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

type t = c1 of t1 | c2 of t2 | ... | cn of tn

The value of type t is either:
A value of type c1 placed in a box labeled c1
Or a value of type c2 placed in a box labeled c2
Or...
Or a value of type cn 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`?

```ocaml
# let a1 = Name "John";;
val a1 : attrib = Name "John"
# 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 `attrib`
- Elements are one of:
  - `string`
  - `int`
  - `int*int*int`
  - `float`
  - `bool`
  - `int*int`
  - `string`

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

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

  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

Benefits of `match-with`

- Simultaneous test-extract-bind
- Compile-time checks for:
  - missed cases: ML warns if you miss a `e` value
  - redundant cases: ML warns if a case never matches

What about “Recursive” types?

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

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

Type rules?
- `e1, e2, ..., en` must have same type
- Which is type of whole expression
What about “Recursive” types?

```plaintext
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
- Cons 3,
- Cons 2,
- Cons 1,
- []

Lists aren’t built-in!

```plaintext
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

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

Some functions on Lists: Append

```plaintext
let rec append (l1,l2) =
```

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: What’s a ML datatype for such expressions?
Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

- \(4.0 \times 2.9 = 6.9\)
- \(3.78 - 5.92 = -2.14\)
- \((4.0 \times 2.9) \times (3.78 - 5.92) = -14.766\)

What's a ML function for evaluating such expressions?

Next: Functions

Expressions \(\xrightarrow{f} \text{Values}\)

Q: What's the value of a function?

Functions

Two questions about function values:

What is the value:

1. ... of a function?
2. ... of a function “application” (call)?

Values of functions: Closures

- “Body” expression not evaluated until application
- but type-checking takes place at compile time
- i.e. when function is defined
- Function value =
  - <code> + environment at definition>
  - “closure”

```
# let x = 2+2;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x ;;
val x : int = 8
# f 2;;
val 10 : int = 10
```

Binding used to eval \(f \ldots\)

<table>
<thead>
<tr>
<th>(x)</th>
<th>(4) : int</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>2</td>
</tr>
<tr>
<td>(z)</td>
<td>3</td>
</tr>
</tbody>
</table>
Values of function application

Application: fancy word for "call"

• “apply” the argument \( a_2 \) to the (function) \( a_1 \)

Application Value:
1. Evaluate \( a_1 \) in current env to get (function) \( v_1 \)
   - \( v_1 \) is code + env
   - code is (formal \( x \) + body \( e \)), env is \( E \)
2. Evaluate \( a_2 \) in current env to get (argument) \( v_2 \)
3. Evaluate body \( e \) in env \( E \) extended by binding \( x \) to \( v_2 \)

Example 1

```ocaml
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

Example 2

```ocaml
let x = 1;;
let f y = let x = 2 in fun z -> x + y + z;;
let x = 100;;
let g = (f 4);;
let y = 100;;
g 1;;
```

Example 3

```ocaml
let f g =
  let x = 0 in
g 2
;;
let x = 100;;
let h y = x + y;;
f h;;
```
Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment

- Static/Lexical: Determined from the program text
  - Without executing the program

- Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence

Alternative: dynamic scoping

```
let x = 100
let f y = x + y
let g x = f 0
let z = g 0
(* value of z? *)
```