What about more complex data?

- We’ve seen some base types and values:
  - Integers, Floats, Bool, String etc.

- Some ways to build up types:
  - Products (tuples), records, “lists”
  - Functions

- Design Principle: Orthogonality
  - Don’t clutter core language with stuff
  - Few, powerful orthogonal building techniques
  - Put “derived” types, values, functions in libraries

Many kinds of expressions:

1. Simple
2. Variables
3. Functions
Next: Building datatypes

Three key ways to build complex types/values

1. “Each-of” types
Value of T contains value of T1 and a value of T2

2. “One-of” types
Value of T contains value of T1 or a value of T2

3. “Recursive”
Value of T contains (sub)-value of same type T

Suppose I wanted ...

… a program that processed lists of attributes
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (float)
- Alive (boolean)
- Phone (int-int)
- email (string)

Many kinds of attributes (too many to put in a record)
- can have multiple names, addresses, phones, emails etc.
Want to store them in a list. Can I?

Constructing Datatypes

t is a new datatype.

A value of type t is either:
- a value of type t1 placed in a box labeled C1
- a value of type t2 placed in a box labeled C2
- …
- a value of type tn placed in a box labeled Cn

\[ \text{type } t = \text{C1 of } t1 | \text{C2 of } t2 | \ldots | \text{Cn of } tn \]
Constructing Datatypes

\[\text{type } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \mid \ldots \mid C_n \text{ of } t_n\]

Suppose I wanted ...

Attributes:
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- Email (string)

\[\text{type attrib = Name of string \mid Age of int \mid DOB of int*int*int \mid Address of string \mid Height of float \mid Alive of bool \mid Phone of int*int \mid Email of string;};\]

How to PUT values into box?

How to create values of type `attrib`?

\[# \text{let } a1 = \text{Name } \text{"Ranjit"};;;\]
\[# \text{let } a2 = \text{Height } 5.83;;\]
\[# \text{let } a3 = \text{DOB } (9,8,\text{year}) ;;\]
\[# \text{let } a_l = [a1;a2;a3];;\]
\[# \text{val } x : \text{attrib} = \text{Name } \text{"Ranjit"};;\]
\[# \text{val } a2 : \text{attrib} = \text{Height } 5.83;;\]
\[# \text{val } year : \text{int} = 1977 ;;\]
\[# \text{val } a_l : \text{attrib list} = \ldots\]
Constructing Datatypes

<table>
<thead>
<tr>
<th>type attrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>= Name of string</td>
</tr>
<tr>
<td>Address of string</td>
</tr>
<tr>
<td>Phone of int*int</td>
</tr>
</tbody>
</table>

One-of types

- We’ve defined a “one-of” type named attrib
- Elements are one of:
  - string,
  - int,
  - int*int*int,
  - float,
  - bool ...

- Can create uniform attrib lists
- Say I want a function to print attribs...

How to TEST & TAKE what's in box?

Is it a ... string? or an int? or an int*int*int? or ...
Pattern-match expression: check if e is of the form ...

- On match:
  - value in box bound to pattern variable
  - matching result expression is evaluated
- Simultaneously test and extract contents of box

None of the cases matched the tag (Name)
Causes nasty *Run-Time Error*
Beware! Handle All TAGS!

None of the cases matched the tag (Name)
Causes nasty Run-Time Error

Compiler To The Rescue!!

### Compile-time checks for:

**missed cases:** ML warns if you miss a case!

### Compile-time checks for:

**redundant cases:** ML warns if a case never matches
Benefits of `match-with`

1. **Simultaneous test-extract-bind**
2. **Compile-time checks for:**
   - missed cases: ML warns if you miss a `t` value
   - redundant cases: ML warns if a case never matches

`match-with` is an Expression

```
match e with
  C1 x1 -> e1
  | C2 x2 -> e2
  | ...
  | Cn xn -> en
```

**Type Rule**

- `e1, e2,…,en` must have same type `T`
- Type of whole expression is `T`

Next: Building datatypes

Three key ways to build complex types/values

1. **“Each-of”** types `t1 * t2`
   Value of `T` contains value of `T1` and a value of `T2`

2. **“One-of”** types `type t = C1 of t1 | C2 of t2`
   Value of `T` contains value of `T1` or a value of `T2`

3. **“Recursive”** type
   Value of `T` contains (sub)-value of same type `T`
“Recursive” types

\[
\text{type } \text{nat} = \text{Zero} \mid \text{Succ of nat}
\]

Wait a minute! \text{Zero} of what?!

Relax. Means “empty box with label \text{Zero}”

What are values of \text{nat}?
type nat = Zero | Succ of nat

What are values of nat?

One nat contains another!


**“Recursive” types**

```
type nat = Zero | Succ of nat
```

What are values of `nat`?

One `nat` contains another!

