> int
f:  int -> int -> int
mycar:
  {make:  string, built:  int}
evolve:
  {make:  'a, built:  int}->
  {make:  'a, built:  int}
mylist:  int list
length:  'a list -> int
+ :  int * int -> int
```

Here `length` is an example of a *polymorphic* function, i.e., a function which can be applied to many types of arguments (in this case: all lists). We use *ty* to range over type expressions.

14 Type Abbreviations

A `type` declaration declares a type constructor to be an abbreviation for a type. A common form is

`type` *tycon* `=` *ty*

The `type` declaration does not declare a new type. Rather, it establishes a binding between *tycon* and the type denoted by *ty*.

Example 14.1 Here is a type abbreviation

```
type car = {make: string,
            built: int}
```

In the scope of this declaration, the type of `mycar` (Sec. 9) can be written simply `car` and the `evolve` function has type `car -> car`. □

15 Datatype Declarations

A datatype declaration binds type constructors to new types. It also introduces value constructors. If one wants to declare one new type, called *tycon*, with n value constructors $con_1, \ldots, con_n$, one can write

$$\texttt{datatype}\ tycon = con_1\ \texttt{of}\ ty_1 \\ \mid con_2\ \texttt{of}\ ty_2 \\ \cdots \\ \mid con_n\ \texttt{of}\ ty_n$$

The "`of` ty_i" is omitted when con_i is nullary. Nullary value constructors are also called *constants (of type tycon)*. The above declaration binds *tycon* to a *type structure*. The type structure consists of a *type name*, t, and a *value environment*. The type name is a stamp which distinguishes this datatype from all other datatypes. The value environment maps every con_i, which, formally, is just a value identifier, to its type. This type is is simply t, if con_i is a constant, and $\tau_i \rightarrow t$, if con_i is not a constant and τ_i is the type denoted by ty_i.

Moreover, the datatype declaration gives con_i constructor status.

Example 15.1

```
datatype colour =
  BLACK
| WHITE
```

Many ML programmers capitalize value constructors, to make it easy to distinguish them from other value identifiers. However, the built-in constructors `true`, `false` and `nil` are all lower case. □

Datatypes are recursive by default. There is additional syntax for dealing with mutually recursive datatypes (`and`), datatypes that take one or more type arguments and type abbreviations inside datatype declarations (`withtype`).

Example 15.2 The built-in `list` datatype corresponds to this declaration:

```
infixr 5 ::
datatype 'a list =
  nil
| op :: of 'a * 'a list
```

The directive `infixr 5 ::` declares the identifier `::` to have infix status and precedence level 5. Precedence levels vary between 0 and 9. There is also a directive `infix` for declaring left-associative infix status and a directive `nonfix` *id*, for cancelling the infix status of identifier *id*. □

16 Exceptions

There is a type called `exn`, whose values are called *exception values*. This type resembles a datatype, but, unlike datatypes, new constructors can be added to `exn` at will. These constructors are called *exception constructors*. We use *excon* to range over exception constructors. Formally, exception constructors are value identifiers. Many ML programmers capitalize exception constructors, to make it easy to distinguish them from other value identifiers.

A new exception constructor is generated by the declaration

`exception` *excon*

in case the exception constructor is nullary (an *exception constant*), and by

`exception` *excon* `of` *ty*

otherwise.

Most of what has been said previously about value constructors applies to exception constructors as well. In particular, one can construct a value by applying an exception constructor to a value and one can do pattern matching on exception constructors.

Example 16.1 The following declarations declare two exception constructors

