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**
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

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

\( t \) is a new datatype.
A value of type \( t \) is either:
- a value of type \( t_1 \) placed in a box labeled \( C_1 \)
- a value of type \( t_2 \) placed in a box labeled \( C_2 \)
- …
- a value of type \( t_n \) placed in a box labeled \( C_n \)
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)

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

How to PUT values into box?

How to create values of type \( \text{attrib} \) ?

```plaintext
# 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 a_l : attrib list = …
```
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;
```

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;
```

How to TEST & TAKE what's in box?

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

Look at TAG!
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!!**
Be sure to handle all TAGS!
None of the cases matched the tag (Name) Causes nasty **Run-Time Error**

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

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 \texttt{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

\texttt{match-with} is an Expression

\begin{align*}
\texttt{match } e \texttt{ with} & \quad \texttt{type } t = \\
& \quad \texttt{C1 } x_1 \rightarrow e_1 \\
& \quad | \texttt{C2 } x_2 \rightarrow e_2 \\
& \quad | \ldots \\
& \quad | \texttt{Cn } x_n \rightarrow e_n
\end{align*}

Type Rule

- \( e_1, e_2, \ldots, e_n \) must have same type \( T \)
- Type of whole expression is \( T \)

Next: Building datatypes

Three key ways to build complex types/values

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

2. \texttt{“One-of”} types \( \texttt{type } t = C_1 \texttt{ of } t_1 | C_2 \texttt{ of } t_2 \)
   Value of \( T \) contains value of \( T_1 \) or a value of \( T_2 \)

3. \texttt{“Recursive”} type
   Value of \( T \) contains (sub)-value of same type \( T \)
“Recursive” types

type nat = Zero | Succ of nat

Wait a minute! Zero of what?!

Relax.
Means “empty box with label Zero”

What are values of nat?
“Recursive” types

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

What are values of \text{nat}?

Zero

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

Succ
Zero

Succ
Succ
Zero

Succ
Succ
Succ
Zero
“Recursive” types

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

Next: Lets get cosy with Recursion

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

let rec to_int n =

match n with
| Zero   -> 0
| Succ m -> 1 + to_int m
of_int : int → nat

let rec of_int n =

Base pattern
Inductive pattern

Base pattern
Inductive pattern

Base pattern
Inductive pattern

Base pattern
Inductive pattern
plus : nat*nat -> nat

\[
\text{type nat} =
\begin{cases}
\text{Zero} \\
\text{Succ of nat}
\end{cases}
\]

let rec plus (n,m) =

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

Base pattern
Inductive pattern
**times**: nat*nat -> nat

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

let rec times (n,m) =
```

**plus**: nat*nat -> nat

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

let rec times (n,m) =
```

```ocaml
match m with
| Zero    -> Zero
| Succ m' -> plus n (times (n,m'))
```
Next: Let's get cosy with Recursion

Recursive Code Mirrors Recursive Data

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”

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

Some functions on Lists: Length

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

Lists are recursive types!

type int_list =
  Nil
| Cons of int * int_list
Some functions on Lists: Append

```ml
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

\[
\text{type } \text{tree} = \begin{cases} 
\text{Leaf of int} \\
\text{Node of tree*tree} 
\end{cases}
\]

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

Leaf 1

Leaf 2

Leaf 3
Representing Trees

```ocaml
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
```
```ocaml
let rec sum_leaf t = match t with
  | Leaf n -> n
  | Node(t1,t2) -> sum_leaf t1 + sum_leaf t2
```

```ocaml
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:

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

What's 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:

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

What's a ML FUNCTION for EVALUATING expressions?

```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:

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

What's a ML FUNCTION for EVALUATING expressions?

```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
```

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?

```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
```
Random Art from Expressions
PA #2
Build more funky expressions, evaluate them, to produce: 85