Tips for Computer Scientists

on

Standard ML

Mads Tofte

Department of Computer Science, University of Copenhagen, December 20, 1993

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. Of course this is completely false, as Dienhart no doubt immediately would point out.

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 language definition[9]. Some of the existing textbooks also contain a BNF for the language[10,5]. 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 variables. A common form of value declaration is

`val` *var* `=` *exp*

We use *var* to range over value variables. 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 variables. A common form is

`fun` *var*`(`var_1`)` `=` *exp*

where *var* is the name of the function, var_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` *var* `=>` *exp* denotes the function with formal parameter *var* 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` *var* `=` . If one wants to bind a function value to a variable, *var*, without introducing formal parameters, one can write `val` *var* `=` *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 variables 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 variable `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` *var*`(`*pat*`)` `=` *exp*

We use *pat* to range over patterns. A value variable 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 car = {make = "Ford",
           built = 1904}
fun modernize{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 variables. 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 variable with the same name as a label: in a record pattern, *lab* `=` *var* can be abbreviated *lab*, if *var* and *lab* are the same identifier.

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

```
fun modernize{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 a variable for a value in a pattern, when no variable 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 `=>` *exp*

Matching a value v against *pat* will either succeed or fail. If it succeeds, the match rule binds the variable of *pat* (if any) 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` *var* pat_1 `=` exp_1
`|` *var* pat_2 `=` exp_2
$\cdots$
`|` *var* 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 *var* is a Curried function of m arguments ($m \geq 2$); in this case *var* 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

var `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 *var* 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 variables 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 variables introduced in the previous sections together with their types:

```
x:  int
fac:  int -> int
f:  int -> int -> int
modernize:
  {make:  string, built:  int}->
  {make:  string, 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 alias 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 the `modernize` function (Sec. 9) can be written simply `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

$$\begin{array}{ll}\texttt{datatype}\ \mathit{tycon}\ \texttt{=} & \mathit{con}_1\ \texttt{of}\ \mathit{ty}_1 \\ & \texttt{|}\ \mathit{con}_2\ \texttt{of}\ \mathit{ty}_2 \\ & \cdots \\ & \texttt{|}\ \mathit{con}_n\ \texttt{of}\ \mathit{ty}_n\end{array}$$

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 *constructor environment*. The type name is a stamp which distinguishes this datatype from all other datatypes. The constructor environment maps every con_i to its type. This type is is simply t, if con_i is a constant, and $\tau_i \to t$, if con_i is not a constant and τ_i is the type denoted by ty_i.

Moreover, the datatype declaration implicitly introduces every con_i as a value variable and binds it to the constructor con_i.

Example 15.1

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

Many ML programmers capitalize value constructors, to make it easy to distinguish them from value variables. 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 could have been declared by the programmer as follows:

```
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.

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

exp `handle` *match* (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 $h_{n+1} =$ `fn` *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 compared 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

$$\texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}_1\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{exp}_n\ \texttt{end}$$
$$(\mathit{exp}_1\ \texttt{;}\ \cdots\ \texttt{;}\ \mathit{exp}_n)$$

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 `std_in` and a built-in outstream `std_out`. 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

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

The built-in function

```
input:  instream*int->string
```

is used for reading: `input(`*is*`,`*n*`)` reads the first *n* characters from instream *is*, if possible. Here is a function for reading in a line terminated by a newline character (or by the end of the stream):

```
fun readLine(is) =
  let val c = input(is,1)
  in
    case c of
      "\n" => c
    | ""   => c (* end of
                 stream *)
    |  _   => c ^ readLine(is)
  end
```

□

If fewer than *n* characters are available on *is*, then `input(`*is*`,`*n*`)` waits for the remaining characters to become available, or for the stream to be closed.

Streams are values. In particular, they can be bound to variables 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` 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.

The principal concepts in Standard ML Modules are *structures*, *signatures* and *functors*. Very roughly, these correspond to values, types and functions, respectively. However, they live at a "higher level". For example, a structure can contain values, types, exceptions — in short all the things we saw how to declare in the Core. A signature is a "structure type" (so it will have to give some "type" to types declared in the structure). Finally, a functor is roughly a function from structures to structures. It is by using functors that one really can exploit the power of the Standard ML Modules.

22 Structures

A structure can be declared thus:

$$\texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \tag{2}$$

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

$$\texttt{struct}\ \mathit{strdec}\ \texttt{end} \tag{3}$$

which is called the *generative* structure expression (because it generates a fresh structure). 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*.

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

```
structure Ford =
struct
  type car = {make: string,
              built: int}
  val first ={make = "Ford",
              built= 1904}
  fun mutate (car:car) year=
      {make = #make car ,
       built = year}
  fun built(c:car) = #built c
  fun show(c)=
    if built c< built first
    then " - "
    else " (generic Ford) "
end
```

□

A *long* identifier takes the form

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

and is used for referring to structure components.

Example 22.2

