Evaluating declarations
- Evaluation uses an environment $\rho$
- To evaluate a (simple) declaration $\text{let } x = e$
  - Evaluate expression $e$ in $\rho$ to value $v$
  - Update $\rho$ with $x$ and $v$: $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in $\rho_1$ and all those in $\rho_2$ that are not rebound in $\rho_1$
  $$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}\} + \{y \rightarrow 100, b \rightarrow 6\}$$
  $$= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}, b \rightarrow 6\}$$

Evaluating declarations

Evaluating expressions
- Evaluation uses an environment $\rho$
- A constant evaluates to itself
- To evaluate an variable, look it up in $\rho$: $\rho(v)$
- To evaluate uses of $+$, $-$, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: $\text{let } x = e_1 \text{ in } e_2$
  - Eval $e_1$ to $v$, eval $e_2$ using $\{x \rightarrow v\} + \rho$

Evaluating expressions

Evaluating conditions expressions
- To evaluate a conditional expression: $\text{if } b \text{ then } e_1 \text{ else } e_2$
  - Evaluate $b$ to a value $v$
  - If $v$ is True, evaluate $e_1$
  - If $v$ is False, evaluate $e_2$

Evaluating conditions expressions

Evaluation of Application with Closures
- Given application expression $f(e_1, \ldots, e_n)$
- Evaluate $(e_1, \ldots, e_n)$ to value $(v_1, \ldots, v_n)$
- In environment $\rho$, evaluate left term to closure, $c = \langle(x_1, \ldots, x_n) \rightarrow b, \rho'\rangle$
  - $(x_1, \ldots, x_n)$ variables in (first) argument
- Update the environment $\rho'$ to $\rho'' = \{x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n\} + \rho'$
- Evaluate body $b$ in environment $\rho''$

Evaluation of Application with Closures

Evaluation of Application of $\text{plus}_x$;
- Have environment: $\rho = \{\text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, \ldots, y \rightarrow 3, \ldots\}$
  where $\rho_{\text{plus}_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}$
- Eval $\text{plus}_x(y, \rho)$ rewrites to
- App $\text{Eval}(\text{plus}_x, \rho), \text{Eval}(y, \rho))$ rewrites to
- App $\text{Eval}(\text{plus}_x, \rho), 3)$ rewrites to
- App $\langle y \rightarrow y + x, \rho_{\text{plus}_x}, 3\rangle$ rewrites to
  ...
Evaluation of Application of `plus_x`;

Have environment:

\[ \rho = \{ \text{plus}_x \mapsto <y \mapsto y + x, \rho_{\text{plus}_x}>, \ldots, x \mapsto 3, \ldots \} \]

where \( \rho_{\text{plus}_x} = \{ x \mapsto 12, \ldots, y \mapsto 24, \ldots \} \)

- App \((<y \mapsto y + x, \rho_{\text{plus}_x}>, 3)\) rewrites to
- Eval \((y + x, \{y \mapsto 3\} + \rho_{\text{plus}_x})\) rewrites to
- Eval \((\rho_{\text{plus}_x}) + 12\) rewrites to
- \(3 + 12 = 15\)

Evaluation of Application of `plus_pair`

Assume environment

\[ \rho = \{ x \mapsto 3\ldots, \text{plus}_\text{pair} \mapsto <n,m) \mapsto n + m, \rho_{\text{plus}_\text{pair}}\} + \rho_{\text{plus}_\text{pair}} \]

- App \((\text{Eval} (\text{plus}_\text{pair}, \rho), (4,3))\) =
- \(\text{App} (\{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) =
- \) Eval \((4, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) =
- \) Eval \((3, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) = 4 + 3 = 7\)

Evaluation of Application of `plus_pair`

Assume environment

\[ \rho = \{ x \mapsto 3\ldots, \text{plus}_\text{pair} \mapsto <(n,m) \mapsto n + m, \rho_{\text{plus}_\text{pair}}\} + \rho_{\text{plus}_\text{pair}} \]

- App \((\text{Eval} (\text{plus}_\text{pair}, \rho), (4,3))\) =
- \(\text{App} (\{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) =
- \) Eval \((4, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) =
- \) Eval \((3, \{n \mapsto 4, m \mapsto 3\} + \rho_{\text{plus}_\text{pair}}) = 4 + 3 = 7\)

Evaluation of Curried Functions

Assume \(\rho_{\text{add}_\text{three}}\) is the environment when `add_three` is defined, and \(\rho\) comes after `add_three` is define.

Recall:

\[ \text{let add}_\text{three} x y z = x + y + z;\]
\[ \text{val add}_\text{three} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{<fun>}\]
\[ \# \text{let } t = \text{add}_\text{three} 6 3 2;\]

- Eval \(((\text{add}_\text{three} 6) 3) 2, \rho)\) =
- \(\text{App} (\text{Eval} (6, \rho), (\text{Eval}(4, \text{rho}), \text{Eval}(x, \rho)))\) =
- \(\text{App} (\text{Eval} (\text{add}_\text{three} 6) 3, \rho), 2) =
- \(\text{App} (\text{Eval} (\text{add}_\text{three} 6) 3, \rho)), 2) =\)

Evaluation of `add_three 6 3 2`

\[ \rho = \{ x \mapsto 3\ldots, \text{plus}_\text{pair} \mapsto <(n,m) \mapsto n + m, \rho_{\text{plus}_\text{pair}}\} + \rho_{\text{plus}_\text{pair}} \]

- App \((\text{App} (\text{App} (\text{Eval} (\text{add}_\text{three} 6) 3, \rho), \text{Eval}(6, \rho)), 3) 2) =\)
- \(\text{App} (\text{App} (\text{Eval} (\text{add}_\text{three} 6) 3, \rho), \text{Eval}(6, \rho)), 3) 2) =\)
- \(\text{App} (\text{Eval} (\text{add}_\text{three} 6) 3, \rho)), 2) =\)
- \(\text{App} (\text{Eval}(6, \rho)), 3) 2) =\)

- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
- \(\text{Eval}(\text{add}_\text{three} 6) 3, \rho) =\)
Evaluation of `add_three 6 3 2`
- \(\text{Eval}(x + y, \{z \to 2, y \to 3, x \to 6\} + \text{radd_three}) + \text{Eval}(z, \{z \to 2, y \to 3, x \to 6\} + \text{radd_three}) = (6+3)+2 = 9 + 2 = 11\)

Recursive Functions
- \# let rec factorial n =
  \[
    \begin{align*}
      & \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{factorial} \ (n - 1);; \\
      & \text{val factorial : int -> int = <fun>}
    \end{align*}
  \]
- \# factorial 5;;
  - : int = 120
- \# (* rec is needed for recursive function declarations *)

Recursion Example
- Compute \(n^2\) recursively using:
  \[
    n^2 = (2 \times n - 1) + (n - 1)^2
  \]
- \# let rec nthsq n =
  \[
    \begin{align*}
      & \text{match } n \text{ with} \\
      & 0 \rightarrow 0 \\
      & n \rightarrow (2 \times n - 1) + \text{nthsq} \ (n - 1);
    \end{align*}
  \]
  \[
    \text{val nthsq : int -> int = <fun>}
  \]
  - : int = 9

Recursion and Induction
- \# let rec nthsq n = match n with 0 -> 0
  \[
    \begin{align*}
      & \mid n \rightarrow (2 \times n - 1) + \text{nthsq} \ (n - 1);
    \end{align*}
  \]
  - Base case is the last case; it stops the computation
  - Recursive call must be to arguments that are somehow smaller - must progress to base case
  - \text{if or match} must contain base case
  - Failure of these may cause failure of termination

