Review so far

- 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

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)
- ...
Suppose I wanted ...

... a program that processed lists of attributes

• Name (string)
• Age (integer)
• Date of Birth (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

Constructing Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

All have the type t

How to PUT values into box?

How to create values of type attrib?

let a1 = Name "Bob";;
val a1 : attrib = Name "Bob"

let a2 = Height 5.83;;
val a2 : attrib = Height 5.83

let a3 = DOB (9,8,1977);;
val a3 : attrib = DOB (9,8,1977)

let a_list = [a1; a2; a3];;
val a_list : attrib list = ...
Constructing Datatypes

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>DOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Bob&quot;</td>
<td>34</td>
<td>(9, 8, 77)</td>
</tr>
</tbody>
</table>

All have type attrib

One-of types

- We've defined a “one-of” type named attrib
- Elements are one of:
  - string,
  - int,
  - int*int,
  - float,
  - bool ...
- Can create uniform attrib lists
- Say I want a function to print attribs...

How to TEST & TAKE whats in box?

Is it a ...
string?
or an int?
or an int*int*int?
or ...

Look at TAG!

How to tell whats in the box ?

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

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
How to tell what's in the box

```haskell
# match (Name "Bob") with
    | Name s -> printf "Hello %s\n" s
    | Age i -> printf "%d years old" i
    ;;
Hello Bob
- : unit = ()

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

How to TEST & TAKE what's in box?

```haskell
BEWARE!!
Be sure to handle all TAGS!
```

Beware! Handle All TAGS!

```haskell
# match (Name "Bob") with
    | Age i -> Printf.printf "%d" I
    | Email s -> Printf.printf "%s" s
    ;;
Exception: Match Failure!!

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

Compiler to the Rescue!

```haskell
# let printAttrib a =  match a with
    | Name s -> Printf.printf "%s" s
    | Age i -> Printf.printf "%d" i
    | DOB (d,m,y) -> Printf.printf "%d / %d / %d" d m y
    | Address addr -> Printf.printf "%s" addr
    | Height h -> Printf.printf "%d" h
    | Alive b -> Printf.printf "%b" b
    | Email e -> Printf.printf "%s" e
    ;;
Warning P: this pattern-matching is not exhaustive.Here is an example of a value that is not matched:Phone (_, _)
```

**Compiler To The Rescue!!**

Compile-time checks for:
*missed* cases: ML warns if you miss a case!

```haskell
# let printAttrib a = match a with
    | Name s -> Printf.printf "%s" s
    | Age i -> Printf.printf "%d" i
    | DOB (d,m,y) -> Printf.printf "%d / %d / %d" d m y
    | Address addr -> Printf.printf "%s" addr
    | Height h -> Printf.printf "%d" h
    | Alive b -> Printf.printf "%b" b
    | Email e -> Printf.printf "%s" e
    ;;
Warning U: this match case is unused.
```

**Compiler To The Rescue!!**

Compile-time checks for:
*redundant* cases: ML warns if a case never matches
Another Few Examples

```ocaml
# let printAttrib a = match a with
| Name s -> Printf.printf "%s" s
| Age i -> Printf.printf "%d" i
| DOB (d, m, y) -> Printf.printf "%d / %d / %d" d m y
| ... |
| Age i -> Printf.printf "%d" i ;;

Warning U: this match case is unused.
```

See code text file

**match-with** is an Expression

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

**Type Rule**
- \(e_1, e_2, ..., e_n\) must have the same type \(T\)
- Type of whole expression is \(T\)

**match-with** is an Expression

```
match e with
| Name s -> e1
| Age i -> e2
| DOB (d, m, y) -> e3
| Address a -> e4
| Height h -> e5
| Alive b -> e6
| Phone (a, n) -> e7
| Email e -> e8
```

**Type Rule**
- \(e_1, e_2, ..., e_n\) must have the same type \(T\)
- Type of whole expression is \(T\)

**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

Next: Building datatypes

Three key ways to build complex types/values

1. **“Each-of”** types \(t_1 * t_2\)
   Value of \(T\) contains value of \(T_1\) and a value of \(T_2\)

2. **“One-of”** types \(type t = C1 of t1 | C2 of t2\)
   Value of \(T\) contains value of \(T_1\) or a value of \(T_2\)

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

“Recursive” types

```ocaml
type nat = Zero | Succ of nat
```
“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}? One \text{nat} contains another!
“Recursive” types

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

What are values of \text{nat}?

One \text{nat} contains another!

Next: Building datatypes

Three key ways to build complex types/values

1. “Each-of” types \text{t1} * \text{t2}
   Value of \text{T} contains value of \text{T1} and a value of \text{T2}

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

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

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

\[
\text{to_int : nat} \rightarrow \text{int}
\]

```ocaml
let rec to_int n =
```

Next: Lets get cosy with Recursion

Code Structure = Type Structure!!!
to_int : nat -> int

let rec to_int n =

of_int : int -> nat

let rec of_int n =

of_int : int -> nat

let rec of_int n =
plus : nat*nat → nat

let rec plus m n =

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

times : nat*nat → nat

let rec times n m =

match m with
| Zero -> n
| Succ m' -> Succ (times n m')
10/3/2013

**times: nat*nat -> nat**

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

let rec times n m =
  match m with
    Zero -> n
  | Succ m' -> plus n (times n m')
```

**Lists are recursive types!**

```plaintext
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 |
|--------------------------------|------------------------|--------------|
| CAS:                           | 1                       | 2            | 3   |

**Lists aren’t built-in!**

```plaintext
datatype int_list =
    Nil
  | Cons of int * int_list
```

Lists are a derived type: built using elegant core!

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

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

**Some functions on Lists: Length**

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

No binding for head

Pattern-matching in order
Some functions on Lists: Append

```
let rec append (l1, l2) =
```

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

Well designed datatype gives strategy

Some functions on Lists: Max

```
let rec max xs =
```

- 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

Representing Trees

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

Leaf 2
Representing Trees

```
<table>
<thead>
<tr>
<th>Leaf</th>
<th>Leaf 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
```

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

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
```

```
let rec sum_leaf =
```

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

sum_leaf: tree -> int

let rec sum_leaf t =

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)

Recursive Code Mirrors Recursive Data

Code almost writes itself!

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:
• \(4.0 + 2.9 \Rightarrow 6.9\)
• \(3.78 - 5.92 \Rightarrow -2.14\)
• \((4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766\)

What's a ML TYPE for REPRESENTING expressions?

<table>
<thead>
<tr>
<th>type expr =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of float</td>
</tr>
<tr>
<td>Add of expr*expr</td>
</tr>
<tr>
<td>Sub of expr*expr</td>
</tr>
<tr>
<td>Mul of expr*expr</td>
</tr>
</tbody>
</table>

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:
• \(4.0 + 2.9 \Rightarrow 6.9\)
• \(3.78 - 5.92 \Rightarrow -2.14\)
• \((4.0 + 2.9) \times (3.78 - 5.92) \Rightarrow -14.766\)

What's a ML FUNCTION for EVALUATING expressions?

<table>
<thead>
<tr>
<th>type expr =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of float</td>
</tr>
<tr>
<td>Add of expr*expr</td>
</tr>
<tr>
<td>Sub of expr*expr</td>
</tr>
<tr>
<td>Mul of expr*expr</td>
</tr>
</tbody>
</table>

Random Art from Expressions

PA #2
Build more funky expressions, evaluate them, to produce: