Learning Standard ML

COMP 105

Contents

Key concepts: algebraic data types, case expressions, and pattern matching 1

Key concept: types and type inference 1

Translating µ/Scheme and µ/ML to ML 1

The elements of ML

Expressions I: Basic expressions and their types 2
Expressions II: Minus signs 2
Expressions III: conditionals and short circuits 2
Data I: Tuples 2
Data II: Lists 2
Data III: Constructed values and patterns that match them: Inexhaustive or redundant pattern matches 3
Types I: Introduction to types 3
Definitions I: Val bindings 3
Definitions II: Semicolons 3
Definitions III: Function definitions 3
Definitions IV: Clausal (function) definitions 3
Expressions IV: ML’s let 4
Expressions V: ML’s lambda 4
Expressions VI: Infix operators and precedence 4
Expressions VII: Infix operators as functions 4
Expressions VIII: Parentheses 4
Types II: Polymorphic functions 5
Curried functions 5
Exceptions 5
Types III: Type abbreviations 5
Data IV: Datatype definitions 5
Basis I: The option type 5
Data V: Record types, values, expressions, and patterns 6
Basis II: Access to functions defined in modules 6
Basis III: Getting to know the Standard Basis 6
Basis IV: Vectors 6

This guide is available both in HTML and PDF.

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: 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)" (http://www.cs.cmu.edu/~rwh/isml/book.pdf)
• Bob Harper’s draft Programming in Standard ML

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

• Ramsey, opening two pages of Chapter 10 (“value constructors”)

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

Translating µ/Scheme and µ/ML to ML

You are welcome to start out by writing µ/Scheme and translating it to Standard ML. But you will have to learn pattern matching. You are also welcome to learn your pattern matching in µ/ML, which closely resembles µ/Scheme, then translate that to Standard ML.


/ml.html
/ml.pdf
/tofte-tips.pdf
To help in translation, pages 858 to 861 of Ramsey contain some tables of equivalent syntax. Unfortunately, the tables could not be completed before press time, and some key equivalences are missing:

• \(\mu\text{ML}'s \ (\text{lambda} \ (x_1 \cdots x_n) \ e) \text{ is equivalent to Standard ML's } (\text{fn} \ (x_1, \ldots, x_n) => e). \) The parentheses are not always needed, but you would be wise to include them.

• \(\mu\text{ML}'s \ (\text{case} \ e \ (\lfloor p_1 e_1 \rfloor \cdots \lfloor p_n e_n \rfloor)) \text{ is equivalent to Standard ML's } (\text{case } e \text{ of } p_1 => e_1 \mid \cdots \mid p_n => e_n). \) Again, the parentheses are not always needed, but you would be wise to include them.

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 \(\sim 1\) or \(\sim (n+1)\). The tilde is in fact an ordinary function and may be treated as such.

Expressions III: conditionals and short circuits

ML uses \(\text{if } e_1 \text{ then } e_2 \text{ else } e_3\), as well as short-circuit \(\text{infix andalso, orelse (never and).}\)

• The abstract syntax is exactly like \(\mu\text{Scheme's } (\text{if } e_1 e_2 e_3) \) (\(\&\&\ e_1 e_2\)), and \((|| e_1 e_2)\)
• Ullman, sections 2.1.5 and 2.1.6
• Tofte, sections 2.3.4 and 2.3.5
• (This material is not covered by Tofte)

Data I: Tuples

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

• 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)

\textit{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:

\begin{verbatim}
fun fst (x, _) = x
fun snd (_, y) = y
\end{verbatim}

Never write this:

\begin{verbatim}
fun bogus_first p = #1 p (\* WRONG *)
fun bogus_second p = #2 p (\* WRONG *)
\end{verbatim}

(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 \texttt{val} to do the pattern matching:

\begin{verbatim}
val (x, y) = lookup_pair mumble
\end{verbatim}

But usually you can include matching in ordinary \texttt{fun} matching.

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

\begin{verbatim}
fun third (_, _, z) = z
\end{verbatim}

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 \(xs = \text{nil (wrong)}.\) \textit{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 \texttt{List} module.

\begin{verbatim}
fun first (head, tail) = head
fun second (head, tail) = tail
\end{verbatim}

6The \texttt{fun} is function-definition syntax, like \(\mu\text{Scheme's } \text{define. It is described below.}\)
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 10.1, 10.2.1, and 10.2.2 (page 749 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 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 371
- Ramsey, section 10.8.2
- Ullman, section 3.3.6 (inadvertent redundancy)
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 771–772
- 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 (x₁, ..., xₙ) => 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 (x), 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 350b.

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 10.1 and 10.2.3 (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 744, the example about option@{2}, for understanding about multiple types with the same name

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

```ml
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 743 and 744
- 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 calls 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.

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`. 