```
exception NoSuchPerson
exception BadLastName of string
```

Examples of exception values are: `NoSuchPerson`, `BadLastName("Wombat")`. □

An exception value can be *raised* with the aid of the expression

`raise` *exp*

which is evaluated as follows: if *exp* evaluates to an exception value v then an *exception packet*, written $[v]$, is formed, the current evaluation is aborted and the search for a handler which can handle $[v]$ is begun.

A *handler* can be thought of a function which takes arguments of type `exn`. (Different handler functions can have different result types.) Handlers are installed by handle expressions, which are described below, not by using some form of function declaration.

Exception handlers can only be applied by evaluating a raise expression, for it is the raise expression that supplies the argument to the application. Moreover, evaluation does not return to the raise expression after the application is complete. Rather, evaluation resumes as though the handler had been applied like a normal function at the place it was installed.

Exception handlers are installed by the expression

$$\mathit{exp}\ \texttt{handle}\ \mathit{match} \qquad (1)$$

Assume that, at the point in time where (1) is to be evaluated, handlers $h_1, \ldots, h_n$ have already been installed. Think of this sequence as a stack, with h_n being the top of the stack. First we push the new handler $h_{n+1} = \texttt{fn}\ \mathit{match}$ onto the stack. Then we evaluate exp. If exp produces a value v then h_{n+1} is removed from the stack and v becomes the value of (1). But if exp produces an exception packet $[v]$, then the following happens. If v matches one of the match rules in match then h_{n+1} is removed and the result of (1) is the same as if we had applied `fn` match to v directly. But if v does not match any match rule in match, then h_{n+1} and other handlers are popped from the stack in search for an applicable handler. If one is found, evaluation resumes as though we were in the middle of an application of the handler function to argument v at the point where the matching handler was installed.

If no handler is applicable, the exception packet will abort the whole evaluation, often reported to the user as an "uncaught exception".

Example 16.2 In the scope of the previous exception declarations, we can continue

```
fun findFred [] =
      raise NoSuchPerson
  | findFred (p::ps) =
      case p of
        {name = "Fred",
         location} => location
      | _ => findFred ps

fun someFredWorking(staff)=
  (case findFred(staff) of
     "office" => true
   | "conference" => true
   | _ => false
  )handle NoSuchPerson => false
```

□

In the handle expression (1) all the expressions on the right-hand-sides of the match rules of match must have the same type as exp itself.

Corresponding to the built-in operators (for example the operations on numbers) there are built-in exceptions which are raised in various abnormal circumstances, such as overflow and division by zero. These exceptions can be caught by the ML program (if the ML programmer is careful enough to catch such stray exceptions) so that computation can resume gracefully.

17 References

Standard ML has updatable references (pointers). The function `ref` takes as argument a value and creates a reference to that value. The function `!` takes as argument a reference r and returns the value which r points to. Dangling pointers cannot arise in Standard ML. Pointers can be tested for equality. `ref` is also a unary type constructor: ty `ref` is the type of references to values of type ty. Assignment is done with the infix operator `:=`, which has type:

$$\tau\ \texttt{ref} * \tau \to \texttt{unit}$$

for all types τ. Programs that use side-effects also often use the two phrase forms

$$\texttt{let}\ dec\ \texttt{in}\ exp_1\ \texttt{;}\ \cdots\ \texttt{;}\ exp_n\ \texttt{end}$$
$$\texttt{(}exp_1\ \texttt{;}\ \cdots\ \texttt{;}\ exp_n\texttt{)}$$

where $n \geq 2$. In both cases, the n expressions are evaluated from left-to-right, the value of the whole expression being the value of exp_n (if any).

Example 17.1 The following expression creates a reference to 0, increments the value it references twice and returns the pointer itself (which now points to 2):

```
let val r = ref(0)
in r:= !r + 1;
   r:= !r + 1;
   r
end
```

□

Example 17.2 The following function produces a fresh integer each time it is called:

```
local
  val own = ref 0
in
  fun fresh_int() =
   (own:= !own + 1;
    !own
   )
end
```

□

It is possible to use references in polymorphic functions, although certain restrictions apply.

18 Procedures

Standard ML has no special concept of procedure. However, a function with result type `unit` can often be regarded as a procedure and a function with domain type `unit` can often be regarded as a parameterless procedure or function.

Example 18.1 Function `P` below has type `int ref * int -> unit`.

```
val i = ref 0;
fun P(r,v)=
  (i:= v;
   r:= v+1
  )
```

19 Input and Output

In Standard ML a *stream* is a (possibly infinite) sequence of characters through which the running ML program can interact with the surrounding world. There are primitives for opening and closing streams. There are two types of streams, `instream` and `outstream`, for input and output, respectively. There is a built-in instream, `TextIO.stdIn`, and a built-in outstream, `TextIO.stdOut`. In an interactive session they both refer to the terminal. There are functions for opening and closing streams.

Example 19.1 The built-in function

```
TextIO.output:
  outstream*string->unit
```

is used for writing strings on an outstream. Inside strings `\n` is the ASCII newline character and `\t` is the ASCII tab character. Long strings are broken across lines by pairs of `\` . Finally, `^` is string concatenation.

