CSE 130
Programming Languages

Datatypes
Many kinds of expressions:

1. Simple
2. Variables
3. Functions
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
Three key ways to build complex types/values

1. “Each-of” types \((T1 \ast T2)\)
   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)
- 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

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

\( t \) is a new datatype.

A value of type \( t \) is either:

- a value of type \( t1 \) placed in a box labeled \( \text{C1} \)
- a value of type \( t2 \) placed in a box labeled \( \text{C2} \)
- \( \ldots \)
- a value of type \( tn \) placed in a box labeled \( \text{Cn} \)
Constructing Datatypes

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

All have the type \( t \)
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;
How to PUT values into box?
How to PUT values into box?

How to create values of type `attrib`?

```ocaml
# let a1 = Name "Bob";;
val x : attrib = Name "Bob"
# 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 = …
```

```ocaml
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;;
```
Constructing Datatypes

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;

All have type attrib
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...

```plaintext
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;
```
How to TEST & TAKE whats in box?

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

How to TEST & TAKE what's in box?

Look at TAG!
How to tell what's in the box?

```latex
match e with
| Name  | s   | -> printf "\%s" s |
| Age   | i   | -> printf "\%d" i  |
| DOB(d, m, y) | -> printf "\%d/\%d/\%d" d m y |
| Address | s   | -> printf "\%s" s  |
| Height  | h   | -> printf "\%f" h   |
| Alive   | b   | -> printf "\%b" b s |
| Phone(a, r) | -> printf "(\%d)-\%d" a r |
```

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 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
```

```
match e with
| Name s  -> ...(*s: string *)
| Age i   -> ...(*i: int *)
| DOB(d,m,y) -> ...(*d: int,m: int,y: int*)
| Address a  -> ...(*a: string*)
| Height h  -> ...(*h: int *)
| Alive b   -> ...(*b: bool*)
| Phone(a,r)  -> ...(*a: int, r: int*)
```

**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

None of the cases matched the tag (Name)

Causes nasty Run-Time Error
How to TEST & TAKE whats in box?

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!

# 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!!

```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
    | Address addr -> Printf.printf "%s" addr
    | Height h -> Printf.printf "%f" 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 (_, _)
```

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

**Compile-time checks for:**

redundant cases: **ML warns if a case never matches**
Another Few Examples

```ml
# 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

\[
\text{match } e \text{ with} \\
\quad C_1 \ x_1 \rightarrow e_1 \\
\| \ C_2 \ x_2 \rightarrow e_2 \\
\| \ \ldots \\
\| \ C_n \ x_n \rightarrow e_n
\]

**Type Rule**
- \( e_1, e_2, \ldots, e_n \) must have same type \( T \)
- Type of whole expression is \( T \)
**match-with is an Expression**

\[
\text{Type Rule}
\]

- \(e_1, e_2, \ldots, e_n\) must have 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
Three key ways to build complex types/values

1. “Each-of” types \( t_1 \ast t_2 \)
   Value of T contains value of T1 and a value of T2

2. “One-of” types \( \text{type } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \)
   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

```haskell
type nat = Zero | Succ of nat
```
“Recursive” types

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

Wait a minute! Zero of what?!
“Recursive” types

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

Wait a minute! Zero of what?! Relax.
Means “empty box with label Zero”
“Recursive” types

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

What are values of \text{nat}?
“Recursive” types

type nat = Zero | Succ of nat

What are values of nat?
“Recursive” types

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

What are values of \text{nat}?

One \text{nat} contains another!
“Recursive” types

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

What are values of \text{nat}?
One \text{nat} contains another!
“Recursive” types

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

What are values of \text{nat}?

One \text{nat} contains another!
“Recursive” types

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

What are values of \text{nat}?
One \text{nat} contains another!

\text{nat} = \text{recursive type}
Next: Building datatypes

Three key ways to build complex types/values

1. “Each-of” types \( t_1 \ast t_2 \)
   Value of T contains value of T1 and a value of T2

2. “One-of” types \( \text{type } t = \text{C1 of } t_1 \mid \text{C2 of } t_2 \)
   Value of T contains value of T1 or a value of T2

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

Recursive Code Mirrors Recursive Data
Next: Lets get cosy with Recursion

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

type nat =
| Zero
| Succ of nat

let rec to_int n =
to_int : nat -> int

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

```ocaml```
let rec to_int n =
```
**to\_int : nat \rightarrow int**

**type nat =**
- Zero
- Succ of nat

**let rec to\_int n = match n with**
- Zero \rightarrow 0
- Succ m \rightarrow 1 + to\_int m
of_int : int -> nat

type nat =
| Zero
| Succ of nat

let rec of_int n =
let rec of_int n =
of_int : int -> nat

```ml
let rec of_int n =
  if n <= 0 then
  else
```

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

Base pattern

Inductive pattern

Base pattern

Inductive pattern
of_int : int -> nat

**Type Definition**

type nat =
  | Zero
  | Succ of nat

**Recursive Definition**

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

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

let rec plus n m =
```
plus : nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec plus n m =
plus : nat*nat -> nat

**Type Definition**

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

**Recursive Function**

```
let rec plus n m =
  match m with
  | Zero ->
  | Succ 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\times\text{nat} \rightarrow \text{nat}

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

let rec times n m =
**times**: `nat*nat -> nat`

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

```
let rec times n m =
```
times: nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec times n m =
match m with
| Zero ->
| Succ m' ->
times: nat*nat -> nat

type nat =
| Zero
| Succ of nat

let rec times n m =
match m with
| Zero -> Zero
| Succ m' -> plus n (times n m')
Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data
Lists are recursive types!

```haskell
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

::: is just a pretty way to say “Cons”
[ ] is just a pretty way to say “Nil”
Some functions on Lists: Length

\[
\text{let rec } \text{len} \ l = \\
\quad \text{match } l \text{ with} \\
\quad \quad \text{Nil} \to 0 \\
\quad \quad \text{Cons}(h,t) \to 1 + (\text{len } t)
\]

Base pattern
Inductive pattern
Base expression
Inductive expression

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

```ml
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

\[
\begin{align*}
\text{type } & \text{tree } = \\
| \text{Leaf of int} & \\
| \text{Node of tree*tree} & \\
\end{align*}
\]
Representing Trees

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

Node(Leaf 1, Leaf 2)
Representing Trees

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

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

Node(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.

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

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

let rec sum_leaf t =
```
sum_leaf: tree -> int

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

let rec sum_leaf t =
```

sum_leaf: tree -> int

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

let rec sum_leaf t =
match t with
  | Leaf n ->
  | Node(t1,t2)->
sum_leaf: tree -> int

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

let rec sum_leaf t =
match t with
  | 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:

- $4.0 + 2.9$
- $3.78 - 5.92$
- $(4.0 + 2.9) \times (3.78 - 5.92)$
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

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

What's a ML **TYPE** for **REPRESENTING** expressions?
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**

What's a ML **TYPE** for **REPRESENTING** expressions?

```ml
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?

```haskell
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:

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

What's a ML FUNCTION for EVALUATING expressions?

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

let rec eval e =
match e with
| Num f ->
| Add (e1, e2) ->
| Sub (e1, e2) ->
| Mul (e1, e2) ->
```
Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

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

What's a ML FUNCTION for EVALUATING expressions?

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

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