Lists
- List can take one of two forms:
  - Empty list, written \[
    \[
  \]
  - Non-empty list, written \[
    x :: xs
  \]
  - x is head element, xs is tail list, :: called “cons”
  - Syntactic sugar: \[
    [x] == x :: [ ]
  \]
  - \[
    [x1; x2; ...; xn] == x1 :: x2 :: ... :: xn :: [ ]
  \]
- \# let fib5 = [8;5;3;2;1];;
  val fib5 : int list = [8; 5; 3; 2; 1]
- \# let fib6 = 13 :: fib5;;
  val fib6 : int list = [13; 8; 5; 3; 2; 1]
- \# (8::5::3::2::1::1::[ ]) = fib5;;
  - : bool = true
- \# fib5 @ fib6;;
  - : int list = [8; 5; 3; 2; 1; 13; 8; 5; 3; 2; 1; 1]
Lists are Homogeneous

# let bad_list = [1; 3.2; 7];;
Characters 19-22:
    let bad_list = [1; 3.2; 7];;
       ^^^
This expression has type float but is here used with type int

Question

- Which one of these lists is invalid?
  1. [2; 3; 4; 6]
  2. [2,3; 4,5; 6,7]
  3. [(2.3,4); (3.2,5); (6,7.2)]
  4. [['hi'; "there"] ; ['wahcha']; [ ]; ['doin']]

Answer

- Which one of these lists is invalid?
  1. [2; 3; 4; 6]
  2. [2,3; 4,5; 6,7]
  3. [(2.3,4); (3.2,5); (6,7.2)]
  4. [['hi'; "there"] ; ['wahcha']; [ ]; ['doin']]

  3 is invalid because of last pair

Functions Over Lists

# let rec double_up list =
    match list
    with [] -> []  (* pattern before ->, expression after *)
          | (x :: xs) -> poor_rev xs @ [x];;
val double_up : 'a list -> 'a list = <fun>

poor_rev silly;;

-: string list = ["there"; "there"; "hi"; "hi"]

Functions Over Lists

# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]

# let rec poor_rev list =
    match list
    with [] -> []
          | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

poor_rev silly;;

-: string list = ["there"; "there"; "hi"; "hi"]

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function
Question: Length of list

- Problem: write code for the length of the list
  - How to start?

let length l = 

9/7/21  25

Question: Length of list

- Problem: write code for the length of the list
  - How to start?

let rec length l =
  match l with

9/7/21  26

Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

let rec length l =
  match l with
  | [] ->
  | (a :: bs) ->

9/7/21  27

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when l is empty?

let rec length l =
  match l with
  | [] -> 0
  | (a :: bs) ->

9/7/21  28

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when l is not empty?

let rec length l =
  match l with
  | [] -> 0
  | (a :: bs) ->

9/7/21  29

9/7/21  30
Question: Length of list

Problem: write code for the length of the list

What result do we give when $l$ is not empty?

```ocaml
let rec length l =
  match l with
    | [] -> 0 (* Nil case *)
    | (a :: bs) -> 1 + length bs
```

Structural Recursion: List Example

```ocaml
# let rec length list = match list
  with [] -> 0 (* Nil case *)
  | x :: xs -> 1 + length xs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

Nil case [] is base case

Cons case recurses on component list xs

Same Length

How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
  match list1 with
    | [] -> true
    | (y :: ys) -> false
    | (x :: xs) -> (match list2 with
            | [] -> false
            | (y :: ys) -> same_length xs ys)
```

Higher-Order Functions Over Lists

```ocaml
# let rec map f list =
  match list
    with [] -> []
    | (h :: t) -> (f h) :: (map f t);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
```

Recursing over lists

```ocaml
# let rec fold_right f list b =
  match list
    with [] -> b
    | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# fold_right
  (fun s -> fun () -> print_string s)
  "hi";;
val therehi : unit = ()
```

The Primitive Recursion Fairy
Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