```
fun displayCountry(country,cap)=
  TextIO.output(TextIO.stdOut,
    "\n" ^ country
    ^ "\t" ^ cap)
fun displayCountries L =
  (TextIO.output(TextIO.stdOut,
     "\nCountries\
      \ and their capitals:\n\n");
    map displayCountry L;
    TextIO.output(TextIO.stdOut,
      "\n\n")
  )
```

The built-in function

```
TextIO.inputLine:
  instream->string
```

is used for reading a line of text terminated by newline character (or by the end of the stream).

```
val _ =
  TextIO.output(TextIO.stdOut,
    "Type a line:\n");

val myLine =
  TextIO.inputLine TextIO.stdIn
```

□

Streams are values. In particular, they can be bound to value identifiers and stored in data structures. The functions `input`, `output` and the other input/output related functions raise the built-in exception `Io` when for some reason they are unable to complete their task.

20 The top-level loop

As a novice Standard ML programmer, one does not have to use input/output operations, for Standard ML is an interactive language. The method of starting an ML session depends on the operating system and installation you use. (From a UNIX shell, the command `sml` or `mosml` might do the trick.) Once inside the ML system, you can type an expression or a declaration *terminated by a semicolon* and the ML system will respond either with an error message or with some information indicating that it has successfully compiled and executed you input. You then repeat this loop till you want to leave the system. The system remembers declarations made earlier in the session. If you are unfamiliar with typed programming, you are likely to discover that once your programs get through the type checker, they usually work!

The way to leave the ML system varies, but typing `^D` (control-D) on a UNIX installation usually gets the job done. Similarly, typing `^I` interrups the ongoing compilation or execution.

Most ML systems provide facilities that let you include source programs from a file, compile files separately (for example a make system), preserve an entire ML session in a file or create a stand-alone application.

21 Modules

All constructs described so far belong to the Core language. In addition to the Core, Standard ML has *modules*, for writing big programs. Small programs have a tendency to grow, so one might as well program with modules from the beginning.

In Standard ML, the basic form of module is called a *structure*. A *signature* is a “structure type”. Finally, a *functor* is a parameterised module.

22 Structures

In its most basic form, a structure encapsulates a Core declaration of values, types and exceptions.

A structure can be declared thus:

$$\texttt{structure}\ \mathit{strid}\ =\ \mathit{strexp} \qquad (2)$$

We use *strid* to range over *structure identifiers*. Moreover, we use *strexp* to range over *structure expressions*. A structure expression denotes a structure. One common form of structure expression is

$$\texttt{struct}\ \mathit{strdec}\ \texttt{end} \qquad (3)$$

which is called the *basic* structure expression. Here $strdec$ ranges over *structure-level declarations*, i.e. the declarations that can be made at structure level. A structure-level declaration can be simply a Core declaration dec.

A *long* identifier takes the form

$$strid_1\texttt{.}\cdots\texttt{.}strid_k\texttt{.}id \quad (k \geq 1) \qquad (4)$$

and refers to component id inside a structure, $strid_1$.

Example 22.1 The following declaration generates a structure and binds it to the structure identifier `Year`.

```
structure Year =
struct
  type year = int
  fun next(y:year)= y+1
  fun toString(y)=
    Int.toString(y)
end
```

Here `Int.toString` is a long value identifier. It refers to the `toString` function in the structure `Int`. This structure is part of the Standard ML Basis Library[1]. The `toString` function converts an integer to its string representation. □

A structure-level declaration can also declare a structure. Thus it is possible to declare structures inside structures. That is why k can be greater than 1 in (4). The inner structures are said to be *(proper) substructures* of the outer structure.

23 Signatures

A signature specifies a class of structures. It does so by specifying types, values and substructures, each of them with a description. Some common forms of specifications are:

$$\texttt{type}\ tycon \qquad (5)$$
$$\texttt{eqtype}\ tycon \qquad (6)$$
$$\texttt{type}\ tycon\texttt{=}ty \qquad (7)$$
$$\texttt{datatype}\ datdesc \qquad (8)$$
$$\texttt{val}\ vid\texttt{:}ty \qquad (9)$$
$$\texttt{structure}\ strid\texttt{:}sigexp \qquad (10)$$
$$spec_1\texttt{;}\,spec_2 \qquad (11)$$

We use $spec$ to range over specifications. The form (11) allows for sequencing of specifications. (The semicolon is optional.) A specification of the form (5) specifies the type $tycon$, without saying what the type is.

A specification of the form (6) specifies the type $tycon$ without saying what the type is, except that it must be a type that admits equality. Values can only be tested for equality if their types admit equality. Function types do not admit equality.

Finally, the specification (7) specifies the type $tycon$ and introduces it as an abbreviation for the type ty.

Example 23.1 The specification