```
structure Year =
struct
  type year = int
  val first = 1900
  val final = 2000
  fun new_year(y:year) =
    y+1
  fun show(y) = Int.string(y)
end
```

Here `Int.string` is a long value variable. It refers to the `string` function in the structure `Int`. (This structure is part of the Edinburgh Standard ML Library. The `string` 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.

Example 22.3

```
structure MutableCar=
struct
  structure C = Ford
  structure Y = Year
end
```

□

23 Signatures

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

$$\texttt{type}\ \ typdesc \qquad (5)$$
$$\texttt{datatype}\ \ datdesc \qquad (6)$$
$$\texttt{val}\ \ var\ :\ \ ty \qquad (7)$$
$$\texttt{structure}\ \ strid:\ \ sigexp \qquad (8)$$
$$spec_1 ; spec_2 \qquad (9)$$

We use *spec* to range over specifications. The form (9) allows for sequencing of specifications. (The semicolon is optional.) In (5), *typdesc* to range over *type descriptions*. The simplest type description is simply a type constructor.

Example 23.1

```
type year
```

□

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

A datatype specification (6) 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 (7) specifies a value variable together with its type.

The last form of specification listed above is the structure specification (8). An example of a structure specification is given at the end of this section.

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

`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 `MANUFACTURER`:

```
signature MANUFACTURER =
sig
  type car
  val first: car
  val built: car -> int
  val mutate: car -> int -> car
  val show: car -> string
end
```

□

No specification starts with `fun` (for functions are just values of functional type).

In a type specification, one can use `eqtype` instead of `type` to indicate that the type must admit equality. Values can only be compared for equality if their types admit equality. Function types do not admit equality.

Example 23.3

```
signature YEAR =
sig
  eqtype year
  val first: year
  val final: year
  val new_year: year-> year
  val show: year -> string
end
```

□

A signature identifier can be used as a signature expression.

Example 23.4

```
signature Sig=
sig
  structure C: MANUFACTURER
  structure Y: YEAR
end
```

□

24 Structure Matching

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 `Ford` matches `MANUFACTURER`. □

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

Example 24.2 Consider

```
structure Year' =
struct
  type year = string
  fun new_year(y)=y+1
  fun show(y)=y
  val first = 1900
  val final = 2000
end
```

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

25 Signature Constraints

A common form of structure-level declaration is

`structure` *strid* `:` *sigexp* `=` *strexp*

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 signature constraint can be used to keep down the number of long identifiers in scope (note that *strexp* can be some huge generative structure expression which declares perhaps hundreds of functions and dozens of types). However, the value and `type` components that stay in scope are not themselves affected by the constraint. In the case of a `datatype` component, the constraint may make all the constructors inaccessible, but it does not generate a new type. In the case of a structure component, the constraint may recursively restrict the components of the substructure.

Example 25.1

```
structure Year1:YEAR =
struct
  type year = int
  val first = 1900
  val final = 2000
  fun new_year(y)=y+1
  fun decade y =
    (y-1900)div 10
  fun show(y) =
    if y<1910 orelse y>= final
    then Int.string y
    else "the '"
       ^Int.string (decade y)
       ^ "0s"
end;
val long_gone = Year1.show 1968;
```

□

26 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 constraint :*sigexp′* (i.e., the result signature) can be omitted.

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 is the fundamental form of abstraction provided by ML Modules and perhaps the main source of their strength.

Whenever a functor has been 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.

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

Functor application takes the form

funid`(`*strexp*`)`

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

Example 26.1

```
functor ProductLine(M: MANUFACTURER)=
struct
  (* print table of car models
     till the year 2000 *)
  fun line(y,c) =
    if y=2000 then ()
   else
      (output(std_out,
             "\n" ^ Int.string y
             ^"\t" ^ M.show c);
       line(y+1, M.mutate c (y+1))
      )
  fun show() = line(M.built M.first,
                    M.first)
end;
structure FordLine=ProductLine(Ford);
val _ = FordLine.show()
```

□

Imagine that we define another manufacturer, possibly with a completely different `car` type. As long as the manufacturer matches the `MANUFACTURER` signature, the ability to print the new product line is obtained by a single functor application.

Similarly, if we want to modify the `Ford` structure (Ford might object to the present `show` function) we can do so without touching the `Productline` functor. To get a new `FordLine` structure, it suffices to re-apply the functor to the revised structure.

27 Sharing

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. (In reality the alternative forms are just syntactic sugar for a functor which has one, anonymous structure argument.)

Example 27.1

