Learning Standard ML

COMP 105

Contents

This guide is available both in HTML[1] and PDF[2].

For someone with a background in COMP 11 and COMP 15, the fastest and easiest way to learn Standard ML is to buy Ullman's book and work through chapters 2, 3, 5, and 6. But many students choose not to buy Ullman—a move that saves money but costs time. You can recover some of the time by reading this guide[3]; it enumerates the most important concepts, and it tells you where to find key information, not just in Ullman, but also in three other sources:

- Jeff Ullman's *Elements of ML Programming* (ML'97 edition)
- Norman Ramsey's *Programming Languages: Build, Prove, and Compare*
- Mads Tofte's "Tips for Computer Scientists on Standard ML (Revised)[4]"
- Bob Harper's draft *Programming in Standard ML*[5]

Know your sources! Mads Tofte and Bob Harper both worked with Robin Milner on the design of Standard ML, and they helped write the *Definition of Standard ML*. They know what they're talking about, and they have good taste—though Tofte's use of the ML modules is considered idiosyncratic. Norman Ramsey at least knows some functional programming. Jeff Ullman, by contrast, got his start in the theory of formal languages and parsing, then switched to databases. He may like ML, but he doesn't understand it the way the others do.

Key concepts: Algebraic data types, case expressions, and pattern matching

Here is an excerpt from a chapter of Ramsey that is not included in the abridged edition:

[1] ./ml.html
[2] ./ml.pdf
[3] ./ml.pdf
[4] ./tofte-tips.pdf
[5] http://www.cs.cmu.edu/~rwh/isml/book.pdf

S-expressions are great, but if you think carefully, you might notice that S-expressions work as a kind of high-level assembly language on top of which you craft your own data structures. ML, by contrast, offers a powerful tool for expressing data structures directly: the *algebraic data type*. Don't let the mathy name worry you; algebraic data types solve real programming problems:

- Algebraic data types enable you to define types that are *recursive*, like trees that contain subtrees of the same type as themselves.
- Algebraic data types enable you to define types by specifying *choices*, like a list that is made with either `'()` or `cons`.
- Algebraic data types give you easy access to the *parts* of data structures: instead of applying functions like `car` and `cdr`, you use *pattern matching* to name the parts.

Recursive types and definition by choices enable you to express representations that are very similar to what you might like to define in , and that also give you the benefits of polymorphic type inference that you get with ML. Types help you write good code, and they provide documentation that is checked by the compiler. Pattern matching eliminates the name-space clutter and cognitive overhead associated with choice-identifying and part-extracting functions like `null?`, `car`, `cdr`, `fst`, and `snd`. Using pattern matching, you make choices and extract parts by naming the choices and parts directly. The resulting code is shorter and more perspicuous than code without pattern matching. Algebraic data types and pattern matching are ubiquitous in the ML family and in languages derived from it, including Standard ML, OCaml, HAskell, Agda, Coq/Gallina, and Idris.

When you add algebraic data types to a language, you get a new species of value and type, a new expression form for looking at the values, and a new definition form for introducing the types and values.

The new species of value is called a *constructed value*. A constructed value is made by applying some *value constructor* to zero or more other values. In the syntax, however, we don't apply zero-argument value constructors; a zero-argument value constructor constitutes a value all by itself. As examples, in ML, `[]` is a value constructor for lists, and it expects no arguments, so it constitutes a constructed value all by itself. And cons (spelled `::`) is also a value constructor for lists, but it expects arguments, so to make a constructed value, cons must be applied to two other values: an element and a list.

A constructed value is observed or interrogated by a new syntactic form: the *case expression*. The case expression does *pattern matching*: a pattern can match a particular value constructor, and when it does, it can *name* each of the values to which the constructor was applied. For example, the pattern `(y :: ys)` matches any cons cell, and when it matches, it binds the name `y` to the car and `ys` to the cdr. A case expression includes a sequence of *choices*, each of which comprises a pattern and an associated right-hand side. One reason that programmers really like case expressions is that if your choices don't cover all possible cases, a compiler can alert you ().

Here's an example case expression, a definition of `null`:

```
fun null xs = (case xs of [] => true
                        | y :: ys => false)
```

And algebraic data type and its value constructors are created by a new form of definition, which uses the keyword `datatype`. In a datatype definition, you can name a type whatever you want. For example, you can define a new algebraic data type that is called `int`, which then hides the built-in `int`.

Key concept: Types and type inference

- Ullman, section 2.2
- Harper, section 2.2, especially 2.2.1, sections 2.3 and 2.4
- Tofte, section 13

The elements of ML

Expressions I: Basic expressions and their types

Ullman is the only resource that goes into simple syntax at any length. If you want more than you find below, search the Web for the "Gentle Introduction to ML."

