What about “Recursive” types?

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

Think about this! What are values of `int_list`?

- `Nil`
- `Cons 3, Nil`
- `Cons 2, Cons 3, Nil`
- `Cons 1, Cons 2, Cons 3, Nil`

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

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

Base Expression

Inductive Expression

Base pattern

Ind pattern

Matches everything, no binding

Pattern-matching in order
- Must match with Nil

Some functions on Lists: listMax

```ocaml
let rec listMax xs = match xs with
  | Nil -> 0
  | Cons(_, t) -> max h (listMax t)
```

Base Expression

Inductive Expression

Base pattern

Ind pattern

Matches everything, no binding

Pattern-matching in order
- Must match with Nil
Some functions on Lists: Append

```plaintext
let rec append (l1, l2) =
  Base Expression
  Inductive Expression
  Base pattern
  Ind pattern
• 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!

Another Example: Calculator

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

**Q:** What's a ML datatype for such expressions?
Code Mirrors Data

Next: Variables

Variables and Bindings

Q: How to use variables in ML?
Q: How to “assign” to a variable?

# let x = 2+2;;
val x : int = 4

let x = e;;

"Bind the value of expression e to the variable x"
Variables and Bindings

```ocaml
# let x = 2+2;;
val x : int = 4
# let y = x * x * x;;
val y : int = 64
# let z = [x;y;x+y];;;
val z : int list = [4;64;68]
```

Later declared expressions can use `x` – Most recent “bound” value used for evaluation

Sounds like C/Java? NO!

Environments (“Phone Book”)

How ML deals with variables
- Variables = “names”
- Values = “phone number”

Environments and Evaluation

ML begins in a “top-level” environment
- Some names bound

```ocaml
let x = e;;
```

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order
1. Evaluate expr `e` in current env to get value `v : t`
2. Extend env to bind `x` to `v : t`
(Repeat with next binding)
### Example

```
# let x = 2+2;;
val x : int = 4

# let y = x * x * x;;
val y : int = 64

# let z = [x;y;x+y];;
val z : int list = [4;64;68]

# let x = x + x ;;
val x : int = 8
```

### Environments

1. **Evaluate**: Use most recent bound value of var
2. **Extend**: Add new binding at end

**How is this different from C/Java’s “store”?**

```
# 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 0;
val it : int = 4
```

**New binding!**

```
# let x = 2+2;;
val : int x = 4

let f = fun y -> x + y;;
val f : int -> int = fn

let x = x + x ;;
val x : int = 8

f 0;
val it : int = 4
```

**Environments**

<table>
<thead>
<tr>
<th>x</th>
<th>4 : int</th>
</tr>
</thead>
</table>

```
# 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 0;
val it : int = 4
```

**Binding used to eval (f ...)**

```
# 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 0;
val it : int = 4
```

**Binding for subsequent x**

```
# 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 0;
val it : int = 4
```
Cannot change the world

Cannot “assign” to variables
• Can extend the env by adding a fresh binding
• Does not affect previous uses of variable

Environment at fun declaration frozen inside fun “value”
• Frozen env used to evaluate application \( f \ldots \)

Q: Why is this a good thing?

A: Function behavior frozen at declaration
• Nothing entered afterwards affects function
• Same inputs always produce same outputs
  – Localizes debugging
  – Localizes reasoning about the program
  – No “sharing” means no evil aliasing

Examples of no sharing

Remember: No addresses, no sharing.
• Each variable is bound to a “fresh instance” of a value

Tuples, Lists ...

• Efficient implementation without sharing?
  • There is sharing and pointers but hidden from you

Compiler’s job is to optimize code
• Efficiently implement these “no-sharing” semantics

• Your job is to use the simplified semantics
  • Write correct, cleaner, readable, extendable systems

Function bindings

Functions are values, can bind using val

\[
\text{let } \text{fname} = \text{fun } x \rightarrow e ;;
\]

Problem: Can’t define recursive functions!
• \text{fname} is bound after computing rhs value
• No (or “old”) binding for occurrences of \text{fname} inside \( e \)

\[
\text{let rec } \text{fname} x = e ;;
\]

Occurrences of \text{name} inside \( e \) bound to “this” definition

\[
\text{let rec } \text{fac} x = \text{if } x<1 \text{ then } 1 \text{ else } x \text{fac} (x-1)
\]
Local bindings

So far: bindings that remain until a re-binding (“global”)
Local, “temporary” variables are useful inside functions
• Avoid repeating computations
• Make functions more readable

Let-in is an expression!

Evaluating let-in in env $E$:
1. Evaluate $expr$ $e_1$ in env $E$ to get value $v : t$
2. Use extended $E[x \mapsto v : t]$ (only) to evaluate $e_2$

let $x = e_1$ in
  $e_2$
;;
### Nested bindings

```plaintext
let
  x = 10
in
let
  y = 20
in
  x * y
;;
```

Correct Formatting

### Example

```plaintext
let rec filter (f, l) = 
  if l = [] then []
  else 
    let h = hd l in 
    let t = filter (f, tl l) in 
    if (f h) then h::t else t
```

### Nested function bindings

```plaintext
let a = 20;;

let f x = 
  let y = 10 in
  let g z = y + z in
    a + (g x)
  
  f 0;
```

### Recap

- **Variables are names for values**
  - Environment: dictionary/phonebook
  - Most recent binding used
  - Entries never changed, new entries added

- **Environment frozen at fun definition**
  - Re-binding variables cannot change a function
  - Same I/O behavior at every call

- **Env frozen with function**
  - Used to evaluate fun application
  - Values in application are those frozen in env at definition
Recap

• Build complex expressions with local bindings
  – let-in expression
  – The let-binding is visible (in scope) inside in-expression
  – Elsewhere the binding is not visible

Static/Lexical Scoping

• For each occurrence of a variable, there is a unique place in program text
  where the variable was 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

Next: Functions

Q: What’s the value of a function?