**Datatypes**

**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

**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

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 \( t_1 \) placed in a box labeled \( C_1 \)
Or
• a value of type \( t_2 \) placed in a box labeled \( C_2 \)
Or
• ... 
Or
• a value of type \( t_n \) placed in a box labeled \( C_n \)

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

```
# let a1 = Name "Bob";;
val a1 : attrib = Name "Bob"
# let a2 = Age 5.83;;
val a2 : attrib = Age 5.83
# let a3 = DOB (9,8,1977) ;;
val a3 : attrib = DOB (9,8,1977)
# let a1 = [a1,a2,a3];;
val a1 : 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;

All have type attrib

**One-of types**
- We’ve defined a “one-of” type named attrib
- Elements are one of:
  - string,
  - int,
  - int*int,
  - float,
  - bool ...
- Can create uniform attrib lists
- Say I want a function to print attribs...

**How to TEST & TAKE whats in box?**

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

Look at TAG!

**How to tell whats in the box ?**

<table>
<thead>
<tr>
<th>Pattern-match expression: check if e is of the form ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>On match:</td>
</tr>
<tr>
<td>- value in box bound to pattern variable</td>
</tr>
<tr>
<td>- matching result expression is evaluated</td>
</tr>
<tr>
<td>- Simultaneously test and extract contents of box</td>
</tr>
</tbody>
</table>

**How to tell whats in the box ?**

<table>
<thead>
<tr>
<th>Pattern-match expression: check if e is of the form ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>On match:</td>
</tr>
<tr>
<td>- value in box bound to pattern variable</td>
</tr>
<tr>
<td>- matching result expression is evaluated</td>
</tr>
<tr>
<td>- Simultaneously test and extract contents of box</td>
</tr>
</tbody>
</table>
How to tell what's in the box

```ocaml
# match (Name "Bob") with
  | Name s -> printf "Hello %s\n" s
  | Age i -> printf "%d years old\n" i

Hello Bob
- : unit = ()
```

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

---

How to TEST & TAKE what's in box?

**BEWARE!!**

Be sure to handle all TAGS!

---

Beware! Handle All TAGS!

```ocaml
# 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
# 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!

---

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 U: this match case is unused.
```

Compile-time checks for:
redundant cases: ML warns if a case never matches
Another Few Examples

```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
| Age i -> Printf.printf "%d" i

Warning U: this match case is unused.
```

See code text file

match-with is an Expression

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

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

<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>match e with</td>
</tr>
<tr>
<td>Name a -&gt; e1</td>
</tr>
<tr>
<td>Age i -&gt; e2</td>
</tr>
<tr>
<td>DOB (m, d, y) -&gt; e3</td>
</tr>
<tr>
<td>Address a -&gt; e4</td>
</tr>
<tr>
<td>Height h -&gt; e5</td>
</tr>
<tr>
<td>Alive b -&gt; e6</td>
</tr>
<tr>
<td>Phone (s, n) -&gt; e7</td>
</tr>
<tr>
<td>Email e -&gt; e8</td>
</tr>
<tr>
<td>T</td>
</tr>
</tbody>
</table>

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

Next: Building datatypes

Three key ways to build complex types/values

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

2. **“One-of”** types \( \text{type } t = \text{C1 of } t_1 | \text{C2 of } t_2 \)
   Value of \( T \) contains value of \( T_1 \) or a value of \( T_2 \)

3. **“Recursive”** type
   Value of \( T \) contains (sub)-value of same type \( T \)

“Recursive” types

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

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

Wait a minute! \textbf{Zero} of what?!

Relax. Means “empty box with label \textbf{Zero}”

What are values of \textbf{nat}?

What are values of \textbf{nat}?

One \textbf{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!

Next: Building datatypes
Three key ways to build complex types/values

1. “Each-of” types \( t_1 \times t_2 \)
   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 \text{type } t = \ldots \mid C \text{ of } (\ldots \times t)
   Value of T contains (sub)-value of same type T

Next: Let's get cosy with Recursion
Recursive Code Mirrors Recursive Data

Next: Let's get cosy with Recursion
Code Structure = Type Structure!!!
to_int : nat -> int

let rec to_int n =

of_int : int -> nat

let rec of_int n =

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

\[
\text{let rec plus n m = }
\]

\[
\text{match } m \text{ with}
\]
\[
| \text{Zero} \rightarrow n
\]
\[
| \text{Succ } m' \rightarrow \text{Succ (plus } n m'\text{)}
\]\n
+ Base pattern
+ Inductive pattern
+ Base pattern
+ Inductive pattern

\[
\text{let rec plus n m = }
\]

\[
\text{match } m \text{ with}
\]
\[
| \text{Zero} \rightarrow n
\]
\[
| \text{Succ } m' \rightarrow \text{Succ (plus } n m'\text{)}
\]\n
-times: nat*nat -> nat

\[
\text{let rec times n m = }
\]

\[
\text{match } m \text{ with}
\]
\[
| \text{Zero} \rightarrow n
\]
\[
| \text{Succ } m' \rightarrow \text{Succ (times } n m'\text{)}
\]\n
+ Base pattern
+ Inductive pattern
+ Base pattern
+ Inductive pattern

\[
\text{let rec times n m = }
\]

\[
\text{match } m \text{ with}
\]
\[
| \text{Zero} \rightarrow n
\]
\[
| \text{Succ } m' \rightarrow \text{Succ (times } n m'\text{)}
\]\n
+ Base pattern
+ Inductive pattern
+ Base pattern
+ Inductive pattern
Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

Lists are recursive types!

type int_list =
  Nil | Cons of int * int_list

Think about this! What are values of int_list?

<table>
<thead>
<tr>
<th>Cons(1,Cons(2,Nil))</th>
<th>Cons(2,Nil)</th>
<th>Cons(3,Nil)</th>
<th>Nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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)

No binding for head

Pattern-matching in order
Some functions on Lists: Append

\[
\text{let rec append (l1, l2) } = \\
\]

- Find the right induction strategy
  - Base case: pattern + expression
  - Induction case: pattern + expression

Well designed datatype gives strategy

Some functions on Lists: Max

\[
\text{let rec max xs } = \\
\]

- 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: Let's get cozy with Recursion

Recursive Code Mirrors Recursive Data

Representing Trees

1 2 3

Leaf

\[
\text{type tree } = \\
| \text{Leaf of int} \\
| \text{Node of tree*tree} \\
\]

Representing Trees

1 2 3

Leaf

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

Next: Let's 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

let rec sum_leaf t =
```

```ocaml
```
sum_leaf: tree -> int

```ml
let rec sum_leaf t =
```

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) * (3.78 - 5.92)

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

Whats 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

Whats a ML TYPE for REPRESENTING expressions?

```plaintext
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 ===> 6.9
• 3.78 - 5.92 ===> -2.14
• (4.0 + 2.9) * (3.78 - 5.92) ===> -14.766

Whats a ML FUNCTION for EVALUATING expressions?

```plaintext
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 ===> 6.9
• 3.78 - 5.92 ===> -2.14
• (4.0 + 2.9) * (3.78 - 5.92) ===> -14.766

Whats a ML FUNCTION for EVALUATING expressions?

```plaintext
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
```