- Ullman, sections 2.1 and 2.3
- Tofte, sections 1 to 5
- Examples from Ramsey, section 5.1: `find`, `bind`
- Examples from Ramsey, section 5.2: `equalatoms`, `equalpairs`
- Example from Ramsey, section 5.3: `duplicatename`

Expressions II: Minus signs

For reasons best known to Robin Milner, Standard ML does not use the unary minus sign used in every other known language (except APL). Instead, Standard ML uses the tilde as a minus sign, as in `~1` or `~(n+1)`. The tilde is in fact an ordinary function and may be treated as such.

Expressions III: Conditionals and short circuits

ML uses `if` e_1 `then` e_2 `else` e_3, as well as short-circuit *infix* `andalso`, `orelse` (**never** `and`).

- The abstract syntax is exactly like μScheme's `(if` e_1 e_2 e_3`)` `(&&` e_1 e_2`)`, and `(||` e_1 e_2`)`
- Ullman, sections 2.1.5 and 2.1.6
- Harper, section 2.3 (and desugaring in section 6.3)
- (This material is not covered by Tofte)

Data I: Tuples

In μML, tuples are ordinary value constructors of ordinary abstract data types (see below). But in Standard ML, they have special syntax:

- Ullman, section 2.4.1 (**not** section 2.4.2, which is an utter disaster, as noted below), plus 2.4.6 (tuple types)
- Tofte, sections 8, "Pairing and Tupling" (but don't use function $\#i$)
- Harper, section 5.1.1 (tuples)

Ullman pitfall: Jeff Ullman doesn't understand how to program with tuples. His section 2.4.2 should be torn out of your book and shredded (just kidding). The use of `#1` and `#3` violates all established customs for writing ML code—in part because `#1` and `#3` are not really functions, and they don't have types! The right way to extract an element from a tuple is by pattern matching, like this:[6]

```
fun fst (x, _) = x
fun snd (_, y) = y
```

Never write this:

```
fun bogus_first  p = #1 p    (* WRONG *)
fun bogus_second p = #2 p    (* WRONG *)
```

(For reasons I don't want to discuss, but will answer in class if asked, these versions don't even typecheck.) If your pair or tuple is not an argument to a function, use `val` to do the pattern matching:

```
val (x, y) = lookup_pair mumble
```

But usually you can include matching in ordinary `fun` matching.

You probably won't need to extract elements from a bigger tuple, but if you do, try

```
fun third (_, _, z) = z
```

Any uses of `#1`, `#2`, and their friends will result in point deductions on homework.

Data II: Lists

- Ullman, sections 2.4.3 to 2.4.5, plus 2.4.6 (list types)
- Tofte, section 4
- Harper, chapter 9 (which is short)

The most common mistake I see in list-related code is to write `xs = nil` (**wrong**). *Never* write this. Either use `null xs` (that's why `null` is in the initial basis) or use pattern matching.

You'll find most of your favorite list functions in the initial basis, either defined at top level or in the `List` module.

- Ullman, section 5.6
- Neither Tofte nor Harper says much about the initial basis
- Try `help "list";` at the Moscow ML prompt, then select `structure List`. ("Structure" is ML's name for "module".)

Data III: Constructed values and patterns that match them

Aside from the type system, the big new thing in ML is the system of "constructed values," which belong to *algebraic data types*.

- Ramsey, sections 8.1, 8.2.1, and 8.2.2 (page 559 includes examples of patterns that do and don't match)
- Ullman, sections 3.3 and 5.1
- Tofte, sections 8, 9, and 10
- Harper, sections 10.2, 10.3, and possibly 10.4

Tuples and records should also be considered constructed values. In μML, tuples and records are simulated with ordinary algebraic data types. In Standard ML, tuples and records have their own syntax and their own rules, but the ideas of construction and deconstruction (pattern matching) are the same.

Lists are constructed values that are supported with extra syntactic sugar for constructing and matching lists. Some useful list patterns include these patterns, to match lists of *exactly* 0, 1, 2, or 3 elements:

```
[]
[x]
[x, y]
[a, b, c]
```

You can also use the `::` (cons) constructor in patterns, where it appears infix. These patterns match lists of *at least* 0, 1, 2, or 3 elements:

```
xs
x :: xs
x1 :: x2 :: xs
a :: b :: c :: xs
```

Inexhaustive or redundant pattern matches

In any `case` expression or function definition, the patterns you provide must match *all* cases. If they don't, the pattern match is considered "inexhaustive" and is rejected by the compiler.

And in any `case` expression or function definition, every pattern you provide must match *some* case. If one pattern doesn't match

[6] The `fun` is function-definition syntax, like μScheme's `define`. It is described below.

any case, that pattern is considered "redundant," and the match is rejected by the compiler.

