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

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

t is a new datatype.
A value of type t is either:
- a value of type t1 placed in a box labeled C1
- a value of type t2 placed in a box labeled C2
- ... Or
- a value of type tn 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;

How to PUT values into box?

let a1 = Name "Bob";;
val a1 : 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 a_l : attrib list = ...

How to create values of type attrib?

let a1 = Name "Bob";;
val a1 : 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 a_l : attrib list = ...

Constructing Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

| Label=C1 | Value:t1 OR Label=C2 | Value:t2 OR Label=Cn | Value:tn

All have the type t
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 what’s in box?

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

How to tell what’s in the box?

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

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

```ocaml
# match (Name "Bob") with
  | Name s -> printf "Hello %s
" s
  | Age i -> printf "%d years old" i
  |;
Hello Bob
- : unit = ()
```

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

---

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
# 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.
```

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 \( T \)

**Benefits of match-with**
1. Simultaneous test-extract-bind
2. Compile-time checks for:
   - missed cases: ML warns if you miss a case
   - redundant cases: ML warns if a case never matches

Next: Building datatypes

Three key ways to build complex types/values

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

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

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

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

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

What are values of \text{nat}?

One \text{nat} contains another!
"Recursive" types

What are values of nat?

One 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 $type \ t = C_1 of t_1 \mid C_2 of t_2$
   Value of T contains value of T1 or a value of T2

3. "Recursive" type $type \ t = \ldots \mid C of (...*t)$
   Value of T contains (sub)-value of same type T

Next: Lets get cosy with Recursion

Recursive Code Mirrors Recursive Data

to_int : nat -> int

let rec to_int n =
to_int : nat -> int

let rec to_int n =

type nat =
  | Zero
  | Succ of nat

of_int : int -> nat

let rec of_int n =

type nat =
  | Zero
  | Succ of nat

of_int : int -> nat

let rec of_int n =

type nat =
  | Zero
  | Succ of nat

of_int : int -> nat

let rec of_int n =

type nat =
  | Zero
  | Succ of nat

of_int : int -> nat

let rec of_int n =

type nat =
  | Zero
  | Succ of nat
plus : nat*nat -> nat

\[
\text{let rec plus n m =} \\
\text{match m with} \\
\text{Zero ->} \text{Zero} \\
\text{Succ m' ->} \text{Succ (plus n m')} \\
\]
Next: Lets 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”

Some functions on Lists : Length

<table>
<thead>
<tr>
<th>let rec len l</th>
<th>match l with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>0</td>
</tr>
<tr>
<td>Cons h t</td>
<td>1 + (len t)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>let rec len l</th>
<th>match l with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>0</td>
</tr>
<tr>
<td>Cons h t</td>
<td>1 + (len t)</td>
</tr>
</tbody>
</table>

No binding for head

Pattern-matching in order
Some functions on Lists: Append

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

- 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

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

1
2
3
Leaf

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

1
2
3
Leaf
Representing Trees

```ocaml
type tree = | Leaf of int | 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.

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

```
let rec sum_leaf t =
```

Another Example: Calculator

Want an arithmetic calculator to evaluate expressions like:

- \(4.0 + 2.9\)
- \(3.78 - 5.92\)
- \((4.0 + 2.9) \times (3.78 - 5.92)\)

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

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

What's a ML FUNCTION for EVALUATING expressions?

```
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

What's a ML FUNCTION for EVALUATING expressions?

```
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
```