1. Evaluation
Environments: typographical conventions vs implementation

• When in the handout we write something like \([x \rightarrow 5, + \rightarrow \text{builtin-addn}]\)

• …in ocaml, we’ll represent this as a list in which the last item in the representation above is the first in the list, i.e., we’ll have

\[
[(\text{ID } "+", \text{plus}_\text{func}); (\text{ID } "x", \text{Vnum } 5)]
\]

• If we want to create a new envt with one more binding, say, \([x \rightarrow 5, + \rightarrow \text{builtin-addn}, y \rightarrow 2]\)

• …we’ll “cons” this new binding on to the existing list, preserving our “backwards order” thing:

\[
[(\text{ID } "y", \text{Vnum } 2); (\text{ID } "+", \text{plus}_\text{func}); (\text{ID } "x", \text{VNum } 5)]
\]
Review from last time

• Definition-handling:
  • Input: definition, environment
    • Definition: identifier and expression
  • Output: \textit{new} environment, containing all bindings from input envt, and one more.
    • Recall: envt is a binding list
    • Binding is an (identifier, \textit{value}) pair
  • This would be easy \textbf{if} the “definition” in the input were an identifier and a \textit{value}:
    
    ```
    let function handle_def definition * environment -> environment = function
    ((Id name, Value v), env) -> (Id name, Value v)::env ;;
    ```
    • Alas, we have an \textit{expression} instead of a \textit{value}.
The converter from expressions to values

- ...is called “evaluation”
- It’s not quite that simple: it converts an expression AND an environment into the value of that expression, given that environment.
- Or to be honest: two environments
  - top-level
  - local
Evaluation

• This is the tough part

• Involves recursion

• Has many steps: divide and conquer by doing one thing at a time
  • Write helper functions!

• Basic template:
let eval : environment * environment * expression -> value = function (t1e, env, expr) -> match expr with
  | NumE n -> ...
  | IdentE name -> ...
  | AndE (e1, e2) -> ...
  | ...
let eval : environment * environment * expression -> value =
function (tle, env, expr) -> match expr with
  | NumE n -> VNum n
  | IdentE name -> ...
  | AndE (e1, e2) -> ...
  | ...
A contrived example: “simplify” an integer

• Try to divide it by 2 (with no remainder)
• Try to divide it by 3 (with no remainder)
• Give up.
Explicit
(*
simplify: int -> int option
Input: a positive integer, n
Output:
  Some n/2, if n is a multiple of two (i.e., even)
  Some n/3, if n is odd but a multiple of three
  None, if n is odd, but not a multiple of 3.
Examples:
simplify 12  -> Some 6
simplify 15  -> Some 5
simplify 19  -> None
*)
Helper: simplify_2

(*
simplify_2: int -> int option
Input: a positive integer, n
Output:
  Some n/2, if n is a multiple of two (i.e., even)
  None, if n is odd
Examples:
simplify 12 -> Some 6
simplify 15 -> None
*)

let simplify_2: int -> int option = function
  n -> if (n mod 2) == 0 then Some (n/2) else None ;;
Helper: simplify_3

(*
  simplify_3: int -> int option
Input: a positive integer, n
Output:
  Some n/3, if n is a multiple of three
  None, if n is not a multiple of three
Examples:
simplify 12 -> Some 4
simplify 16 -> None
*)

let simplify_3: int -> int option = function
  n -> if (n mod 3) == 0 then Some (n/3) else None ;;
Now let’s write “simplify”

• If divisible by 2, then divide by 2
• If not, then try dividing by 3
• If you can’t, then answer is “None”.
(*)
simplify: int -> int option
Input: a positive integer, n
Output:
    Some n/2, if n is a multiple of two (i.e., even)
    Some n/3, if n is odd but a multiple of three
    None, if n is odd, but not a multiple of 3.
*)
let simplify: int -> int option = function
    n -> ...
let simplify: int -> int option = function
  n -> ...

I should try simplify_2, and if it works, return that value. If not, I’ll have to do something else...
let simplify: int -> int option = function
  n -> if (simplify_2 n) = None
      then
          (something goes here)
      else
          simplify_2 n
let simplify: int -> int option = function
  n -> if (simplify_2 n) = None
    then
      if (simplify_3 n) = None
        then
          None
        else
          simplify_3 n
      else
        simplify_2 n
    else
      simplify_2 n
Test it!

# simplify 12;;
- : int option = Some 6
# simplify 15;;
- : int option = Some 5
# simplify 19;;
- : int option = None
#
Let’s “polish” the code
let simplify: int -> int option = function
  n -> if (simplify_2 n) = None
      then
        if (simplify_3 n) = None
        then
          None
        else
          simplify_3 n
      else
        simplify_2 n
  • Computes “simplify_3 n” twice
  • Regardless of the result, it returns the result!
let simplify: int -> int option = function
  n -> if (simplify_2 n) = None
      then
        simplify_3 n
      else
        simplify_2 n
let simplify: int -> int option = function
  n -> if (simplify_2 n) = None
    then
      simplify_3 n
    else
      simplify_2 n
• Computes “simplify_2 n” twice (sometimes)
• We could record the value and reuse it!
let simplify: int -> int option = function
    n -> let result = simplify_2 n in
        if result = None
        then
            simplify_3 n
        else
            result

• The “if” clause tests which of two kinds of “option” we’ve got
• We could do this with matching instead
let simplify: int -> int option = function
  n -> let result = simplify_2 n in
  match result with
   | None -> simplify_3 n
   | _ -> result
A student suggestion for even better code:

```
let simplify: int -> int option = function
    n -> match simplify_2 n with
        | None -> simplify_3 n
        | Some k -> Some k
```
Back to evaluation

• Sometimes (when evaluating a SymbolE, for instance) the “rules” say to look up the value
  • In the local environment
  • ...and if it’s not found there, then in the top-level environment

• Suppose you’ve already written

let lookup: expression * environment -> value option =

• If the expression’s not a SymbolE, it’s an error
• If it’s a symbol, and defined in the environment, then return Some v, where v is the value bound to the symbol
• If it’s not defined in the environment, then returns None.
• Suppose you’ve already written and tested
lookup: expression * environment -> value option
• Now write
lookup2: expression * environment * environment ->
value option
...which looks up the symbol in the first environment, and if that fails, looks it up in the second.
Quiz: write

let lookup2: expression * environment * environment -> value option = function
  (expr, local, tle) -> ... fill in code here ...

BIG HINT:
let simplify: int -> int option = function
  n -> let result = simplify_2 n in
    match result with
    | None -> simplify_3 n
    | _  -> result
Back to evaluation in Rackette!
let eval : environment * environment * expression -> value =
    function
(tle, env, expr) -> match expr with
    | NumE n -> VNum n
    | IdentE name -> ...
    | AndE (e1, e2) -> ...
    | ...
    ;;
let eval : environment * environment * expression -> value =
function
(tle, env, expr) -> match expr with
  | NumE n -> VNum n
  | IdentE name -> lookup2 ...
  | AndE (e1, e2) -> ...
  | ...
;;
What just happened?

• We took a complex problem, evaluation
• Broke it down into parts via a “match” expression
• Dealt with one case at a time
• When a case looked complex, we wrote a helper!
Same deal with AndE

• Write a little helper function, “handle_and” to process the two expressions that constitute your and-expression
A small problem

• The code structure looks like this now:
  let rec eval: ... = function
      match expr with
      ... 
      | AndE (exp1, exp2) -> handle_and(exp1, exp2) ...
  let handle_and: ... = function
      (exp1, exp2) -> ... (eval exp1) ... (eval exp2);
• As Ocaml reads the definition of “eval”, it doesn’t know the type of “handle_and”: error!
• If we move handle_and to top, then it doesn’t know the type of “eval”.
Solution: define them at the same time, with “and”

let rec eval: ... = function
    match expr with
    ...
    | AndE (exp1, exp2) -> handle_and(exp1, exp2)
    ...
    and

let handle_and: ... = function
    (exp1, exp2) -> ... (eval exp1) ... (eval exp2);;

• This same kind of code works in Racket without any special treatment. Be sure you understand why so that you can use this to test Rackette!
Another mutual recursion example: test natural numbers as even or odd

let rec is_even: int -> bool = function
    | 0 -> true
    | n -> is_odd (n-1) and
let rec is_odd: int -> bool = function
    | 0 -> false
    | n -> is_even (n-1) ;;

• Horrible inefficient and silly, but a nice complete example of mutual recursion.
  • Easy to translate to Racket.