- Harper, section 6.4 (recommended) and page 105
- Ramsey, glossary page 385
- Ramsey, section 8.8.2
- Ullman, section 3.3.6 (inadvertent redundancy)

Types I: Introduction to types

Base types `int`, `string`, `char`, `real`, `bool`. *Type constructors* like `'a list` take a *type parameter* `'a`.

- Ullman, section 2.4.6
- Harper, chapter 2 (of which you already know section 2.2.2)
- Tofte, sections 1 and 13

Definitions I: Val bindings

ML's `val` bindings resemble those from μScheme, although μScheme's `val` corresponds to ML's `val rec`, which is μML's `val-rec`.

What we call a "definition" form is, to Standard ML, a "declaration" form.

- Ullman, section 2.3
- Tofte, section 6
- Harper, sections 3.2.2 and 3.3

Definitions II: Semicolons

It's a sad fact that if you're working interactively with an ML compiler, the compiler can't tell where a definition ends. You have to mark the end with a semicolon. But such a semicolon should **never** appear in your code. *Ullman's book is full of unnecessary semicolons, and you must learn to ignore him.* Emulate the style in Ramsey's book, which has no unnecessary semicolons. Use a semicolon only to sequence effects in imperative code.

Definitions III: Function definitions

As in μScheme and μML, functions can be defined using `lambda` with `val` or `val rec`. But it is more idiomatic to use `fun`, which is the analog of the `define` found in μScheme and μML.

- Ullman, sections 3.1 and 3.2
- Tofte, section 6
- Harper's chapter 4 is about functions. Although the chapter is long-winded and is more about the mathematical idea of functions than it is about how to program with functions, it is still useful—even though his examples include many unnecessary type annotations. The most helpful part of the chapter is probably Section 4.2, which contains several examples, especially at the end.
- Harper's chapter on recursive functions contains some more useful examples in sections 7.1, 7.2, and 7.4. The Fibonacci example in section 7.3 may also be useful. This chapter also includes substantial material on the mathematical justification for recursion and on the nature of inductive reasoning. You can learn ML without reading this material.

Definitions IV: Clausal (function) definitions

Standard ML's `fun` also provides *clausal definitions*, which in μML are written `define*`. These definitions look a lot like algebraic laws.

- Ramsey, `define*`, pages 575-576
- Ullman, section 3.3. Do *not* emulate Ullman's disgraceful placement of the vertical bar, and do not emulate his gratuitous semicolons.
- Tofte, section 11
- Harper, sections 6.2 and 6.4

Expressions IV: ML's `let`

ML's `let` most closely resembles Scheme's `let*`, but instead of a sequence of name/expression pairs, it uses a sequence of definition forms. The effect of `letrec` can be approximated by using a `fun` definition form with keyword `and`. Standard ML has nothing corresponding to Scheme's `let` form.

- Ullman, section 3.4
- Tofte, section 6
- Harper, section 3.4

Expressions V: ML's `lambda`

As noted above, ML's `lambda` expressions are written `fn (`x_1`, ...,` x_n`) =>` e.

- Ullman, section 5.1.3
- Tofte, section 7
- Harper, section 4.2

Expressions VI: Infix operators and precedence

The initial basis of Standard ML defines the following names as *infix identifiers*:

```
infix  7  * / div mod
infix  6  + - ^
infixr 5  :: @
infix  4  = <> > >= < <=
infix  3  := o
infix  0  before
```

The arithmetic you know, although you may not know that `/` is for floating point; `div` and `mod` are for integers. Here are the others:

- Operation `^` is string concatenation.
- Operations `::` and `@` are "cons" and "append" on lists.
- Operation `:=` is assignment to a mutable reference cell.
- Operation `o` is function composition.

- Operation `before` is used to add a side effect to a computation.

Function application has higher precedence than any infix operator. That means a function application underneath an infix operator should *never* be parenthesized!

Expressions VII: Infix operators as functions

The mechanism that ML uses for infix operators is very different from what you are used to from C and C++.

- The infix symbols are names, and they stand for ordinary functions.
- The names are set up to be used as infix operators by so-called *fixity declarations*. A fixity declaration for an infix name specifies precedence and associativity.
- When you want to *use* an infix name in a function application, you just write it as an infix operator.
- When you want to *refer* to the function as a value, you have to put the syntactic particle `op` in front of the name.