Forward Recursion: Examples

```ocaml
# let rec double_up list = match list
  with [ ] -> [ ] | (x :: xs) -> (x :: x :: double_up xs);
val double_up : 'a list -> 'a list = <fun>
```

```ocaml
# let rec poor_rev list = match list
  with [ ] -> [ ] | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Encoding Forward Recursion with Fold

```ocaml
# let rec append list1 list2 = fold_right (fun x y -> x :: y) list1 list2;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list = List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
```

```
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list = List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
```

```
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Folding Recursion

Another common form “folds” an operation over the elements of the structure

```ocaml
let rec multList list = match list with
| [] -> 1
| x::xs -> x * multList xs
val multList : int list -> int = <fun>

# multList [2;4;6];;
- : int = 48
```

Folding Functions over Lists

How are the following functions similar?

```ocaml
let rec sumList list = match list with
| [] -> 0
| x::xs -> x + sumList xs
val sumList : int list -> int = <fun>

# sumList [2;3;4];;
- : int = 9

let rec prodList list = match list with
| [] -> 1
| x::xs -> x * prodList xs
val prodList : int list -> int = <fun>

# prodList [2;3;4];;
- : int = 24
```

How long will it take?

- Remember the big-O notation from CS 225 and CS 374
- Question: given input of size $n$, how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power
How long will it take?

Common big-O times:
- Constant time $O(1)$
- input size doesn’t matter
- Linear time $O(n)$
- double input $\Rightarrow$ double time
- Quadratic time $O(n^2)$
- double input $\Rightarrow$ quadruple time
- Exponential time $O(2^n)$
- increment input $\Rightarrow$ double time

Linear Time
- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList, append`
- Integer example: `factorial`

Quadratic Time
- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:
  ```ml
  # let rec poor_rev list = match list with [] -> [] | (x::xs) -> poor_rev xs @ [x];;
  val poor_rev : 'a list -> 'a list = <fun>
  ```

Exponential running time
- Poor worst-case running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

Exponential running time
  ```ml
  # let rec slow n =
  if n <= 1 then 1
  else 1+slow (n-1) + slow(n-2);;
  val slow : int -> int = <fun>
  
  # List.map slow [1;2;3;4;5;6;7;8;9];;
  - : int list = [1; 3; 5; 9; 15; 25; 41; 67; 109]
  ```

An Important Optimization
- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?
An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.
- What if \( f \) calls \( g \) and \( g \) calls \( h \), but calling \( h \) is the last thing \( g \) does (a tail call)?
- Then \( h \) can return directly to \( f \) instead of \( g \).

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls.
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls.
- Tail recursion generally requires extra “accumulator” arguments to pass partial results.
- May require an auxiliary function.

Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

Comparison

```ocaml
poor_rev [1,2,3] =
(poor_rev [2,3]) @ [1] =
((poor_rev [3]) @ [2]) @ [1] =
(((poor_rev [ ]) @ [3]) @ [2]) @ [1] =
(( [ ] @ [3]) @ [2]) @ [1]) =
([3] @ [2]) @ [1] =
(3:: ([ ] @ [2])) @ [1] =
[3,2] @ [1] =
3 :: ([2] @ [1]) =
3 :: (2:: ([ ] @ [1])) = [3, 2, 1]
```

Comparison

```ocaml
rev [1,2,3] =
rev_aux [1,2,3] [ ] =
rev_aux [2,3] [1] =
rev_aux [3] [2,1] =
rev_aux [ ] [3,2,1] = [3,2,1]
```

Folding - Tail Recursion

```ocaml
- # let rev list =
-   fold_left
-   (fun l -> fun x -> x :: l)  //comb op
-   []  //accumulator cell
-   list
```
Iterating over lists

```ocaml
# let rec fold_left f a list = match list with [] -> a | (x :: xs) -> fold_left f (f a x) xs;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```ocaml
# fold_left (fun () -> print_string) ()
```

```

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
- Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition
```