```
type year
```

specifies a type `year` without saying what the type is, whereas the specification

```
type year=int
```

specifies a type `year` and introduces it as an abbreviation for `int`. □

There are also type descriptions for types that take one or more type parameters.

A *datatype specification* (8) looks almost the same as a datatype declaration. However, a datatype declaration generates a particular type with certain constructors of certain types, whereas a datatype specification specifies the class of *all* datatypes that have the specified constructors. Thus two datatypes can be different and still match the same datatype specification.

A *value specification* (9) specifies a value identifier together with its type.

Signatures are denoted by *signature expressions*. We use $sigexp$ to range over signature expressions. A common form of signature expression is the *basic* signature expression:

sig $spec$ end

It is possible to bind a signature by a signature identifier using a *signature declaration* of the form

signature $sigid$ = $sigexp$

We use $sigid$ to range over signature identifiers.

Example 23.2 The following signature declaration binds a signature to the signature identifier `YR`:

```
signature YR =
sig
  type year
  val next: year-> year
  val toString: year -> string
end;
```

□

No specification starts with `fun`, for functions are just values of functional type and can thus be specified using `val`.

24 Structure Matching

Analogous to the notion that a value can have a type, a structure can *match* a signature.

For a structure S to match a signature Σ, it must be the case that every component specified in Σ is matched by a component in S. The structure S may declare more components that Σ specifies.

Example 24.1 The structure `Year` matches the signature `YR`. □

Matching of value components is dependent on matching of type components.

Example 24.2 Consider

```
structure Year′ =
struct
  type year = string
  fun next(y)=y+1
  fun toString(y) =
    Int.toString(y)
end;
```

Here `Year′` does not match `YR`, for `Year′.next` would have to be of type `string -> string` (since `Year′.year` = `string`). □

25 Transparent Signature Constraints

Transparent signature constaints are used for hiding components of a structure, typically because the extra components are details of implementation that one wishes to keep inside the structure.

A structure-level declaration can take the form

structure $strid$:$sigexp$ = $strexp$ (12)

This form of signature ascription to a structure, recognizable by the use of the colon (`:`), is called a *transparent* signature constraint. Suppose $strexp$ denotes structure S and $sigexp$ denotes signature Σ. Then the above declaration checks whether S matches Σ; if so, the declaration creates a restricted view S' of S and binds S' to $strid$. The restricted view S' will only have the components that Σ specified. The type information of the components of S' will be as specified in Σ, but the constraint will not hide the identity of the types carried over from S, which is why the constraint is said to be transparent.

Example 25.1 Consider

```
structure Year1: YR =
struct
  type year = int
  fun next(y:year)= y+1
  fun toString(y) =
    Int.toString(y)
end

val s = Year1.toString 1900;
```

Here `Year1` matches `YR`. Because the signature constraint is transparent, the fact that years are integers is not hidden by the constaint, so the subsequent declaration of `s` is legal. □

26 Opaque Signature Constraints

Opaque signature constraints are used for hiding components of a structure and the identity of types of that structure, typically because the extra components and the identity of the types are details of implementation that one wishes to keep inside the structure.

An *opaque* signature constraint, distinguishable from the transparent signature constraint by the use of the keyword `:>` , can be used in a structure declaration:

structure *strid*:>*sigexp* = *strexp*

The resulting structure has only the components specified by the signature. Moreover, the only information about the identity of the types of the constrained structure is that which is specified in the signature.

Example 26.1 Consider

```
structure Year2:> YR =
struct
  type year = int
  fun next(y:year)= y+1
  fun toString(y) =
    Int.toString(y)
end