For details, see

- Ullman, section 5.4.4
- Harper, super-brief mention on page 78

Expressions VIII: Parentheses

It's easy to be confused about when you need parentheses. Here's a checklist to tell you when to use parentheses around an expression or a pattern:

1. Is it an argument to a (possibly Curried) function, and if so, is it more than a single token?
2. Is it an infix expression that has to be parenthesized because the precedence of another infix operator would do the wrong thing otherwise?
3. Are you forming a tuple?
4. Are you parenthesizing an expression involving `fn`, `case`, or `handle`?
5. Are you parenthesizing an infix operator marked with `op`?

If the answer to any of these questions is yes, use parentheses. Otherwise, you almost certainly don't need them—so get rid of them!

Especially,

- *Never parenthesize the condition in an `if` expression.* Such parentheses brand you as an unreconstructed C programmer.

- *Never put parentheses around a single token.* For example, never write something like ~~`(0)`~~ or ~~`(xs)`~~, as in ~~`double(0)`~~ or ~~`length(xs)`~~. Write `double 0` or `length xs` instead. (Ullman breaks this rule all the time. We hates it!)

Types II: Polymorphic functions

In ML, as in Scheme, you can write polymorphic functions simply by writing functions that are agnostic about some aspects of their arguments. The difference is that in ML, type system *infers* at compile time the knowledge that the function is polymorphic. You can probably learn all you need by feeding some of your μScheme code to the nano-ML (`nml`) or μML (`uml`) interpreters; you can identify a polymorphic function by the `forall` in the type. (Most unfortunately, Standard ML omits the `forall` from the type. You're supposed to imagine it.)

- For an introduction, Ullman, section 5.3
- For a deep look, including some technical details, Harper, chapter 8
- (Not covered in Tofte)

Curried functions

What we call a "partially applied" function, Ullman calls "partially instantiated." (He's thinking of a substitution model. Bad Ullman.) There are no new concepts here, but the concrete syntax is radically different from what you're used to in μScheme.

- Ullman, section 5.5
- Tofte, section 7
- Harper, section 11.3 (as usual, a very technical approach)

Exceptions

ML exceptions behave a lot like the Hanson `Except_T` you may have seen in COMP 40, and somewhat like exceptions in C++ or Java.

- Ullman, section 5.2
- Tofte, section 16
- Harper, opening of section 2.2 (example of an "effect")
- Harper, chapter 12 (this is a long chapter, but after you read through the first example in section 12.2, you'll know enough to get started)

Types III: Type abbreviations

Type abbreviations are a leading cause of confusion for beginning ML programmers. A type abbreviation, which begins with the keyword `type`, creates a new name for an old type. But in its error messages, the compiler may not honor the abbreviation—it may insist on referring to the old type instead.

If you're asked to "define a type," you have to decide if you want a type *abbreviation* with `type`, or whether you want a "datatype definition" with `datatype`. Because both have similar effects on the type environment, both count as "define a type."

- Ullman, section 6.1
- Tofte, section 14
- Harper, section 3.2.1

A type abbreviation can take *type parameters*. A type parameter is identified by a name that begins with a tick mark, and the names traditionally used are 'a, 'b, 'c, and so on. On the topic of type abbreviations with type parameters, both Harper and Tofte are unaccountably silent. Ullman at least gives a sketch in section 6.1.3. For an example, I recommend the definition of type env in Ramsey, chunk 359.

Data IV: Datatype definitions

A datatype definition creates a brand new type, which is distinct from any other type—even one that has the same name. If you create multiple types with the same name, you will become confused.

- Ullman, section 6.2
- Ramsey, section 5.2, example datatype definitions, in Standard ML, for the μScheme interpreter
- Ramsey, sections 8.1 and 8.2.3 (sensible only after you already understand what is going on with the types)
- Tofte, section 15
- Harper, chapter 10 (and for something with a type parameter, section 10.3)
- Ramsey, page 548, the example about option@{2}, for understanding about multiple types with the same name

Type pitfall I: "Equality types" (two tick marks)

Some type variables begin with two tick marks, as in

```
- fun isNothing v = (v = NONE);
> val ''a isNothing = fn : ''a option -> bool
```

The type variable ''a can be instantiated only by an "equality type." A type variable like this, called an "equality type variable," is *almost always a programming mistake.* The code above should have used pattern matching (compare Ullman, page 215).

For details:

- Ullman, section 5.3.4 on page 150
- Ullman, box on page 209
- MLton documentation: the rules that say what types can be compared with equality[7], and explanation of the primitive = function[8].
- (Neither Harper nor Tofte cover equality types.)

Primitive types exn and real (floating-point numbers) are *not* equality types.

