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

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

\( t \) 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 \( C_n \)

Suppose I wanted ...

Attributes:

• Name (string)
• Age (integer)
• DOB (int-int-int)
• Address (string)
• Height (real)
• Alive (boolean)
• Phone (int-int)
• email (string)

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

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 \( \text{attrib} \)?

\[ \text{type } \text{attrib} = \text{Name of string} | \text{Age of int} | \text{DOB of int*int*int} | \text{Address of string} | \text{Height of float} | \text{Alive of bool} | \text{Phone of int*int} | \text{Email of string}; \]
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*int,
  - float,
  - bool ...
- Can create uniform `attrib` lists
- Say I want a function to print `attrib`...

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?

```go
match e with
| Name s -> printf "%s" s
| Age i -> printf "%d" i
| DOB (d,m,y) -> printf "%d/%d/%d" d m y
| Address a -> printf "%s" a
| Height h -> printf "%.4f" h
| Alive b -> printf "%b" b
| Phone(a,r) -> printf "(%d) - %d" a r
```

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!

Beware! Handle All TAGS!

```ocaml
# match (Name "Bob") with
  | Age i -> printf "%d" i
  | Email s -> 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 (_, _)
```

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

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

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

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

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

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

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

Type Rule
- `e1, e2,...,en` 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 `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
   Value of `T` contains (sub)-value of same type `T`

“Recursive” types

```
type nat = Zero | Succ of nat
```
"Recursive" types

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

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!

What are values of \text{nat}?

One \text{nat} contains another!
“Recursive” types

type nat = Zero | Succ of nat

What are values of nat?
One nat contains another!

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

to_int : nat -> int

let rec to_int n =
```ml
let rec to_int n =
  type nat =
  | Zero
  | Succ of nat

  match n with
  | Zero -> 0
  | Succ m -> 1 + to_int m

let rec of_int n =
  type nat =
  | Zero
  | Succ of nat

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

let rec plus n m =

Base pattern
Inductive pattern

Base pattern
Inductive pattern

match m with
| Zero      ->
| Succ m'   ->

Inductive Expression

plus : nat*nat -> nat

let rec plus n m =

Base pattern
Inductive pattern

Base pattern
Inductive pattern

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

Inductive Expression

times: nat*nat -> nat

let rec times n m =

Base pattern
Inductive pattern

Base pattern
Inductive pattern

match m with
| Zero      ->
| Succ m'   ->

Inductive Expression
times: nat*nat -> nat

let rec times n m = match m with |
    Zero -> Zero |
    Succ m' -> plus n (times n m')

Base pattern
Inductive pattern

Base pattern
Inductive pattern

Lists are recursive types!

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

大涨6 | 1, | 2, | 3, | Nil|

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: 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

\[
\text{type tree =}

\begin{array}{l}
| \text{Leaf of int} \\
| \text{Node of tree*tree}
\end{array}
\]

Leaf 1

Representing Trees

\[
\text{type tree =}

\begin{array}{l}
| \text{Leaf of int} \\
| \text{Node of tree*tree}
\end{array}
\]

Leaf 2
Representing Trees

Next: Lets 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
```
**sum_leaf: tree -> int**

```ml
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) * (3.78 - 5.92)

**Recursive Code Mirrors Recursive Data**

Code almost writes itself!
Another Example: Calculator
Want an arithmetic calculator to evaluate expressions like:
- \(4.0 + 2.9 \rightarrow 6.9\)
- \(3.78 - 5.92 \rightarrow -2.14\)
- \((4.0 + 2.9) \times (3.78 - 5.92) \rightarrow -14.766\)
What's a ML TYPE for REPRESENTING expressions?

```ocaml
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:
- \(4.0 + 2.9 \rightarrow 6.9\)
- \(3.78 - 5.92 \rightarrow -2.14\)
- \((4.0 + 2.9) \times (3.78 - 5.92) \rightarrow -14.766\)
What's a ML FUNCTION for EVALUATING expressions?

```ocaml
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
```