(* does not typecheck: *)
val s = Year2.toString 1900;
```

The above opaque signature constraint is legal, but it makes the type `Year2.year` different from `int`. Therefore, the declaration of `s` does not typecheck. Indeed, `Year2` is so well encapsulated that there is no way of creating a value of type `Year2.year`. So let us introduce a function which can create years out of integers:

```
signature YEAR =
sig
  eqtype year
  val ad: int->year
  val next: year-> year
  val toString: year -> string
end

structure Year:> YEAR =
struct
  type year = int
  fun next(y:year)= y+1
  fun ad i = i
  fun toString(y) =
    Int.toString(y)
end

val s = Year.toString(
          Year.ad 1900)
```

The signature `YEAR` specifies year using `eqtype`, so that using `YEAR` in an opaque constraint does not remove the ability to test values of type `year` for equality. Moreover, `YEAR` specifies a function `ad` for creating years from integers. As it happens, `Year.ad` is the identify function, but having it is useful, because it makes explicit which integer constants in the program are actually year constants. □

27 Structures as Modules

A popular style of modular programming in Standard ML is the following: take a "module" to be a declaration of a signature, followed by a structure declaration which uses the signature in a opaque signature constraint. The signature then serves an the "interface" to the module.

Example 27.1 Here is a module which implements a structure, `Ford`, using the `Year` structure defined in Example 26.1, giving `Ford` an interface called `MANUFACTURER`:

```
signature MANUFACTURER =
  sig
    type year
    type car
    val first: car
    val built: car -> year
    val evolve: car  -> car
    val toString: car -> string
  end;

structure Ford:> MANUFACTURER =
  struct
    type year = Year.year
    type car = {make: string,
                built: Year.year,
                price: real}
    fun built(c:car) = #built c
    val first =
      {make = "Ford Model A",
       built= Year.ad 1903,
       price = 750.0} (*USD*)
    (* assume 4 % inflation
       per year: *)
    val yr_inflation = 1.04
    fun evolve(c:car)=
      {make= #make c,
       built= Year.next(built c),
       price= yr_inflation
               * #price c}
    fun toString(c)=
      #make c ^
      "  USD "  ^
      Int.toString(
        Real.floor(#price c))
  end;
```

Here `MANUFACTURER` specifies a type `car`, without revealing what the type is. It also specifies a type `year`, in order to specify the type of `built`. The structure implements cars as triples consisting of the name of the car, the year it was built, and the price of the car. Function `evolve` takes a car as argument and returns a car which is one year younger and 4 % more expensive. Note the long identifiers `Year.year`, `Year.ad` and `Year.next`, which show that the `Ford` structure depends on the `Year` structure. □

This style of programming sometimes requires that one reveals more information about the types of the constained structure than a completely opaque signature constraint does.

Example 27.2 Consider what happens if we, subsequent to the declaration of `Ford`, try the following:

```
val s = Year.toString(
          Ford.built Ford.first)
```

This will not type check, because the opaque signature constraint on the declaration of `Ford` made type `Ford.year` different from type `Year.year`. □

Fortunately, there are several ways in which one can make signature constraints less opaque without making them completely transparent.

One solution is to use a type abbreviation (7) in the signature.

Example 27.3 Below we have inserted a type abbreviation in the signature and renamed it, to avoid confusion between the two signatures. Nothing else has changed.

```
signature MANUFACTURER' =
  sig
    type year = Year.year
    type car
    val first: car
    val built: car -> year
    val evolve: car  -> car
    val toString: car -> string
  end;

structure Ford:> MANUFACTURER' =
  struct
    type year = Year.year
    type car={make: string,
              built: Year.year,
              price: real}
    fun built(c:car) = #built c
    val first =
      {make = "Ford Model A",
       built= Year.ad 1903,
       price = 750.0} (*USD*)
     (* assume 4 % inflation
       per year: *)
    val yr_inflation = 1.04
    fun evolve(c:car)=
      {make= #make c,
       built= Year.next(built c),
       price= yr_inflation
              * #price c}
    fun toString(c)=
      #make c ^
      "  USD "  ^
      Int.toString(
        Real.floor(#price c))
  end;

  val s= Year.toString(
           Ford.built Ford.first)
```

Now the program typechecks. □

Type Realisations

Another way of making signatures less opaque is to use a *type realisation*:

$sigexp$ `where type` $longtycon$ `=` ty

The *long* indicates that we can use a long type constructor — see Sec. 22. This form of signature expression supplements the basic signature expression introduced in Sec. 23. It creates from $sigexp$ a less opaque signature in which $longtycon$ abbreviates ty.

Example 27.4 We can use a type realisation instead of a type abbreviation to get a `Ford` structure with the desired transparency of the `Ford.year` type. The body of the structure is unchanged, only the signature in the opaque signature constraint is new:

```
structure Ford:>
      MANUFACTURER
      where type year=Year.year =
  struct
    type year = Year.year
    type car = {make: string,
                built: Year.year,
                price: real}
    fun built(c:car) = #built c
    val first =
      {make = "Ford Model A",
       built= Year.ad 1903,
       price = 750.0} (*USD*)
     (* assume 4 % inflation
       per year: *)
    val yr_inflation = 1.04
    fun evolve(c:car)=
      {make= #make c,
       built= Year.next(built c),
       price= yr_inflation
              * #price c}
    fun toString(c)=
      #make c ^
      "  USD "  ^
      Int.toString(
        Real.floor(#price c))
  end;

(* now ok: *)
val s = Year.toString(
          Ford.built Ford.first)
```

□

28 Functors

A functor is a parameterised module. A common form of *functor declaration* is

`functor` *funid* `(`*strid*`:`*sigexp*`)` `:`*sigexp′*`=` *strexp*

We use *funid* to range over *functor identifiers*. The structure identifier *strid* is the *formal parameter*, *sigexp* is the *parameter signature*, *sigexp′* is the *result signature* and *strexp* is the *body* of functor *funid*. The transparent constraint `:` *sigexp′* (i.e., the result signature) can be omitted. Alternatively, it can be replaced by an opaque signature constraint: `:>`*sigexp′*.

The result signature, when present, restricts the result of the functor application, as described in Sec. 25 and Sec. 26.

Functor application takes the form

funid `(`*strexp*`)`

and is itself a structure expression; *strexp* is the *actual argument*.

There is an alternative form for functor declaration, namely

`functor` *funid* `(`*spec*`)` `:`*sigexp′*`=`*strexp*

and a corresponding alternative form for functor application

funid `(`*strdec*`)`

These forms make it look like functors can take more than one parameter. (Formally, the alternative forms are just syntactic sugar for a functor which has one, anonymous structure argument.)

Example 28.1 Here is a functor, `Manufacturer`, which is first declared and then applied twice, to obtain two different car manufacturer structures.

```
functor Manufacturer(
  structure Y: YEAR
  val name: string
  val first: Y.year
  val usd: int
  ): MANUFACTURER=
  struct
    type year = Y.year
    type car = {make: string,
                built: Y.year,
                price: real}
    fun built(c:car) = #built c
    val first ={make = name,
                built= first,
                price= real usd}
    (* assume 4 % inflation
       per year: *)
    val yr_inflation = 1.04
    fun evolve(c:car)=
      {make= #make c,
       built= Y.next(built c),
       price= yr_inflation
              * #price c}
    fun toString(c)=
      #make c ^
      "  USD "  ^
      Int.toString(
        Real.floor(#price c))
  end;

structure Ford=
  Manufacturer(
    structure Y= Year
    val name = "Ford Model A"
    val first= Y.ad 1903
    val usd = 750
  )

structure Honda=
  Manufacturer(
    structure Y= Year
    val name = "Honda S500"
    val first = Y.ad 1963
    val usd = 1275
  )
```

Because the result signature constraint is transparent, the types `Ford.year`, `Honda.year` and `Year.year` are equal. However, the transparency of the signature constraint also results in the types `Ford.car` and `Honda.car` being exposed as the tuple types they are. If one wants to prevent this, without loosing the equality of the year types, once can use an opaque result signature together with a type realisation:

```
functor Manufacturer(
  structure Y: YEAR
  val name: string
  val first: Y.year
  val usd: int
  ):> MANUFACTURER where
      type year=Y.year   =
struct
  (* ... as before... *)
end
```

In subsequent examples, we will assume the latter definition of `Manufacturer`. □
There are many advantages to using functors.

First, one can develop and typecheck the functor before one has written the structures to which it is eventually going to be applied. This gives increased flexibity in the development proces.

Example 28.2 Referring to Example 28.1, we can develop and typecheck `YEAR`, `MANUFACTURER` and `Manufacturer` before we decide on the implementation of `YEAR`. □

Second, one can use a functor to make dependencies on other structures explicit by declaring these structures as parameters to the functor. This can make the program easier to read than if one has to read throught the functor body to detect dependencies.

Example 28.3 In example Example 27.1, the dependency of `Ford` on `Year` is burried inside the body of `Ford`, in the form of long identifiers starting with `Year`. By contrast, in Example 28.1, we see immediately from the formal parameters of `Manufacturer`, that the module depends on a structure that implements years plus various values (`name`, `first` and `usd`). □

Third, during the type checking of the body of a functor, all that is assumed about the formal parameter is what the parameter signature reveals. This removes dependencies on implentation choices in previosly declared structures.

Example 28.4 Referring to Example 28.1, any attempt inside the body of the functor to assume, for example, that `Y.year` is `int` will make the type checker reject the functor declaration. □

Fourth, whenever a functor has been successfully type-checked, it can be applied to any structure which matches the parameter signature and the application is certain to yield a well-typed structure. So changes to the actual parameter structure will not force changes to the functor body, as long as the revised argument structure matches the formal parameter signature of the functor.

Example 28.5 Referring to Example 28.1, changes to the `Year` structure do not force changes to the `Manufacturer` functor, as long as the revised `Year` structure matches `YEAR`. □

Finally, a functor can be applied to different structures, so that one can get different instances of the functor body without copying the code of the functor body.

Example 28.6 Referring to Example 28.1, we saw that the functor was applied twice to obtain two different structures, `Ford` and `Honda`. □

Specification of Sharing

Sometimes it is necessary to specify that two (or more) types in different formal parameter structures of a functor are equal. The specification

spec `sharing type`
$longtycon_1 = \ldots = longtycon_n$

specifies that the types $longtycon_1, \ldots, longtycon_n$, which must all be specified in *spec*, are equal (without saying what they are).

Example 28.7 Below is a functor `InflationTable`. It has formal parameter structures `Y: YEAR` and `M: MANUFACTURER`. It produces af function `print`, which, when applied to unit, prints a table of the price of the original car adjusted for inflation, from the year it was first produced till 2020.

```
functor InflationTable(
  structure Y: YEAR
  structure M: MANUFACTURER
  sharing type Y.year = M.year):
   sig
     val print: unit -> unit
   end=
struct
  fun line(y,c) =
    if y= Y.ad 2020 then ()
    else
      (TextIO.output(
         TextIO.stdOut,
         Y.toString y
         ^ "\t" ^ M.toString c
         ^ "\n");
       line(Y.next y,
            M.evolve c )
      )
  val y0 = M.built M.first
  fun print() = line(y0,M.first)
end;

structure P1 = InflationTable(
  structure Y = Year
  structure M = Ford);

structure P2 = InflationTable(
  structure Y = Year
  structure M = Honda);

val _ = P1.print();
val _ = P2.print();
```

The sharing constraint is necessary because the body of the functor assumes the equality of the two types, in the expression `line(y0,M.first)`, where `(y0,M.first)` has type `M.year * M.car` and function `line` has type `Y.year * M.car -> unit`. □

It is also possible to specify sharing between entire structures, which can be useful if one programs with substructures.

29 Programs

A *top-level* declaration is either a functor declaration, a signature declaration or a structure-level declaration. A *program* is a top-level declaration terminated by a semicolon and constitutes the unit of compilation in an interactive session (see Sec. 20). Notice that structures can be declared inside structures, but signatures and functor can be declared at the top level only.

30 Further Reading

The Definition of Standard ML[2] defines Standard ML formally. The Standard ML Basis Library[1] contains a large collection of modules and is recommended to anyone who wants to program real applications in Standard ML.

The textbooks on Computer Programming, using Standard ML as a programming language, include Paulson's book [3].

All examples in this document were checked on the Moscow ML implementation of Standard ML version 2.1, which is available from `www.itu.dk/people/sestoft`. Other Standard ML implementations include Standard ML of New Jersey, which is available from `www.smlnj.org`, and the ML Kit, which is available from `www.itu.dk/research/mlkit`.

References

[1] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[2] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[3] Laurence C. Paulson. *ML for the Working Programmer 2nd Edition*. Cambridge University Press, 1996.