Type pitfall II: Value polymorphism

In Standard ML, no *function* is ever trusted to create a polymorphic value. Only value constructors can create polymorphic values. If you violate this so-called "value restriction," you get a message like this one:

[7] http://mlton.org/EqualityType
[8] http://mlton.org/PolymorphicEquality

```
- val empty = rev [];
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier
> val empty = [] : 'a list
```

For details:

- Harper, section 8.2 on page 70
- Ullman, section 5.3.1 on page 145
- (Not covered in Ramsey or Tofte)

Basis I: The `option` type

Let's suppose you want to represent a value, except the value might not actually be known. For example, I could represent a grade on a homework by an integer, except if a grade hasn't been submitted. Or the contents of a square on a chessboard is a piece, except the square might be empty. This problem comes up so often that the initial basis for ML has a special type constructor called option, which lets you handle it. The definition of option is

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

and it is *already defined* when you start the interactive system. You need not and should not define it yourself. As in a type abbreviation, the type parameter 'a stands for an unknown type—you can substitute *any* type for the type variable 'a.

Read

- Ramsey, pages 547 and 548
- Ullman, section 4.1 (pages 111–113) and page 208.
- (Not covered in Tofte)
- Harper, section 10.2 (in passing)

Here are some more examples:

```
- datatype chesspiece = K | Q | R | N | B | P
- type square = chesspiece option
- val empty : square = NONE
- val lower_left : square = SOME R
- fun play piece = SOME piece : square;
> val play = fn : chesspiece -
> chesspiece option

- SOME true;
> val it = SOME true : bool option
- SOME 37;
> val it = SOME 37 : int option

- SOME "fish" = SOME "fowl";
> val it = false : bool
- SOME "fish" = NONE;
> val it = false : bool
- "fish" = NONE;
! Toplevel input:
! "fish" = NONE;
!          ^^^^
```

```
! Type clash: expression of type
!   'a option
! cannot be made to have type
!   string
```

Data V: Record types, values, expressions, and patterns

In addition to tuples, Standard ML has records with named fields. A record is notated by a set of key-value pairs, separated by commas, and enclosed in curly braces. The order of the pairs doesn't matter. Unfortunately, records come with some special rules and special syntax that can cause pain for beginners.

- By far the most useful introduction to records is Harper's section 5.2, which has the longest explanation and the best examples. Harper's section 5.3 mixes some useful examples with some dangerous examples. You'll be all right as long as you heed Harper's advice that *"Use of the sharp notation is strongly discouraged."* In COMP 105, **the sharp notation is forbidden**.
- Ullman's sections 7.1.1 and 7.1.5 are reliable, as are parts of section 7.1.4.
- *Ullman pitfall:* Disregard Ullman's section 7.1.2. The #*name* syntax, like the $\#n$ syntax for tuples, is worse than useless—it actively makes it more difficult to write your code. Use pattern matching instead.
- *Avoid* using the ellipsis in record patterns. To use it successfully, you must have a deep understanding of type inference and of Standard ML's peculiar approach to record types.
- Tofte mentions records briefly in section 8. That pitfall is back: you must *avoid* what Tofte calles the #*lab* syntax—#*lab* is not a function, and it will trip you up.

What's wrong with the sharp notation? In brief, `#`*name* is a piece of syntax—it's not a function, and it doesn't have a unique type. What's wrong with ellipsis patterns? Same thing: an ellipsis pattern doesn't have a unique type.

Basis II: Access to functions defined in modules

Standard ML includes a sophisticated module language—one of the most expressive module languages ever designed. But at least to start, you'll use modules in a very stylized way: by selecting components from modules in the initial basis. Such selection uses "dot notation," with the name of the module followed by the name of a component. Examples include `Int.toString` and `List.filter`.

- You can see some examples in Tofte, section 19
- There's another example in Ullman, section 8.2.3
- *Ullman pitfall:* Avoid the `open` technique described in Ullman's section 8.2.4
- I'm not seeing any place where Harper explains this notation.

In section 8.2.4, Ullman shows that you can get access to the contents of a module by *opening* the module, as in `open TextIO`. ***Never*** **do this**—it is bad enough to open structures in the standard basis, but if you open other structures, your code will be hopelessly difficult to maintain. Instead, *abbreviate* structure names as needed. For example, after `structure T = TextIO`, you can use `T.openIn`, etc., without (much) danger of confusion.

Basis III: Getting to know the Standard Basis

The initial basis of Standard ML is called the "Standard Basis," or sometimes the "Standard Basis Library." Get to know it, and use it when you can.

Modules you can learn easily include `List`, `Option`, `ListPair`, `Vector`, and `Array`. You may also have some use for `TextIO`.

Moscow ML ships with an extended version of the standard basis library. Tell Moscow ML `help "lib";`, and you'll see what's there.

```
ledit mosml -P full
```

as your interactive top-level loop, it will automatically load almost everything you might want from the standard basis.

Basis IV: Vectors

Although Ullman describes the mutable `Array` structure in Chapter 7, he doesn't cover the immutable `Vector` structure except for a couple of pages deep in Chapter 9. Like an array, a vector offers constant-time access to an array of elements, but a vector is not mutable. Because of its immutability, `Vector` is often preferred. It is especially flexible when initialized with `Vector.tabulate`.

The functions to start with include `Vector.tabulate`, `Vector.fromList`, `Vector.length`, and `Vector.sub`. The `Vector` structure also includes variations on `app`, `map`, `foldl`, `foldr`, `find`, `exists`, and `all`.

Unit Testing

Standard ML does not come with built-in unit-testing support. For COMP 105, we have created a special module called `Unit`, which exports functions that can be used for unit testing. These and related functions are shown in the table.

Example unit tests

You write a unit test in the form of a `val` definition with the empty tuple as the pattern. Here are two examples, only one of which passes:

```
val () =
    Unit.checkExpectWith Int.toString "2 is third"
    (fn () => List.nth ([1, 2, 3], 2))
    3

val () =      (* this test fails *)
    Unit.checkExpectWith Bool.toString "2 is false"
    (fn () => List.nth ([true, false, true], 2))
    false
```

The difference between this module and the `check-expect` you are used to is that `check-expect` takes just an expression, but `Unit.checkExpectWith` requires that expression be wrapped in a function of no arguments. So if you are expecting expression e to evaluated to value v, you must write

`Unit.checkExpectWith` ... `(fn () =>` e`)` v

Functions `Unit.checkAssert` and `Unit.checkExnWith` work the same way. Here are two more examples, both of which pass:

```
val () =
  Unit.checkAssert "has positive"
  (fn () => List.exists (fn n => n > 0) [~1, 0, 1])

val () =
  Unit.checkExnWith (Unit.listString Int.toString)
   "3rd element of empty list"
  (fn () => List.nth ([], 3))
```

Bureaucracy of getting our module (Moscow ML)

To use the unit tests, you must have a compiled version of the `Unit` module. You may use ours, or you may compile it yourself.

- To use ours, tell Moscow where to look by giving it the command-line option `-I /comp/105/lib`. This option works with both `mosml` and `mosmlc`.
- To compile it yourself, copy files `Unit.sig` and `Unit.sml` from `/comp/105/lib`, and compile them at your shell prompt:

  ```
  $ mosmlc -c Unit.sig
  $ mosmlc -c Unit.sml
  ```

  These commands should produce files `Unit.ui` and `Unit.uo`.

The batch compiler `mosmlc` should now be able to compile your code using `Unit`:

```
$ mosmlc -toplevel -I /comp/105/lib -
c warmup.sml
```

At this stage, you can load your code into the interactive system:

```
$ mosml -I /comp/105/lib -P full
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter `quit();' to quit.
- load "warmup";
> val it = () : unit
- Unit.report();
Both internal Unit tests passed.
> val it = () : unit
```

If you prefer to bring your source code directly into Moscow ML, without compiling first, you can call ML's use function, but you will need to load the `Unit` module first:

```
$ mosml -P full -I /comp/105/lib
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter `quit();' to quit.
- load "Unit";
> val it = () : unit
- use "warmup.sml";
[opening file "warmup.sml"]
...
[closing file "warmup.sml"]
> val it = () : unit
-
```

If you forget this step, you'll get an error message:

```
nr@homedog /tmp> mosml -P full -I /comp/105/lib
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter `quit();' to quit.
- use "warmup.sml";
[opening file "warmup.sml"]
File "warmup.sml", line 2, characters 6-22:
!       Unit.checkAssert "has positive"
!       ^^^^^^^^^^^^^^^^
! Cannot access unit Unit before it has been loaded.
[closing file "warmup.sml"]
-
```

Just `load "Unit";` and you'll be on your way.

Bureacracy of getting our module (MLton)

Compiling multiple modules in MLton is a hassle: you have to have an "MLB" file that lists all the modules you want, in dependency order, plus the version of the basis you want. This is a hassle. To get the unit module, you include `/comp/105/lib/unit.mlb` in your own MLB file. Where you need unit tests for the homework, we'll provide an MLB file you can use.

Documentation of all the functions

Every function in the `Unit` module is declared with a name and a type. These declaration forms, which are shown below, are part of ML's modules system. Each declaration is followed by some informal English. To refer to the function from *outside* the `Unit` module, in your code, you need "Unit dot function-name."

Testing functions

The closest analog to `check-expect` is the function `Unit.checkExpectWith`:

```
val checkExpectWith : (''a -> string) ->
                      string ->
                      (unit -> ''a) ->
                      ''a ->
                      unit
```

Calling `checkExpectWith show name test result` evaluates `test()`, applying `test` to the empty tuple. If the result equals `result`, the test passes; otherwise it fails. If the test fails, `name` is used to identify the failing test.