`nat` = recursive type

---

**Next: Building datatypes**

Three key ways to build complex types/values

1. **“Each-of” types** `t1 * t2`
   Value of `T` contains value of `T1` **and** a value of `T2`

2. **“One-of” types** `type t = C1 of t1 | C2 of t2`
   Value of `T` contains value of `T1` **or** a value of `T2`

3. **“Recursive” type** `type t = ... | C of (...*t)`
   Value of `T` contains (sub)-value of same type `T`

---

**Next: Lets get cosy with Recursion**

- Recursive Code Mirrors Recursive Data
- Code Structure = Type Structure!!!
let rec to_int n = match n with
| Zero   -> 0
| Succ m -> 1 + to_int m

Base pattern
Inductive pattern
Base Expression
Inductive Expression
of_int : int -> nat

let rec of_int n =
  if n <= 0 then
    Zero
  else
    Succ (of_int (n-1))
plus : nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec plus (n,m) =

match m with
| Zero    -> n
| Succ m' -> Succ (plus (n,m'))

plus : nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec plus (n,m) =

match m with
| Zero    -> n
| Succ m' -> Succ (plus (n,m'))
times: nat*nat -> nat

let rec times (n,m) =

match m with
| Zero    -> Zero
| Succ m' -> plus n (times (n,m'))
Lists are recursive types!

```ocaml
type int_list = 
  Nil 
| Cons of int * int_list
```

Think about this! What are values of `int_list`?

- `Cons(1,Cons(2,Cons(3,Nil)))`
- `Cons(2,Cons(3,Nil))`
- `Cons(3,Nil)`
- `Nil`

Lists aren’t built-in!

Lists are a derived type: built using elegant core!

1. Each-of
2. One-of
3. Recursive

- `Nil` is just a pretty way to say “Cons”
- `[]` is just a pretty way to say “Nil”

Some functions on Lists: Length

```ocaml
let rec len l = 
  match l with 
    Nil -> 0
  | Cons(_,t) -> 1 + (len t)
```

Matches everything, no binding

Pattern-matching in order
- Must match with `Nil`
Some functions on Lists: Append

```
let rec append (l1,l2) =
  Base Expression
  Inductive Expression
  Base pattern
  Ind pattern
```

- Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy

null, hd, tl are all functions ...

Bad ML style: More than aesthetics!

Pattern-matching better than test-extract:
- ML checks all cases covered
- ML checks no redundant cases
- ...at compile-time:
  - fewer errors (crashes) during execution
  - get the bugs out ASAP!

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data
Representing Trees

```
type tree =
| Leaf of int
| Node of tree*tree
```

Leaf
1

Node(Node(Leaf 1, Leaf 2), Leaf 3)

Representing Trees

```
type tree =
| Leaf of int
| Node of tree*tree
```

Leaf
2

Node(Node(Leaf 1, Leaf 2), Leaf 3)

Representing Trees

```
type tree =
| Leaf of int
| Node of tree*tree
```

Leaf
3

Node(Node(Leaf 1, Leaf 2), Leaf 3)
Representing Trees

type tree =
| Leaf of int
| Node of tree*tree

Node(Node(Leaf 1, Leaf 2), Leaf 3)

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

sum_leaf: tree -> int

“Sum up the leaf values”. E.g.

# let t0 = Node(Node(Leaf 1, Leaf 2), Leaf 3);;
- : int = 6

**sum_leaf**: tree -> int

```
type tree =
| Leaf of int
| Node of tree*tree

let rec sum_leaf t =
```

**Base pattern**
```
| Leaf n     -> n
| Node(t1,t2) -> sum_leaf t1 + sum_leaf t2
```
Recursive Code Mirrors Recursive Data

Code almost writes itself!

Another Example: Calculator
Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9
- 3.78 - 5.92
- (4.0 + 2.9) * (3.78 - 5.92)

Whats a ML TYPE for REPRESENTING expressions?

type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr

Another Example: Calculator
Want an arithmetic calculator to evaluate expressions like:

- 4.0 + 2.9 ===> 6.9
- 3.78 - 5.92 ===> -2.14
- (4.0 + 2.9) * (3.78 - 5.92) ===> -14.766

Whats a ML TYPE for REPRESENTING expressions?
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 \implies 6.9$
- $3.78 - 5.92 \implies -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) \implies -14.766$

Whats a ML FUNCTION for EVALUATING expressions?

```ml
type expr =
| Num of float
| Add of expr*expr
| Sub of expr*expr
| Mul of expr*expr
```

```ml
let rec eval e = match e with
  | Num f -> f
  | Add(e1, e2) -> eval e1 +. eval e2
  | Sub(e1, e2) -> eval e1 -. eval e2
  | Mul(e1, e2) -> eval e1 * eval e2
```

Saturday, April 7, 2012
Random Art from Expressions

PA #2

Build more funky expressions, evaluate them, to produce:

Saturday, April 7, 2012