

CSE 130 : Spring 2011

Programming Languages

Lecture 4: Datatypes

Ranjit Jhala
UC San Diego



Recap: Environments

“Phone book”

- Variables = “names”
- Values = “phone number”

1. Evaluate:

Find and use most recent value of variable

2. Extend: $\text{let } x = e ; ;$

Add new binding at end of “phone book”

Recap

- Variables are **names** for **values**
 - Environment: dictionary/phonebook
 - Most recent binding used
 - Entries never changed, new entries added
- Environment **frozen at fun definition**
 - Re-binding variables cannot change a function
 - Same I/O behavior at every call

Function bindings

Functions are values, can bind using val

```
let fname = fun x -> e ;;
```

Problem: Can't define recursive functions !

- `fname` is bound **after** computing rhs value
- no (or “old”) binding for occurrences of `fname` inside `e`

```
let rec fname x = e ;;
```

Occurrences of `fname` inside `e` bound to “this” definition

```
let rec fac x = if x <= 1 then 1 else x * fac (x-1)
```

Local bindings

So far: bindings that remain until a re-binding (“global”)

Local, “temporary” variables are useful inside functions

- Avoid repeating computations
- Make functions more readable

Let-in is an expression!

```
let x = e1 in  
  e2  
;;
```

Evaluating let-in in env E:

1. Evaluate expr **e1** in env E to get value **v : t** /
2. Use extended E [x := **v : t**] (only) to evaluate **e2**

Local bindings

Evaluating let-in in env E:

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended E $[x := v : t]$ to evaluate e_2

```
let
  x = 10
in
  x * x
;;
```

...	...
-----	-----

...	...
x	10 : int

...	...
-----	-----

Let-in is an expression!

Evaluating let-in in env E:

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended E $[x := v : t]$ to evaluate e_2

```
let y =
```

```
  let
```

```
    x = 10
```

```
  in
```

```
    x * x
```

```
;;
```

100

M	M
---	---

M	M
x	10 : int

M	M
y	100 : int

+

Nested bindings

Evaluating let-in in env E:

1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended E $[x := v : t]$ to evaluate e_2

```
let
  x = 10
in
  (let
    y = 20
  in
    x * y)
+ x
;;
```

...	...
-----	-----

...	...
x	10 : int

...	...
x	10 : int
y	20 : int

...	...
x	10 : int

...	...
-----	-----

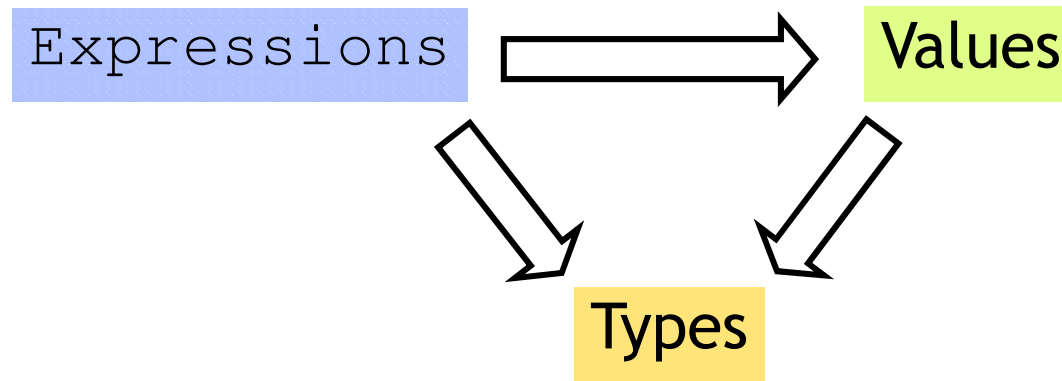
Nested bindings

```
let
  x = 10
in
  let
    y = 20
  in
    x * y
;;
```

```
let x = 10 in
let y = 20 in
  x * y
;;
```

Correct Formatting

What about more complex data ?



Many kinds of expressions:

1. Simple
2. Variables
3. Functions

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**

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

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

t is a new datatype.

A value of type t is either:

a value of type $t1$ placed in a box labeled $C1$

Or a value of type $t2$ placed in a box labeled $C2$

Or ...

Or a value of type tn placed in a box labeled Cn

Suppose I wanted ...

Attributes:

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- email (string)

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

Creating Values

How to create values of type `attrib` ?

```
# let a1 = Name "Ranjit";;  
val x : attrib = Name "Ranjit"  
# let a2 = Height 5.83;;  
val a2 : attrib = Height 5.83  
# let year = 1977 ;;  
val year : int = 1977  
# let a3 = DOB (9,8,year) ;;  
val a3 : attrib = DOB (9,8,1977)  
# let a_l = [a1;a2;a3];;  
val a3 : attrib list = ...
```

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

One-of types

- We've defined a “one-of” type named `attrib`

- Elements are one of:

- `string`,
- `int`,
- `int*int*int`,
- `float`,
- `bool ...`

```
datatype attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of real  
| Alive of bool  
| Phone of int*int  
| Email of string;
```

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

How to tell whats in the box ?

