News

On webpage:
• PA #1 (due Friday 4/20)
• In Class Exercises, bring laptops

“Recursive” types

What are values of \texttt{nat}?
One \texttt{nat} contains another!
\texttt{nat} = \texttt{recursive type}

Next: Building datatypes

Three key ways to build complex types/values

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

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

3. “Recursive” type \texttt{type t = ... | C of (...*t)}
Value of \texttt{T} contains (sub)-value of same type \texttt{T}
Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

of_int : int -> nat

let rec of_int n =
  if n <= 0 then
    Zero
  else
    Succ (of_int (n-1))

of_int 0 ===> Zero
of_int 1 ===> Succ (of_int 0) 
            ===> Succ (Zero)
of_int 2 ===> Succ (of_int 1) 
            ===> Succ (Succ (Zero))

5

Monday, April 16, 2012

5

Monday, April 16, 2012

of_int

Base Expression

Inductive Expression

Base pattern

Inductive pattern

Base pattern

Inductive pattern

plus: nat-> nat -> nat

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

plus Zero (Succ (Succ Zero))
  ===> Succ (Succ Zero)

plus (Succ Zero) (Succ (Succ Zero))
  ===> Succ (plus Zero (Succ (Succ Zero))
  ===> Succ (Succ (Succ (Succ Zero)))

7

Monday, April 16, 2012

7

Monday, April 16, 2012

let rec to_int n = match n with
| Zero   -> 0
| Succ m -> 1 + to_int m

to_int Zero ===> 0

to_int (Succ Zero) ===> 1 + to_int Zero
  ===> 1 + 0
  ===> 1

to_int (Succ (Succ Zero)) ===> 1 + to_int (Succ Zero)
  ===> 1 + 1
  ===> 2

6

Monday, April 16, 2012

6

Monday, April 16, 2012

to_int

Base Expression

Inductive Expression

Base pattern

Inductive pattern

Base pattern

Inductive pattern
minus: nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec minus (n,m) =

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

Think about this! What are values of int list ?

```ocaml
type 'a list =
  Nil
| Cons of 'a * 'a list
```

Think about this! What are values of int list ?

Nil

Cons(3,Nil) Nil

Cons 3 , Nil

Lists are recursive types!
Lists are recursive types!

Think about this! What are values of int list?

Cons(2,Cons(3,Nil))  Cons(3,Nil)  Nil
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

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

Next: Let’s get cozy with Recursion

Recursive Code Mirrors Recursive Data
Some functions on Lists: `len`

```ocaml
define len =
  match l with
  | []       -> 0
  | h::t     -> 1 + (len t)
```

**Base pattern**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
</tr>
<tr>
<td>h::t</td>
<td>1 + (len t)</td>
</tr>
</tbody>
</table>

**Inductive pattern**

Matches everything, no binding

Pattern-matching in order
- Must match with []

Some functions on Lists: `sum`

```ocaml
(* val sum : int list -> int *)
define sum =
  match xs with
  | []       -> 0
  | h::t     -> h + (sum t)
```

**Base pattern**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
</tr>
<tr>
<td>h::t</td>
<td>h + (sum t)</td>
</tr>
</tbody>
</table>

**Inductive pattern**

Some functions on Lists: `member`

```ocaml
(* val mem : 'a -> 'a list -> bool *)
define mem =
  match ys with
  | []       -> false
  | y::ys    -> if x=y then true else mem x ys'
```

**Base pattern**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>false</td>
</tr>
<tr>
<td>y::ys</td>
<td>if x=y then true else mem x ys'</td>
</tr>
</tbody>
</table>

**Inductive pattern**

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

Well designed datatype gives strategy
Some functions on Lists: member

(* val mem : 'a -> 'a list -> bool *)

let rec mem x ys =
  match ys with
  | [] -> false
  | y::ys' -> if x = y then true else mem x ys'

Base Expression
Inductive Expression
Base pattern
Ind pattern

mem 2 (2::[]) ===> true

mem 2 (1::2::[]) ===> mem 2 (2::[]) ===> true

Online

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

Well designed datatype gives strategy
Some functions on Lists: append

```ml
let rec append xs ys =
  Base Expression
  Inductive Expression
  Base pattern
  Ind pattern
  append [] [] ===> []
  append [] (3::4::[]) ===> 3::4::[]
  append (2::[]) (3::4::[]) ===> 2::(3::4::[])
  ===> 2::3::4::[]
  append (1::2::[]) (3::4::[]) ===> 1::2::(3::4::[])
  ===> 1::2::3::4::[]
```

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!

Some functions on Lists: clone

```ml
(* val clone : int -> 'a -> 'a list *)
let rec clone n x =
```

Next: Let's 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)

Leaf 2

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

Leaf 3

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

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

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

```
sum_leaf: tree -> int

"Sum up the leaf values". E.g.
```

```ocaml
# let t0 = Node(Node(Leaf 1, Leaf 2), Leaf 3);;
# sum_leaf t0 ;;
- : int = 6
```
sum_leaf: tree -> int

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

let rec sum_leaf t =
```

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

let rec sum_leaf t =
```

```plaintext
match t with
| Leaf n     -> n
| Node(t1,t2)-> sum_leaf t1 + sum_leaf t2
```

Monday, April 16, 2012
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:

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

What's a ML FUNCTION for EVALUATING expressions?

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

Random Art from Expressions

PA #2

Build more funky expressions, evaluate them, to produce: