What about more complex data?

- We've seen some base types and values:
  - Integers, Reals, 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
    - We'll see this in a couple of weeks

Today: 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
     - tuples, records: (2, "ranjit") - int * string, int and a string
     - Today: the power of these concepts

2. "One-of" types
   - Value of T contains value of T1 or a value of T2
     - Lists! : [1,2,3,4,5] int list, but this is: 1:(1,2,3,4,5)
     - some list

3. "Recursive"
   - Value of T contains (sub)-value of same type T
     - Lists

Suppose I wanted ...

... a program that processed lists of attributes

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

but but lists must have uniform type!

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?

Constituting Datatypes

datatype 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 t1 placed in a box labeled C1
- a value of type t2 placed in a box labeled C2
- or ... 
- or a value of type Cn placed in a box labeled Cn

Labels called datatype "constructors"

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

SHOW CODE

How to create values of type attrib?

- val a1 = Name "Ranjit";
- val a2 = Height 5.83;
- val a3 = DOB (9, 8, 1977);
- val a4 = [a1, a2, a3];

One-of types

- We've defined a "one-of" type named attrib
- Elements are one of:
  - String,
  - int,
  - int*int,
  - real,
  - bool...
- Can create uniform attrib lists
- Suppose I want a function to print attribs...

SHOW PRINT CODE

Case-of is an Expression

- In to "Ranjit"
  - e1 evaluated (with binding)
  - e3 evaluated (with binding)
  - e5 evaluated (with binding)

Pattern-match expression: check if e is of the form ...
- On match:
  - value in box bound to pattern variable
  - corresponding result expression is evaluated
- Simultaneously test and extract contents of box

Benefits of case-of

1. Simultaneous test-extract-bind
2. Compile-time checks for:
   - missed cases: ML warns if you miss a `case` value
   - redundant cases: ML warns if a case never matches
What about “Recursive” types?

```haskell
datatype int_list = Nil
  | Cons of int * int_list
```

Think about this! What are values of \texttt{int\_list}?

- \texttt{Nil}
- \texttt{Cons 1 Nil}
- \texttt{Cons 2 (Cons 3 Nil)}
- \texttt{Cons 2 Nil}
- \texttt{Cons 3 Nil}
- \texttt{Cons Nil}
- \texttt{Nil}

Lists aren’t built-in!

```haskell
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

[Syntactic Sugar]

Some functions on Lists : Length

```haskell
fun len l = case l of
  Nil => 0
  | Cons x t => 1 + (len t)
```

• Find the right \textit{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 \textit{all} cases covered
- ML checks no redundant cases
- ...at \textit{compile-time}:
  - fewer errors (crashes) during execution
  - get the bugs out ASAP!

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

```haskell
• 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 datatype for such expressions?

```haskell
datatype expr =
  Real
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
```

SHOW CODE
Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:
- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) \times (3.78 - 5.92) = -14.766$

What's a ML function for evaluating such expressions?

```ml
fun eval e =
  case e of
    Add (e1,e2) => eval e1 + eval e2;
    Sub (e1,e2) => eval e1 - eval e2;
    Mul (e1,e2) => eval e1 * eval e2;
```

Functions

Expressions $\rightarrow$ Values $\rightarrow$ Types

Functioning recursively in ML

Example: Factorial

```ml
fun fac n =
  if n=0
  then 1
  else n * fac (n-1);
```

Example: Clone

```ml
fun clone (x,n) =
  if n=0
  then []
  else x :: clone(x,n-1);
```

Example: Interval

```ml
fun interval (x,j) =
  if j
  then []
  else x :: interval(+1,j);
```
Example: List Maximum

Find maximum element in +ve int list

\[
\text{fun max (x,y) = if x > y then x else y} \\
\text{fun listMax l =}
\]

\[
\begin{align*}
\text{let helpers} (\max, 1) &= \\
&\begin{cases}
\text{if (null l)} & \text{Base Expression} \\
\text{else helpers (max (\_, Ind i), ti l)} & \text{Inductive Expression}
\end{cases} \\
\text{in helpers (0, l)} & \text{Induction Condition}
\end{align*}
\]

Example: List Append

Roll our own @

\[
\text{fun append [([], l)] = 1} \\
| \text{append [h::t, i]} = h::(append (t, i))
\]

Example: List Filter

\[
\text{fun filter [\_, []] = []} \\
| \text{filter [f, h::t] =}
\]

\[
\begin{align*}
\text{let helpers} (f, t) &= \\
&\begin{cases}
\text{if (f h) then h::(f, t) else t} & \text{Base Expression}
\end{cases} \\
\text{in helpers (0, l)} & \text{Induction Expression}
\end{align*}
\]

Some functions on Lists: Append

\[
\text{fun append (11, 12) =}
\]

\[
\begin{align*}
\text{case 11 of} & \\
\text{Nil} & \rightarrow \text{Nil} \\
\text{Cons \_} & \rightarrow \text{Cons Nil (append (t, i))}
\end{align*}
\]

\[
\begin{align*}
\text{case 12 of} & \\
\text{Nil} & \rightarrow \text{Nil} \\
\text{Cons \_} & \rightarrow \text{Cons Nil (append (t, i))}
\end{align*}
\]

\[
\text{case 10 of} \\
\text{Nil} & \rightarrow \text{Nil} \\
\text{Cons \_} & \rightarrow \text{Cons Nil (append (t, i))}
\]

\[
\text{matches everything, no binding} \quad \text{Pattern-matching in order} \\
\text{- Must match with Nil}
\]