The closest analog to `check-assert` is `Unit.checkAssert`:

```
val checkAssert : string -> (unit -> bool) -
> unit
```

Calling `checkAssert name test` evaluates `test()`, and if the result is `true`, the test passes; otherwise it fails. If the test fails, `name` is used to identify the failing test.

The closest analog to `check-error` is `Unit.checkExnWith`:

```
val checkExnWith : ('a -> string) ->
                   string ->
                   (unit -> 'a) ->
                   unit
```

Calling `checkExn show name test` evaluates `test()`, and if the evaluation raises an exception, the test passes. If `test()` returns a value, the test fails. If the test fails, `name` is used to identify the failing test.

The three functions above will serve you in most situations, but if you want to be sure you know what exception is raised, use this one:

```
val checkExnSatisfiesWith :
                     ('a -> string) ->
                     string ->
                     (unit -> 'a) ->
                     (string * (exn -> bool)) ->
                     unit
```

Calling

```
checkExnSatisfiesWith show name test (ename, pred)
```

evaluates `test()`, and if the evaluation raises an exception `e`, the test passes, provided `pred e` is true. If `test()` raises a non-satisfying exception, or if `test()` returns a value, the test fails, complaining that the exception named `ename` was expected. If the test fails, `name` is used to identify the failing test.

Reporting functions

Use these functions to confirm that all your unit tests pass.

```
val report : unit -> unit
```

If any tests were run, report, on standard output, how many tests were run in total and how many passed.

```
val reportWhenFailures : unit -> unit
```

If any tests failed, report, on standard output, how many tests were run in total and how many passed.

String-conversion functions

Use these functions to create string-conversion functions to pass to `checkExpectWith` and `checkExnWith`.

```
val listString : ('a -> string) ->
                 ('a list -> string)
```

If `showA` converts a value in~A to a string, then `listString showA` returns a function converts a value in "list of A" to a string.

```
val pairString : ('a -> string) ->
                 ('b -> string) ->
                 ('a * 'b -> string)
```

If `showA` converts a value in~A to a string, and `showB` converts a value in~B to a string, `pairString showA showB` returns a function that converts a pair of type $A \times B$ to a string.

```
val showNothing : 'a -> string
```

If you need to convert a value of type A to a string, but you don't have a converter—and you don't want to write one—you can use `showNothing` with any type A. Function `showNothing` converts every value to the string `"a value"`.

An example with a user-defined type

Here is an example of using `checkExpectWith`:

```
datatype color = LAVENDER | BURGUNDY | VIRIDIAN
fun colorString LAVENDER = "LAVENDER"
  | colorString BURGUNDY = "BURGUNDY"
  | colorString VIRIDIAN = "VIRIDIAN"
val favorites = [BURGUNDY, LAVENDER, VIRIDIAN]
val () = (* this test fails *)
  Unit.checkExpectWith colorString
  "least favorite color"
  (fn () => List.last favorites)
  LAVENDER
```

This test fails with this error message:

> In test 'least favorite color', expected value LAVENDER but got VIRIDIAN

Here is an example of using `checkExnWith`:

```
val () = (* this test passes *)
  Unit.checkExnWith colorString
  "last of empty string"
  (fn () => List.last [])
```

Table 1: Functions useful for unit testing

Function	Description
`Unit.checkExpectWith`	An adequate substitute for `check-expect`. Requires a function to convert result to a string. And because an ML function does not know where in the source coded its definition is, `checkExpectWith` also requires a test name, which should uniquely identify the test.
`Unit.checkAssert`	An adequate substitute for `check-assert`. Requires a test name.
`Unit.checkExnWith`	An adequate substitute for `check-error`. Used when you expect evaluating an expression to raise an exception. Also requires a name and a string-conversion function.
`Unit.checkExnSatisfiesWith`	A refined version of `Unit.checkExnWith` that enables you to control *which* exceptions are considered to pass the test.
`Unit.report`	Prints a report of testing outcomes.
`Unit.reportWhenFailures`	Prints a report *only* if some test fails.
`Int.toString`	Convert an integer to a string. Sometimes useful for passing to `checkExpectWith`, `checkExnWith`, or `checkExnSatisfiesWith`.
`Bool.toString`	Convert a Boolean to a string.
`Char.toString`	Convert a character to a string.
`Unit.listString`	A higher-order function. Given a function that converts a value to a string, returns a function that converts a list of values to a string. Useful for passing to `checkExpectWith` and friends.
`Unit.pairString`	Another higher-order function. Given two string-conversion functions, returns a function that converts a pair of values to a string.

