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, & Tofte

Details in course guide Learning Standard ML
Datatype definitions

datatype suit = HEARTS | DIAMONDS | CLUBS | SPADES

datatype 'a list = nil (* copy me NOT! *)
| op :: of 'a * 'a list

datatype 'a heap = EHEAP
| HEAP of 'a * 'a heap * 'a heap

type suit val HEARTS : suit, ...
type 'a list val nil : forall 'a . 'a list
val op :: : forall 'a .
| 'a * 'a list -> 'a list

type 'a heap
val EHEAP: forall 'a.
val HEAP : forall 'a.'a * 'a heap * 'a heap -> 'a heap
Eliminate values of algebraic types

New language construct `case` (an expression)

```plaintext
fun length xs =
  case xs
  of [] => 0
  | (x::xs) => 1 + length xs
```

Clausal definition is preferred
(sugar for `val rec, fn, case`)
case works for any datatype

fun toStr t =
    case t
        of EHEAP => "empty heap"
        | HEAP (v, left, right) =>
            "nonempty heap"

But often a clausal definition is better style:

fun toStr' EHEAP = "empty heap"
    | toStr' (HEAP (v,left,right)) =
        "nonempty heap"
Other constructed data: Tuples

Always only one way to form

- Expressions \((e_1, e_2, \ldots, e_n)\)
- Patterns \((p_1, p_2, \ldots, p_n)\)

Example:

```haskell
let val (left, right) = splitList xs
in  if abs (length left - length right) < 1
    then
      NONE
    else
      SOME "not nearly equal"
end
```
Frequently overlooked

An algebraic data type is a collection of alternatives

Don’t forget:
  • Each alternative must have a name

The thing named is the value constructor

(Also called “datatype constructor”)
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 $atom$ as given)
Wait for it ...
Exercise answers

datatype sx1 = ATOM1 of atom
| LIST1 of sx1 list

datatype sx2 = ATOM2 of atom
| PAIR2 of sx2 * sx2
Exception handling in action

loop (evaldef (reader (), rho, echo))

handle EOF             => finish ()
    | Div            => continue "Division by zero"
    | Overflow       => continue "Arith overflow"
    | RuntimeError 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?

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

\[ ty \Rightarrow \text{TYVAR of string type variable} \]
\[ \mid \text{TYCON of string } \ast \text{ ty list apply type constructor} \]

Each tycon takes fixed number of arguments.

- **nullary**: int, bool, string, ...
- **unary**: list, option, ...
- **binary**: \( \rightarrow \)
- **n-ary**: tuples (infix \( \ast \))
Syntax of ML types

Concrete syntax is baroque:

\[
\begin{align*}
  ty & \Rightarrow tyvar \quad \text{type variable} \\
  & \mid tycon \quad \text{(nullary) type constructor} \\
  & \mid ty \ tycon \quad \text{(unary) type constructor} \\
  & \mid (ty, \ldots, ty) \ tycon \quad \text{(n-ary) type constructor} \\
  & \mid ty \ast \ldots \ast ty \quad \text{tuple type} \\
  & \mid ty \rightarrow ty \quad \text{arrow (function) type} \\
  & \mid (ty)
\end{align*}
\]

\[
\begin{align*}
  tyvar & \Rightarrow ' \text{identifier} \quad 'a, 'b, 'c, \ldots \\
  tycon & \Rightarrow \text{identifier} \quad \text{list, int, bool, \ldots}
\end{align*}
\]
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

op o : \forall 'a, 'b, 'c .
      ('a \to 'b) \times ('c \to 'a) \to 'c \to 'b

length : \forall 'a . 'a list \to int

map : \forall 'a, 'b .
      ('a \to 'b) \to ('a list \to 'b list)

curry : \forall 'a, 'b, 'c .
       ('a \times 'b \to 'c) \to 'a \to 'b \to 'c

id : \forall 'a . 'a \to 'a