Standard ML

Data types
Concrete Datatypes

- The `datatype` declaration creates new types
- These are `concrete` data types, not `abstract`
- Concrete datatypes can be inspected - constructed and taken apart
- ML’s datatypes has two kinds of values: `atomic` and `composite`
- The only operations supported:
  - Use atomic values
  - Construct a composite value
  - Take a composite value apart: pattern matching
Enumeration Types

◆ Simplest example: single valued type
  
  ```plaintext
  datatype single = only;
  - only;
  
  val it = only : single
  ```
  
  • only denotes the only value in the type single
  • Isomorphic to unit

◆ Consisting of a finite number of constants
  
  ```plaintext
  - datatype bool = true | false;
  
  datatype bool = false | true
  - true;
  
  val it = true : bool
  - false;
  
  val it = false : bool
  ```
Enumeration Types

- No order on the elements (unlike Pascal, C)
  - datatype piece = king | queen | rook
    | bishop | knight | pawn;
  
  datatype piece = bishop | king | knight | pawn
    | queen | rook

- fun value king = Real.posInf (* infinity *)
  | value queen = 9.0
  | value rook = 5.0
  | value (bishop | knight) = 3.0
  | value pawn = 1.0;

val value = fn : piece -> real
- value bishop;
val it = 3.0 : int
Branding

**Newton’s second law:**

- `fun a m f = f/m;`
- `val a = fn : real -> real -> real`
- `val (body, engine) = (0.0122, 50.0);`
- `val a engine body; (* oops *)`
- `val it = 4098.36065574 : real`

**Type alias will not suffice here**

```ml
type mass = real and force = real
    and acceleration = real;

- `fun a (m:mass) (f:force) : acceleration = f/m;`
- `val a = fn : mass -> force -> acceleration`
- `val a engine body; (* still oops *)`
- `val it = 4098.36065574 : acceleration`
```
Constructors

Simulate branding using **datatype**:

```ml
datatype mass = Kg of real;
datatype force = Newton of real;
datatype acceleration = m_s'2 of real;
```

Constructors are **functions**

- `Kg`;
  ```ml
  val it = fn : real -> mass
  ```
- `Newton`;
  ```ml
  val it = fn : real -> force
  ```
- `val body = Kg 0.0122; val engine = Newton 50.0;`
  ```ml
  val body = Kg 0.0122 : mass
  val engine = Newton 50.0 : force
  ```

May be passed to other functions

- `map Kg [1.2, 5.3];`
Constructors

- Simulate branding using **datatype**:
  ```ml
datatype mass = Kg of real;
datatype force = Newton of real;
datatype acceleration = m_s'2 of real;
```

- Constructor names are the building blocks of **patterns**:
  ```ml
  fun a (Kg m) (Newton f) = m_s'2 (f / m);
  val a = fn : mass -> force -> acceleration
  - a body engine;
  val it = m_s'2 25.0 : acceleration
  - a engine body;
  Error: operator and operand don't agree [tycon mismatch]
  operator domain: mass
  operand: force
  in expression:
  ```
Variant Types

- We’ve already seen an example: enumerated types
- Shapes
  ```ml
datatype shape =
    point
    | Line of real
    | Circle of real
    | Rectangle of real*real;
  ```
- A tagged union
- Calculate area:
  ```ml
  fun area (point | Line _) = 0.0
  | area (Circle r) = Math.pi*r*r
  | area (Rectangle (w, h)) = w * h;
  val area = fn : shape -> real
  ```
Pattern Matching

In val binding:

val line = Line 5.3;
- val Line length = line;
val length = 5.3 : real
- val Circle radius = line;

uncaught exception Bind [nonexhaustive binding failure]

What will happen for the following:

- val point = point; (* OK. *)
- val point = 5.3; (* FAIL - types mismatch *)

Constructors cannot be rebound in a binding declaration
Let’s recall the definition of lists:

“Every list is either nil or has the form x::xs where x is its head and the list xs is its tail.”

We can do it ourselves!

```
datatype intlist =
    nil
  | :: of int * intlist;
```

What if we omit the “nil” part in the datatype definition?

Defining functions – as usual:

```
- fun length nil = 0
  | length (_::xs) = 1 + length xs;
```

We can’t have the [] syntax. That’s a built-in syntactic sugar.
“intlist”? Seriously? What’s so special about ints?

Well, nothing, of course. Guess what comes next?

```ml
datatype 'a list =
    nil | :: of 'a * ('a list);
```

- "hello" :: "world" :: nil;

```ml
val it = "hello" :: "world" :: nil : string list
```

In OOP terminology, it is called “generics”
Polymorphic Datatypes

- We can denote optional parameters in function:
  ```
  datatype 'a option = NONE | SOME of 'a;
  ```

- We can use datatypes to “unite” two different types. We can abstract on this idea, and create a datatype uniting any two types: ‘a and ‘b
  ```
  datatype ('a,'b) union = type1 of 'a | type2 of 'b;
  ```

- Example – in the next slide
Union - Example

datatype ('a,'b) union = type1 of 'a
    | type2 of 'b;

- val five = type1 5;
val five = type1 5 : (int,'a) union
- val hello = type2 "hello";
val hello = type2 "hello" : (a,string) union
- val five_or_hello = if true then five else hello;
val five_or_hello = type1 5 : (int,string) union
- val int_char_list = [type1 5, type2 #"a"];
val five_or_hello = [type1 5, type2 #"a"] : (int,char) union list
Trees

datatype 'a tree =
    Nil
  | Br of 'a * ('a tree) * ('a tree);

fun Leaf x = Br (x, Nil, Nil);

- val tree2 = Br (2, Leaf 1, Leaf 3);
val tree2 = Br (2, Br (1, Nil, Nil), Br (3, Nil, Nil)) : int tree
- val tree5 = Br (5, Leaf 6, Leaf 7);
- val tree4 = Br (4, tree2, tree5);
val tree4 = Br (4, Br (2, Br #, Br #), Br (5, Br #, Br #)) : int tree

- fun size Nil = 0
  | size (Br (v, t1, t2)) = 1 + size t1 + size t2;
val count = fn : 'a tree -> int
Binary Search Trees

- Implement an associative array using trees
- The tree will be of type `(int * 'a) tree`
  - `int` is the key, `'a` is the data in the node.
- Value may be anything
- Assumption: The tree is sorted
  - Any key on the left subtree is smaller than the current key
  - The current key is larger than any of the keys in the right subtree
- We will define two functions:
  - `insert = fn : (int * 'a) tree -> int * 'a -> (int * 'a) tree`
  - `get = fn : (int * 'a) tree -> int -> 'a`
- Two helpers:
  - `datatype order = LESS | EQUAL | GREATER (* predefined *)`
  - `val compare : int*int->order = Int.compare`
Binary Search Trees

fun get (Br ((node_k, v), left, right)) k = 
  case compare (node_k, k) of
    EQUAL => v
    | GREATER => get right k
    | LESS    => get left k

local
  fun compare (k1,_) (k2,_) = compare (k1, k2)
in
  fun insert Nil item = Br (item, Nil, Nil)
  | insert (Br (node, left, right)) item = 
    case cmp node item of
      EQUAL => Br (item, left, right)
      | GREATER => Br (node, left, insert right item)
      | LESS    => Br (node, insert left item, right)
  end