ML Modules

General information on modules

If you want to review what you heard in class,

- The Code & Co. blog[9] has a nice post on ML modules[10], which covers all three major components: structures, signatures, functors
- Ullman, in Section 8.1, gives the same sort of guff you heard in lecture about information hiding. It's a lightweight, non-technical overview.
- Tofte, Section 21 provides the barest possible introduction.

Signatures, Structures, and Ascription

The foundations are interfaces, implementations, and matching one to the other—or in the gratuitously mathematical lingo of Standard ML: signatures, structures, and ascription.

- Harper, although a bit long-winded, provides the most complete and detailed treatment of these topics. I recommend you start there. Chapter 18 presents signatures and structures—not just the basics, but a fair number of details.

  A detailed explanation of what it means for a signature to *match* a structure is there in Chapter 19. The presentation is very careful and well thought out, with examples.

  Chapter 20 explains ascription, again very well, with examples. Focus your attention on section 20.2 (opaque ascription). You can skip section 20.3 (transparent ascription)—while Harper does his honorable best to make a case, the sad truth is that there is not really a compelling case for transparent ascription. And **transparent ascription is not acceptable in COMP 105**.

  If Harper is too detailed or too mathematical for your taste, try one of the other sources.

- Most of these topics are addressed by Ullman in Section 8.2, but there are quite a few pitfalls to avoid:
  - Section 8.2.2 describes ascription, which Ullman calls "restriction." Unfortunately, Ullman uses the legacy "transparent" ascription style with a bare colon. **This style is not acceptable in COMP 105.** Use proper "opaque" ascription—the one with the "beak" operator (`:>`). Ullman says a little more about opaque ascription in section 8.5.5.
  - Section 8.2.4 describes `open`. I won't waste space here detailing all the reasons `open` is terrible; just be aware that in COMP 105, **use of `open` is grounds for No Credit** on the modules assignment.

[9] https://jozefg.bitbucket.io/about.html
[10] https://jozefg.bitbucket.io/posts/2015-01-08-modules.html

- Tofte covers the basics in sections 22 to 27. Again, the "transparent" constraints in section 25 are the ones to avoid, and the "opaque" constraints in section 26 are the ones to use.

Signature refinement or specialization

Quite often, we need to specify the identity of an abstract type after the fact—usually when writing the result signature of a functor. This is done using the `where type` syntax.

- Ullman completely omits signature refinement using `where type`. Without this tool, opaque ascription is crippled. You'll need to read about it in Tofte or Harper.
- Tofte calls the technique "type realization," and you'll find it described, with a careful example, in section 27.
- Harper describes signature specialization in section 18.1.2.

Modules can nest

An excellent aspect of Standard ML's design is that structures can contain other structures. That's really all you need to know, but if you want to understand *why* this feature is valuable, there are plenty of good examples in Harper, chapter 21.

Functors

Paraphrased from Harper: an ML *functor* is a function that operates on the module level. Its *formal* parameters, if any, are specified by a *sequence of declarations*, and its *actual* parameters are given by a *sequence of definitions*. Its result is a structure. Functors enable a variety of styles of programming, but in 105, we focus on the functor as a mechanism for code reuse.

- The basics are found in Harper, section 23.1.

There is a dirty secret: there is a *second* mechanism for specifying parameter(s) to a functor, which is to package everything up in a single structure. I much prefer Harper's "sequence of declarations/definitions" view, and I don't know why we are stuck with two mechanisms. But you'll see the second mechanisms from time to time.

- Ullman, in Section 8.3, spends a lot of ink mucking about with various alternative ways of defining functors' parameters. Ullman manages to take a simple thing and make it seem complicated. On this subject, I recommend avoiding Ullman entirely.
- Tofte, Section 28, is saner than Ullman, but he doesn't handle the argument issue as well as Harper does. The form that Harper prefers (and that we prefer also) is treated by Tofte as "an alternative form." You'll notice, however, that this form is what Tofte uses in his example.

Sharing constraints

Don't ask. In COMP 105 we do our best to avoid putting you in situations where you need `sharing` constraints. But if you trip over them, Harper's Chapter 22 is the best source. Follow up with sections 23.2 and 23.3, in which Harper explains when sharing constraints are needed and why they are better than the alternatives. They are also mentioned by Tofte in section 28.

Abstract data types

Harper completely understands the importance of data abstraction and the use of ML modules to enforce it.

- Section 20.2 introduces opaque ascription by explaining its role in data abstraction.
- Chapter 32 presents a complete, sophisticated example of data abstraction using the familiar abstraction of a dictionary. The chapter shows implementations using two different forms of binary search tree.