What about more complex data?

Many kinds of expressions:
1. Simple
2. Variables
3. Functions

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

\[ t \text{ is a new datatype.} \]

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

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 \( C_2 \)
- \( \ldots \)
- a value of type \( t_n \) placed in a box labeled \( C_n \)

Suppose I wanted …

Attributes:
- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- Email (string)

Creating Values

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

```
# 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 = …
```

One-of types

- We’ve defined a “one-of” type named \( \text{attrib} \)
- Elements are one of:
  - string,
  - int,
  - int*int,
  - float,
  - bool …
- Can create uniform \( \text{attrib} \) lists
- Suppose I want a function to print attribs…

```
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;
```

```
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 tell what's in the 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

Type rules?
• e1, e2, ..., en must have same type
• Which is type of whole expression

match-with is an Expression

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

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
What about “Recursive” types?

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

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

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

Well designed datatype gives strategy

Some functions on Lists: Append

```ocaml
let rec append (l1,l2) = match l1 with
| Nil -> l2
| Cons(_,t) -> Cons(_, append (t,l2))
```

• Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression
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:
- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) = -14.766$

Q: What's a ML datatype for such expressions?

```
let rec eval e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval e1) + (eval e2)
  | Times (e1, e2) -> (eval e1) * (eval e2)
```

Q: What's a ML function for evaluating such expressions?

```
let rec eval e =
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval e1) + (eval e2)
  | Times (e1, e2) -> (eval e1) * (eval e2)
```

Random Art from Expressions

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