A note about books

Ullman is easy to digest

Ullman costs money but saves time

Ullman is clueless about good style

Suggestion:
• Learn the syntax from Ullman
• Learn style from Ramsey, Harper, and Tofte

Details in course guide Learning Standard ML
Define algebraic data types for $SX_1$ and $SX_2$, where

$$SX_1 = ATOM \cup LIST(SX_1)$$

$$SX_2 = ATOM \cup \{(\text{cons } v_1 \ v_2) \mid v_1 \in SX_2, v_2 \in SX_2\}$$

(take $ATOM$, with ML type $\text{atom}$ as given)
Exercise answers

```haskell
datatype sx1 = ATOM1 of atom |
              LIST1 of sx1 list

datatype sx2 = ATOM2 of atom |
              PAIR2 of sx2 * sx2
```
Eliminate values of algebraic types

New language construct `case` (an expression)

```plaintext
fun length xs =
    case xs
    of [] => 0
    | (x::xs) => 1 + length xs
```
At top level, ‘fun‘ better than ‘case‘

When possible, write

fun length [] = 0
  | length (x::xs) = 1 + length xs
‘case‘ works for any datatype

```haskell
fun toStr t =
  case t
  of Leaf => "Leaf"
      | Node(v,left,right) => "Node"
```

But often pattern matching is better style:

```haskell
fun toStr' Leaf = "Leaf"
  | toStr' (Node (v,left,right)) = "Node"
```
## Types and their ML constructs

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrow</td>
<td>Lambda ($\lambda$)</td>
<td>Application</td>
</tr>
<tr>
<td>algebraic</td>
<td>Apply constructor ($e_1, \ldots, e_n$)</td>
<td>Pattern match!</td>
</tr>
<tr>
<td>tuple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exception handling in action

```plaintext
loop (evaldef (reader (), rho, echo))
handle EOF => finish ()
| Div => continue "Division by zero"
| Overflow => continue "Arith overflow"
| RuntimeException msg => continue ("error: " ^ msg)
| IO.Io {name, ...} => continue ("I/O error: " ^ name)
| SyntaxError msg => continue ("error: " ^ msg)
| NotFound n => continue (n ^ "not found")
```
ML Traps and pitfalls
Order of clauses matters

fun take n (x::xs) = x :: take (n-1) xs
  | take 0 xs       = []
  | take n []       = []

(* what goes wrong? *)
Gotcha — overloading

- fun plus x y = x + y;
> val plus = fn : int -> int -> int
- fun plus x y = x + y : real;
> val plus = fn : real -> real -> real
Gotcha — equality types

- `(fn (x, y) => x = y);`
- `val it = fn : ∀ ′ ′a . ′ ′a * ′ ′a -> bool`

Tyvar ′ ′a is “equality type variable”:
- values must “admit equality”
- (functions don’t admit equality)
Gotcha — parentheses

Put parentheses around anything with case, handle, fn

Function application has higher precedence than any infix operator
Syntactic sugar for lists

- 1 :: 2 :: 3 :: 4 :: nil; (* :: associates to the right *)
> val it = [1, 2, 3, 4] : int list

- "the" :: "ML" :: "follies" :: [];
> val it = ["the", "ML", "follies"] : string list

> concat it;
val it = "theMLfollies" : string
ML from 10,000 feet
The value environment

Names bound to immutable values
  Immutable `ref` and `array` values point to mutable locations

ML has no binding-changing assignment

Definitions add new bindings (hide old ones):
  `val pattern = exp`
  `val rec pattern = exp`
  `fun ident patterns = exp`
  `datatype ... = ...`
Nesting environments

At top level, definitions

Definitions contain expressions:

\[\text{def ::= val pattern = exp}\]

Expressions contain definitions:

\[\text{exp ::= let defs in exp end}\]

Sequence of \textit{defs} has let-star semantics
What is a pattern?

\[
\text{pattern ::= variable} \\
| \text{wildcard} \\
| \text{value-constructor } [\text{pattern}] \\
| \text{tuple-pattern} \\
| \text{record-pattern} \\
| \text{integer-literal} \\
| \text{list-pattern}
\]

Design bug: no lexical distinction between
- VALUE CONSTRUCTORS
- variables

Workaround: programming convention
Function peculiarities: 1 argument

Each function takes 1 argument, returns 1 result

For “multiple arguments,” use tuples!

fun factorial n = 
  let fun f (i, prod) =
    if i > n then prod else f (i+1, i*prod)
  in  f (1, 1)
  end

fun factorial n = (* you can also Curry *)
  let fun f i prod =
    if i > n then prod else f (i+1) (i*prod)
  in  f 1 1
  end
Mutual recursion

Let-star semantics will not do.

Use \textbf{and} (different from \texttt{andalso})!

\begin{verbatim}
fun a x = ... b (x-1) ...
and b y = ... a (y-1) ...
\end{verbatim}
Syntax of ML types

Abstract syntax for types:

\[ t_y \Rightarrow \text{TYVAR of string} \quad \text{type variable} \]

\[ | \text{TYCON of string * ty list} \quad \text{apply type constructor} \]

Each tycon takes fixed number of arguments.

nullary \ int, bool, string, ...

unary \ list, option, ...

binary \ \rightarrow

n-ary \ tuples (infix *)
Syntax of ML types

Concrete syntax is baroque:

\[
\begin{align*}
  ty & \Rightarrow tyvar & \text{type variable} \\
  | & tycon & \text{(nullary) type constructor} \\
  | & ty \ tycon & \text{(unary) type constructor} \\
  | & (ty, \ldots, ty) \ tycon & \text{(n-ary) type constructor} \\
  | & ty * \ldots * ty & \text{tuple type} \\
  | & ty \rightarrow ty & \text{arrow (function) type} \\
  | & (ty) & \\
  tyvar & \Rightarrow \text{'identifier} \quad \text{'}a, \text{'b}, \text{'c}, \ldots \\
  tycon & \Rightarrow \text{identifier} \quad \text{list}, \text{int}, \text{bool}, \ldots
\end{align*}
\]
Polymorphic types

Abstract syntax of type scheme $\sigma$:

$\sigma \Rightarrow \text{FORALL of tyvar list} \ast \text{ty}$

Bad decision: $\forall$ left out of concrete syntax

$$(\text{fn } (f, g) \Rightarrow \text{fn } x \Rightarrow f \ (g \ x))$$

: $\forall \ 'a, 'b, 'c$ .

$$( 'a \rightarrow 'b ) \ast ( 'c \rightarrow 'a ) \rightarrow ( 'c \rightarrow 'b )$$

Key idea: substitute for quantified type variables
Old and new friends

\[ \text{op o} \quad : \quad \forall \ 'a, 'b, 'c . \]
\[ (\ 'a \to 'b) \times (\ 'c \to 'a) \to 'c \to 'b \]

\[ \text{length} \quad : \quad \forall 'a . \ 'a \text{ list} \to \text{int} \]

\[ \text{map} \quad : \quad \forall 'a, 'b . \]
\[ (\ 'a \to 'b) \to (\ 'a \text{ list} \to 'b \text{ list}) \]

\[ \text{curry} \quad : \quad \forall 'a, 'b, 'c . \]
\[ (\ 'a \times 'b \to 'c) \to 'a \to 'b \to 'c \]

\[ \text{id} \quad : \quad \forall 'a . \ 'a \to 'a \]