```
functor GrandTable(
  structure M: MANUFACTURER
  structure Y: YEAR) =
struct
  (* print table of manufacturer's
     cars till the year 2000 *)
  fun line(y,c) =
    if y=2000 then ()
    else
      (output(std_out,
             "\n" ^ Y.show y
             ^"\t" ^ M.show c);
       line(Y.new_year y,
            M.mutate c (y+1))
      )
  val y0 = M.built M.first
  fun show() = line(y0, M.first)
end;
```

□

Interestingly, `GrandTable` is syntactically correct, but it does not type-check. The problem is that we have two conflicting assumptions about `y`. On the one hand, the expression `Y.show y` says that `y` must have the (arbitrary) type `year`, which was specified in the `YEAR`. On the other hand, the expression `y+1` only makes sense if this arbitrary type happens to be `int`. Since functor application must be possible for all actual arguments that match the parameter signature, we must refute this functor.

By default, types specified in the parameter signature of a functor are different from each other and from all already existing types, during the type-checking of the functor body. The phrase form for diminishing the distinctness of specified types is the *sharing specification*. One form of type sharing specification is:

$$\texttt{sharing type } longtycon_1 \texttt{ = } longtycon_2 \tag{10}$$

(The *long* indicates that we can use a long type constructor — see Sec. 22) This form of specification supplements the ones discussed in Sec. 23.

Example 27.2 The problem with `GrandTable` can be solved by inserting a sharing specification:

```
functor GrandTable(
  structure M: MANUFACTURER
  structure Y: YEAR
  sharing type Y.year = int) =
```

□

Sharing specifications that refer to existing types are said to specify *external* sharing. External sharing runs against the desire to keep types abstract. Moreover, there are many cases of external sharing which cannot easily be expressed.

Example 27.3 The following is not grammatically correct:

```
sharing type Y.year = int * int
```

□

A sharing specification can also specify sharing between two specified types. This is called *internal* sharing.

Example 27.4 Another explanation of the difficulty in the above example is that MANUFACTURER is a bad signature, for the type it specifies for mutate implicitly assumes that years are integers! Here is an alternative signature:

```
signature MANUFACTURER =
sig
  type car
  type year
  val built: car -> year
  val first: car
  val mutate: car -> year -> car
  val show: car -> string
end
```

Now we can specify that Y.year and M.year must share. We do not even have to rely on them sharing with int, provided we change 2000 to Y.final and y+1 to Y.new_year y. The final functor is:

```
functor GrandTable(
  structure M: MANUFACTURER
  structure Y: YEAR
  sharing type Y.year = M.year)=
struct
 (* print table of manufacturer's
    cars till the final year *)
  fun line(y,c) =
    if y= Y.final then ()
    else
      (output(std_out,
             "\n" ^ Y.show y
             ^"\t" ^ M.show c);
       line(Y.new_year y,
            M.mutate c
             (Y.new_year y))
      )
  val y0 = M.built M.first
  fun show() = line(y0,M.first)
end;
```

This functor does not rely on any assumption about how Y and M represent years, except that they do it in the same way and that years can be compared with equality!

Provided we modify the declaration of Ford to include the line type year = int, we can complete the GrandTable example:

```
structure GT =
   GrandTable(structure M= Ford;
              structure Y= Year1
             );
GT.show();
```

□

Sharing can be specified between structures by the specification

$$\texttt{sharing}\ \mathit{longstrid}_1\ \texttt{=}\ \mathit{longstrid}_2$$

This specifies sharing between the structures S_1 and S_2 which $\mathit{longstrid}_1$ and $\mathit{longstrid}_2$ are bound to. In addition it implicitly specifies that

1. S_1.*longstrid* and S_2.*longstrid* share, for all structures *longstrid* that are visible in both S_1 and S_2;

2. S_1.*longtycon* and S_2.*longtycon* share, for all types *longtycon* that are visible in both S_1 and S_2;

28 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.

29 Further Reading

The Definition of Standard ML[9] defines Standard ML formally. It is accompanied by a Commentary[8]. Milner's report on the Core Language[7], MacQueen's modules proposal[6] and Harper's I/O proposal were unified in[12].

Several books on Computer Programming, using Standard ML as a programming language, are available[5,11,10,13, 3]. In addition, there are medium-length introductions[4,14].

Compilation techniques are treated by Appel[1]. In this note we have used bits of The Edinburgh Standard ML Library[2].

There is a large body of research papers related to ML, none of which we will cite on this occasion.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Dave Berry. *The Edinburgh SML Library.* Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, April 1991.

[3] Chris Clarck Colin Myers and Ellen Poon. *Programming with Standard ML.* Prentice Hall, 1993.

[4] Robert Harper. *Introduction to Standard ML.* Technical Report ECS-LFCS-86-14, Dept. of Computer Science, University of Edinburgh, 1986.

[5] Åke Wikström. *Functional Programming Using Standard ML. Series in Computer Science*, Prentice Hall, 1987.

[6] D. MacQueen. Modules for Standard ML. In *Conf. Rec. of the 1984 ACM Symp. on LISP and Functional Programming*, pages 198–207, Aug. 1984.

[7] Robin Milner. *The Standard ML Core Language*. Technical Report CSR-168-84, Dept. of Computer Science, University Of Edinburgh, October 1984. Also in[12].

[8] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[10] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[11] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[12] David MacQueen Robert Harper and Robin Milner. *Standard ML*. Technical Report ECS-LFCS-86-2, Dept. of Computer Science, University Of Edinburgh, March 1986.

[13] Ryan Stansifer. *ML Primer*. Prentice Hall, 1992.

[14] Mads Tofte. *Four Lectures on Standard ML*. LFCS Report Series ECS-LFCS-89-73, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, U.K., March 1989.