Useful examples

Let’s fix the fact that our only example datatype so far was silly…

• Enumerations, including carrying other data

```plaintext
datatype suit = Club | Diamond | Heart | Spade

datatype card_value = Jack | Queen | King
                   | Ace | Num of int
```

• Alternate ways of identifying real-world things/people

```plaintext
datatype id = StudentNum of int
            | Name of string
            * (string option)
            * string
```
Don’t do this

Unfortunately, bad training and languages that make one-of types inconvenient lead to common bad style where each-of types are used where one-of types are the right tool.

```plaintext
(* use the studen_num and ignore other fields unless the student_num is ~1 *)
{ student_num : int,
  first       : string,
  middle      : string option,
  last        : string }
```

• Approach gives up all the benefits of the language enforcing every value is one variant, you don’t forget branches, etc.

• And makes it less clear what you are doing
That said…

But if instead the point is that every “person” in your program has a name and maybe a student number, then each-of is the way to go:

```haskell
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```
Expression Trees

A more exciting (?) example of a datatype, using self-reference

datatype exp = Constant of int
  | Negate of exp
  | Add of exp * exp
  | Multiply of exp * exp

An expression in ML of type exp:

Add (Constant (10+9), Negate (Constant 4))

How to picture the resulting value in your head:

```
Add
  Constant
    19
  Negate
    Constant
      4
```
Recursion

Not surprising:
Functions over recursive datatypes are usually recursive

fun eval e =
  case e of
    | Constant i => i
    | Negate e2   => ~ (eval e2)
    | Add(e1,e2)  => (eval e1) + (eval e2)
    | Multiply(e1,e2) => (eval e1) * (eval e2)
Putting it together

```
datatype exp = Constant of int
            | Negate of exp
            | Add of exp * exp
            | Multiply of exp * exp
```

Let's define `max_constant : exp -> int`

Good example of combining several topics as we program:
- Case expressions
- Local helper functions
- Avoiding repeated recursion
- Simpler solution by using library functions

See the `.sml` file...
Careful definitions

When a language construct is “new and strange,” there is more reason to define the evaluation rules precisely…

… so let’s review datatype bindings and case expressions “so far”

– Extensions to come but won’t invalidate the “so far”
**Datatype bindings**

```plaintext
datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

Adds type `t` and constructors `Ci` of type `ti->t`
- `Ci v` is a value, i.e., the result “includes the tag”

Omit “of `t`” for constructors that are just tags, no underlying data
- Such a `Ci` is a value of type `t`

Given an expression of type `t`, use *case expressions* to:
- See which variant (tag) it has
- Extract underlying data once you know which variant
Datatype bindings

```java
case e of p1 => e1 | p2 => e2 | ... | pn => en
```

- As usual, can use a case expressions anywhere an expression goes
  - Does not need to be whole function body, but often is

- Evaluate e to a value, call it v

- If pi is the first pattern to match v, then result is evaluation of ei in environment “extended by the match”

- Pattern Ci(x1,...,xn) matches value Ci(v1,...,vn) and extends the environment with x1 to v1 ... xn to vn
  - For “no data” constructors, pattern Ci matches value Ci
Recursive datatypes

Datatype bindings can describe recursive structures
- Have seen arithmetic expressions
- Now, linked lists:

```ml
datatype my_int_list = Empty
  | Cons of int * my_int_list

val x = Cons(4,Cons(23,Cons(2008,Empty)))

fun append_my_list (xs,ys) = 
  case xs of 
    Empty => ys 
  | Cons(x,xs') => Cons(x, append_my_list(xs',ys))
```
Options are datatypes

Options are just a predefined datatype binding
- **NONE** and **SOME** are *constructors*, not just functions
- So use pattern-matching not **isSome** and **valOf**

```haskell
fun inc_or_zero intoption = 
  case intoption of
    NONE => 0
  | SOME i => i+1
```
Lists are datatypes

Do not use hd, tl, or null either
- [] and :: are constructors too
- (strange syntax, particularly infix)

fun sum_list xs =
    case xs of
        [] => 0
    | x::xs' => x + sum_list xs'

fun append (xs,ys) =
    case xs of
        [] => ys
    | x::xs' => x :: append (xs',ys)
Why pattern-matching

- Pattern-matching is better for options and lists for the same reasons as for all datatypes
  - No missing cases, no exceptions for wrong variant, etc.

- We just learned the other way first for pedagogy
  - Do not use `isSome`, `valOf`, `null`, `hd`, `tl` on Homework 2

- So why are `null`, `tl`, etc. predefined?
  - For passing as arguments to other functions (next week)
  - Because sometimes they are convenient
  - But not a big deal: could define them yourself
Excitement ahead…

Learn some deep truths about “what is really going on”
  – Using much more syntactic sugar than we realized

- Every val-binding and function-binding uses pattern-matching
- Every function in ML takes exactly one argument

First need to extend our definition of pattern-matching…
Each-of types

So far have used pattern-matching for one of types because we needed a way to access the values

Pattern matching also works for records and tuples:

- The pattern \((x_1, \ldots, x_n)\)
  matches the tuple value \((v_1, \ldots, v_n)\)
- The pattern \(\{f_1=x_1, \ldots, f_n=x_n\}\)
  matches the record value \(\{f_1=v_1, \ldots, f_n=v_n\}\)
  (and fields can be reordered)
Example

This is poor style, but based on what I told you so far, the only way to use patterns

- Works but poor style to have one-branch cases

```haskell
fun sum_triple triple = 
  case triple of 
    (x, y, z) => x + y + z

fun full_name r = 
  case r of 
    {first=x, middle=y, last=z} => 
      x ^ " " ^ y ^ " " ^ z
```
Val-binding patterns

• New feature: A val-binding can use a pattern, not just a variable
  – (Turns out variables are just one kind of pattern, so we just told you a half-truth in Lecture 1)

\[
\text{val } p = e
\]

• Great for getting (all) pieces out of an each-of type
  – Can also get only parts out (not shown here)

• Usually poor style to put a constructor pattern in a val-binding
  – Tests for the one variant and raises an exception if a different one is there (like \texttt{hd}, \texttt{tl}, and \texttt{valOf})
Better example

This is okay style

– Though we will improve it again next
– Semantically identical to one-branch case expressions

```haskell
fun sum_triple triple = 
  let val (x, y, z) = triple 
  in 
    x + y + z 
  end 

fun full_name r = 
  let val {first=x, middle=y, last=z} = r 
  in 
    x ^ " " ^ y ^ " " ^ z 
  end 
```
Function-argument patterns

A function argument can also be a pattern
– Match against the argument in a function call

```plaintext
fun f p = e
```

Examples (great style!):

```plaintext
fun sum_triple (x, y, z) =
    x + y + z

fun full_name {first=x, middle=y, last=z} =
    x ^ " " ^ y ^ " " ^ z
```
A new way to go

• For Homework 2:
  – Do not use the # character
  – Do not need to write down any explicit types
Hmm

A function that takes one triple of type int*int*int and returns an int that is their sum:

\[
\text{fun sum_triple } (x, y, z) = \quad \text{x + y + z}
\]

A function that takes three int arguments and returns an int that is their sum

\[
\text{fun sum_triple } (x, y, z) = \quad \text{x + y + z}
\]

See the difference? (Me neither.)

😊
The truth about functions

• In ML, every function takes exactly one argument (*)

• What we call multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
  – Elegant and flexible language design

• Enables cute and useful things you cannot do in Java, e.g.,

```ml
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left (rotate_left t)
```

* “Zero arguments” is the unit pattern () matching the unit value ()