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

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

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

match-with is an Expression

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

match-with is an Expression

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

Type rules ?

- $e_1, e_2, \dots, e_n$ must have same type
- Which is type of whole expression

Benefits of match-with

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

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

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

What about “Recursive” types ?

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

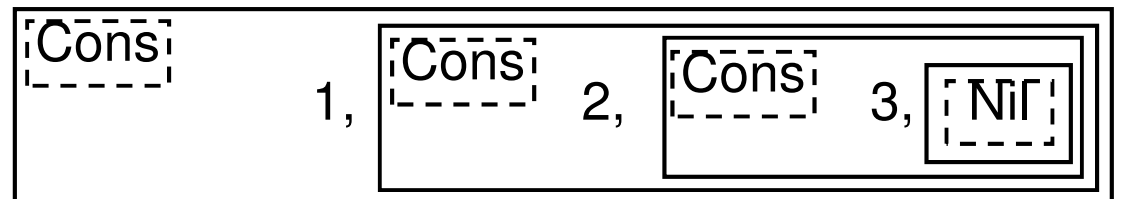
Think about this! What are values of `int_list` ?

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

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

`Cons(3,Nil)`

`Nil`



Lists aren't built-in !

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

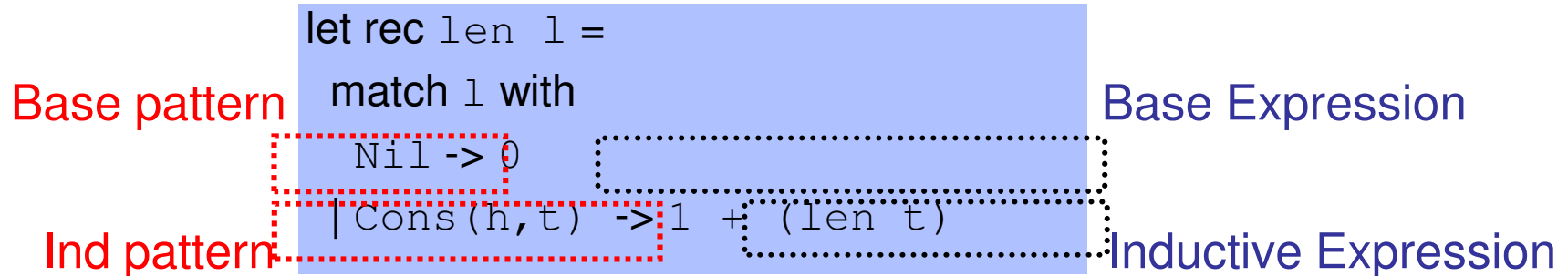
Lists are a **derived** type: built using elegant core!

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



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

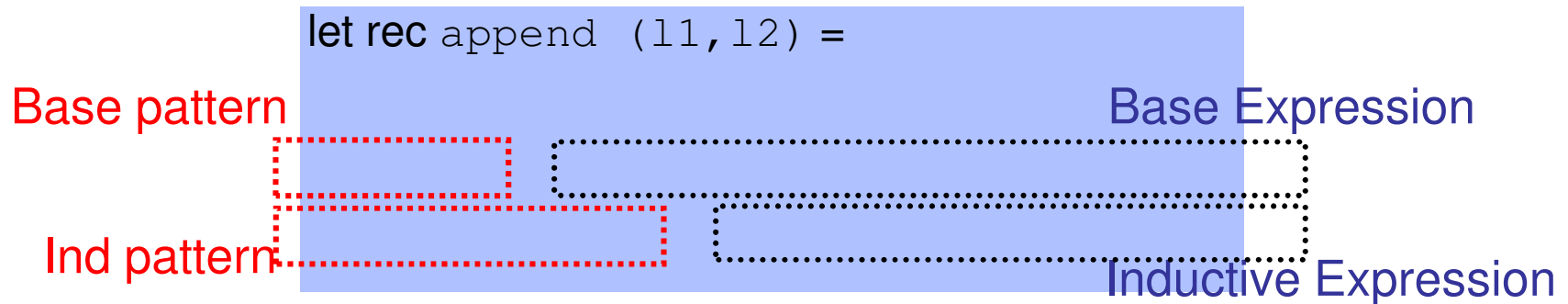
Matches everything, no binding

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

Pattern-matching in order

- Must match with `Nil`

Some functions on Lists : Append



- 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!

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

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

Q: Whats a ML datatype for such expressions ?

Another Example: Calculator

We 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 function for **evaluating** such expressions ?

