CS51 SECOND MIDTERM EXAM CHEAT SHEET

CS51, SPRING 2021

1. PURE OCAML

```
# (* ..................................... BASICS *)
# () ;;
- : unit = ()
# 3 ;;
- : int = 3
# 3.0 ;;
- : float = 3.
# 30 + min (int_of_float 98.6) (145 / 12) ;;
- : int = 42
# min_int ;;
- : int = -4611686018427387904
# max_int ;;
- : int = 4611686018427387903
# 'A' ;;
- : char = 'A'
# "xyz" ;;
- : string = "xyz"
# false ;;
- : bool = false
# 3 < 5 && true ;;
- : bool = true
# Some 3 ;;
- : int option = Some 3
# None ;;
- : 'a option = None
# [3; 4] ;;
- : int list = [3; 4]
# [] ;;
- : 'a list = []
# (2, "xyz", 3.0) ;;
- : int * string * float = (2, "xyz", 3.)
# fst (2, "abc") ;;
- : int = 2
# snd (2, "abc") ;;
- : string = "abc"
# "x" ^ "y" ^ "z" ;;
- : string = "xyz"
# let x = 3 in
# if x < 0 || x > 0 then "nonzero" else "zero" ;;
- : string = "nonzero"
# let e = exp 1. in
# let pi = 2. *. asin 1. in
# Printf.printf "e  is %7.5f\ npi is %7.5f\ n" e pi ;;
e  is 2.71828
pi is 3.14159
- : unit = ()
# (* ..................... CHARACTERS AND STRINGS *)
# Char.code 'a' ;;
- : int = 97
# Char.code 'A' ;;
- : int = 65
# Char.code '0' ;;
- : int = 48
# Char.chr 97 ;;
- : char = 'a'
# String.length "hello" ;;
- : int = 5
# (* ................... FUNCTIONS AND APPLICATION *)
# fun x -> x + 1 ;;
- : int -> int = <fun>
# (fun x -> x + 1) 3 ;;
- : int = 4
# fun x y -> x + y ;;
- : int -> int -> int = <fun>
# fun (x, y) -> x + y ;;
- : int * int -> int = <fun>
# fun () -> 4 ;;
- : unit -> int = <fun>
# let uncurriedplus (x, y) = x + y in
# let curriedplus x y = x + y in
#   uncurriedplus (1, 2), curriedplus 1 2 ;;
- : int * int = (3, 3)
# 42 |> string_of_int |> print_endline ;;
42
- : unit = ()
# (* ................................... MATCHING *)
# let x = 3 in
#   match x with
#   | 0 -> "zero"
#   | 1 -> "one"
#   | _ -> "neither zero nor one" ;;
- : string = "neither zero nor one"
# let x, y = Some 111, 2999 in
#   match x, y with
#   | Some z, _ -> z + y
#   | None, _ -> y ;;
- : int = 3110
# let rec sum (x : int list) : int =
#   match x with
#   | [] -> 0
#   | u :: t -> u + sum t ;;
val sum : int list -> int = <fun>
# (* ..................................... RECORDS *)
# type rcrd = {foo : int; bar : string} ;;
type rcrd = { foo : int; bar : string; }
# {foo = 3; bar = "xyz"} ;;
- : rcrd = {foo = 3; bar = "xyz"}
# (* ...................................... LISTS *)
# 3 :: [4; 5] ;;
- : int list = [3; 4; 5]
# [1; 2; 3] @ [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
# List.length [8; 9; 10] ;;
- : int = 3
```

```
# List.hd [3; 4] ;;
- : int = 3
# List.tl [3; 4] ;;
- : int list = [4]
# List.tl [4] ;;
- : int list = []
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.map (fun x -> x + 100) [2; 3; 4] ;;
- : int list = [102; 103; 104]
# List.map ((+) 100) [2; 3; 4] ;;
- : int list = [102; 103; 104]
# List.map (fun x -> x = 3) [2; 3; 4] ;;
- : bool list = [false; true; false]
# List.filter ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.filter (fun x -> x < 4) [4; 3; 9; 6; 1; 0; 5] ;;
- : int list = [3; 1; 0]
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# List.fold_right (^) ["a"; "b"; "c"] "x" ;;
- : string = "abcx"
# List.fold_left ;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_left (^) "x" ["a"; "b"; "c"] ;;
- : string = "xabc"
# List.init ;;
- : int -> (int -> 'a) -> 'a list = <fun>
# List.init 5 (( * ) 3) ;;
- : int list = [0; 3; 6; 9; 12]
# List.find ;;
- : ('a -> bool) -> 'a list -> 'a = <fun>
# List.find (fun x -> x > 10) [1; 5; 10; 13; 19] ;;
- : int = 13
# List.find (fun x -> x > 10) [1; 5; 10] ;;
Exception: Not_found.
# List.rev [8; 9; 10] ;;
- : int list = [10; 9; 8]
# List.nth [8; 9; 10] 2 ;;
- : int = 10
# List.nth [8; 9; 10] 3 ;;
Exception: Failure "nth".
# (* ......................... VARIANT DATA TYPES *)
# type 'a option =
#   | Some of 'a
#   | None  ;;
type 'a option = Some of 'a | None
# type 'a stack =
#   | Empty
#   | Top of ('a * 'a stack) ;;
type 'a stack = Empty | Top of ('a * 'a stack)
# Top (3, Empty) ;;
- : int stack = Top (3, Empty)
# (* ................................ EXCEPTIONS *)
# Not_found ;;
- : exn = Not_found
# raise Not_found ;;
Exception: Not_found.
# raise (Failure "error") ;;
Exception: Failure "error".
# failwith "error" ;;
Exception: Failure "error".
# try
#   Some (List.find (fun x -> x > 10) [1; 5; 10])
# with
#   | Not_found -> None
#   | e -> raise e ;;
- : int option = None
# (* ...................... MODULES AND SIGNATURES *)
# module type STACK =
#   sig
#     type elt
#     type stack
#     exception Empty of string
#     val empty : unit -> stack
#     val push : stack * elt -> stack
#     val pop : stack -> elt * stack
#     val isEmpty : stack -> bool
#   end ;;
module type STACK =
  sig
    type elt
    type stack
    exception Empty of string
    val empty : unit -> stack
    val push : stack * elt -> stack
    val pop : stack -> elt * stack
    val isEmpty : stack -> bool
  end
# module MakeStack (Arg: sig type t end)
#                : (STACK with type elt = Arg.t) =
#   struct
#     type elt = Arg.t
#     type stack = elt list
#     exception Empty of string
#     let empty () = []
#     let push (s,x) = x::s
#     let pop s = match s with
#     | x :: t -> (x, t)
#     | [] -> raise (Empty "empty")
#     let isEmpty = fun x -> x = []
#   end ;;
module MakeStack :
  functor (Arg : sig type t end) ->
    sig
      type elt = Arg.t
      type stack
      exception Empty of string
      val empty : unit -> stack
      val push : stack * elt -> stack
      val pop : stack -> elt * stack
      val isEmpty : stack -> bool
    end
# module IntStack : (STACK with type elt = int) =
#   MakeStack (struct type t = int end) ;;
module IntStack :
  sig
    type elt = int
    type stack
    exception Empty of string
    val empty : unit -> stack
    val push : stack * elt -> stack
    val pop : stack -> elt * stack
    val isEmpty : stack -> bool
  end
# IntStack.push (IntStack.empty () , 3) ;;
- : IntStack.stack = <abstr>
```

2. IMPURE OCAML

```
# (* ........................ REFS AND MUTABLES *)
# let rint = ref 3 ;;
val rint : int ref = {contents = 3}
# rint := !rint + 5 ;;
- : unit = ()
# rint ;;
- : int ref = {contents = 8}
# !rint ;;
- : int = 8
# type mut_demo = { mutable mut: int ; nonmut: int } ;;
type mut_demo = { mutable mut : int; nonmut : int; }
# let m = { mut = 3; nonmut = 4 } ;;
val m : mut_demo = {mut = 3; nonmut = 4}
# m.mut <- 5 ;;
- : unit = ()
# m ;;
- : mut_demo = {mut = 5; nonmut = 4}
# (* ................................. LAZINESS *)
# let a = lazy (!rint * 2) ;;
val a : int lazy_t = <lazy>
# rint := 42 ;;
- : unit = ()
# Lazy.force a ;;
- : int = 84
# (* ........................ OBJECTS AND CLASSES *)
# class type demo_parent =
#   object
#     val v1 : int
#     method set : int -> unit
#     method is_odd : bool
#     method save : unit
#     method restore : unit
#   end ;;
class type demo_parent =
  object
    val v1 : int
    method is_odd : bool
    method restore : unit
    method save : unit
    method set : int -> unit
  end
# class type demo_child =
#   object
#     inherit demo_parent
#     method getsaved : int
#   end ;;
class type demo_child =
  object
    val v1 : int
    method getsaved : int
    method is_odd : bool
    method restore : unit
    method save : unit
    method set : int -> unit
  end
# class demo (initial_v1 : int) : demo_child =
# object (this)
#     val mutable v1 = initial_v1
#     val mutable v1saved = 0
#     method save = v1saved <- v1
#     method restore = this#set v1saved
#     method set n = v1 <- n
#     method is_odd = v1 mod 2 = 0
#     method getsaved = v1saved
#   end ;;
class demo : int -> demo_child
# let ob = new demo 3 ;;
val ob : demo = <obj>
# ob#save ;;
- : unit = ()
# ob#set 10 ;;
- : unit = ()
# ob#getsaved ;;
- : int = 3
# ob#is_odd ;;
- : bool = true
# ob#restore ;;
- : unit = ()
# ob#is_odd ;;
- : bool = false
# (* ..................................... PRINTING *)
# print_int 42; print_newline () ;;
42
- : unit = ()
# print_float 3.14159; print_newline () ;;
3.14159
- : unit = ()
# print_string "a string\ n" ;;
a string
- : unit = ()
# Printf.printf "int: %d; float: %f; bool: %B; string: %s\ n"
#   42 3.14159 true "a string" ;;
int: 42; float: 3.141590; bool: true; string: a string
- : unit = ()
```

3. STDLIB MODULE

Signatures for functions in the `Stdlib` module.

```
(* Exceptions *)

val raise : exn -> 'a
val invalid_arg : string -> 'a
val failwith : string -> 'a
exception Exit

(* Comparisons *)

val (=) : 'a -> 'a -> bool
val (<>) : 'a -> 'a -> bool
val (<) : 'a -> 'a -> bool
val (>) : 'a -> 'a -> bool
val (<=) : 'a -> 'a -> bool
val (>=) : 'a -> 'a -> bool
val compare : 'a -> 'a -> int
val min : 'a -> 'a -> 'a
val max : 'a -> 'a -> 'a
val (==) : 'a -> 'a -> bool
val (!=) : 'a -> 'a -> bool
```

```
(* Boolean operations *)

val not : bool -> bool
val (&&) : bool -> bool -> bool
val (||) : bool -> bool -> bool

(* Composition operators *)

val (|>) : 'a -> ('a -> 'b) -> 'b
val (@@) : ('a -> 'b) -> 'a -> 'b

(* Integer arithmetic *)

val (~-) : int -> int  (* negation *)
val (~+) : int -> int  (* unary addition *)
val succ : int -> int  (* succcessor *)
val pred : int -> int  (* predecessor *)
val (+) : int -> int -> int
val (-) : int -> int -> int
val ( * ) : int -> int -> int
val (/) : int -> int -> int (* truncated division *)
val (mod) : int -> int -> int (* remainder *)
val abs : int -> int (* absolute value *)
val max_int : int (* largest representable int *)
val min_int : int (* smallest representable int *)

(* Bitwise operations *)

val (land) : int -> int -> int
val (lor) : int -> int -> int
val (lxor) : int -> int -> int
val lnot : int -> int
val (lsl) : int -> int -> int
val (lsr) : int -> int -> int
val (asr) : int -> int -> int

(* Floating-point arithmetic *)

val (~-.) : float -> float (* unary negation *)
val (~+.) : float -> float (* unary addition *)
val (+.) : float -> float -> float
val (-.) : float -> float -> float
val ( *. ) : float -> float -> float
val (/.) : float -> float -> float
val ( ** ) : float -> float -> float (* exp'n *)
val sqrt : float -> float (* square root *)
val exp : float -> float (* exponential *)
val log : float -> float (* natural log *)
val log10 : float -> float (* base 10 log *)
val expm1 : float -> float
val cos : float -> float (* cosine, arg. in radians *)
val sin : float -> float
val tan : float -> float
val acos : float -> float
val asin : float -> float
val atan : float -> float
val atan2 : float -> float -> float
val hypot : float -> float -> float
      (* sqrt(x *. x + y *. y) *)
val cosh : float -> float
val sinh : float -> float
val tanh : float -> float
val ceil : float -> float
val floor : float -> float
val abs_float : float -> float
val mod_float : float -> float -> float
val float_of_int : int -> float
val int_of_float : float -> int (* truncate *)
val infinity : float
val neg_infinity : float
val nan : float
val max_float : float
val min_float : float

(* String operations *)

val (^) : string -> string -> string (* concatenation *)

(* Character operations *)

val int_of_char : char -> int
val char_of_int : int -> char

(* Unit operations *)

val ignore : 'a -> unit

(* String conversion functions *)

val string_of_bool : bool -> string
val bool_of_string : string -> bool
val string_of_int : int -> string
val int_of_string : string -> int
val string_of_float : float -> string
val float_of_string : string -> float

(* Pair operations *)

val fst : 'a * 'b -> 'a
val snd : 'a * 'b -> 'b

(* List operations *)

val (@) : 'a list -> 'a list -> 'a list (* append *)

(* Output functions on standard output *)

val print_char : char -> unit
val print_string : string -> unit
val print_bytes : bytes -> unit
val print_int : int -> unit
val print_float : float -> unit
val print_endline : string -> unit
val print_newline : unit -> unit

(* Input functions on standard input *)

val read_line : unit -> string
val read_int : unit -> int
val read_float : unit -> float

(* Program termination *)

val exit : int -> 'a
```

4. List module

Signatures for functions in the `List` module.

```
(* Basics *)
val length : 'a list -> int
val compare_lengths : 'a list -> 'b list -> int
val compare_length_with : 'a list -> int -> int
val cons : 'a -> 'a list -> 'a list
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val nth : 'a list -> int -> 'a
val nth_opt : 'a list -> int -> 'a option
val rev : 'a list -> 'a list
val init : int -> (int -> 'a) -> 'a list
val append : 'a list -> 'a list -> 'a list
val rev_append : 'a list -> 'a list -> 'a list
val concat : 'a list list -> 'a list
val flatten : 'a list list -> 'a list

(* Iterators *)
val iter : ('a -> unit) -> 'a list -> unit
val iteri : (int -> 'a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val filter_map : ('a -> 'b option)
   -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a)
  -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b)
   -> 'a list -> 'b -> 'b

(* Iterators on two lists *)
val iter2 : ('a -> 'b -> unit)
     -> 'a list -> 'b list -> unit
val map2 : ('a -> 'b -> 'c)
    -> 'a list -> 'b list -> 'c list
val rev_map2 : ('a -> 'b -> 'c)
        -> 'a list -> 'b list -> 'c list
val fold_left2 : ('a -> 'b -> 'c -> 'a) ->
   'a -> 'b list -> 'c list -> 'a
val fold_right2 : ('a -> 'b -> 'c -> 'c) ->
    'a list -> 'b list -> 'c -> 'c

(* List scanning *)
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val for_all2 : ('a -> 'b -> bool)
        -> 'a list -> 'b list -> bool
val exists2 : ('a -> 'b -> bool)
       -> 'a list -> 'b list -> bool
val mem : 'a -> 'a list -> bool
val memq : 'a -> 'a list -> bool

(* List searching *)
val find : ('a -> bool) -> 'a list -> 'a
val find_opt : ('a -> bool) -> 'a list -> 'a option
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool)
  -> 'a list -> 'a list * 'a list

(* Association lists *)
val assoc : 'a -> ('a * 'b) list -> 'b
val assoc_opt : 'a -> ('a * 'b) list -> 'b option
val assq : 'a -> ('a * 'b) list -> 'b
val assq_opt : 'a -> ('a * 'b) list -> 'b option
val mem_assoc : 'a -> ('a * 'b) list -> bool
val mem_assq : 'a -> ('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list
                     -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list
                    -> ('a * 'b) list

(* Lists of pairs *)
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list

(* Sorting *)
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
val stable_sort : ('a -> 'a -> int) -> 'a list
                                    -> 'a list
val fast_sort : ('a -> 'a -> int) -> 'a list
                                  -> 'a list
val sort_uniq : ('a -> 'a -> int) -> 'a list
                                  -> 'a list
val merge : ('a -> 'a -> int) -> 'a list
                              -> 'a list -> 'a list

(* Iterators *)
val to_seq : 'a list -> 'a Seq.t
val of_seq : 'a Seq.t -> 'a list
```

5. Semantics rules

5.1. **Substitution semantics.**

Definition of FV, the set of free variables in expressions for a functional language with naming and arithmetic.

$$FV(\overline{m}) = \varnothing \qquad \text{(integers)}$$

$$FV(x) = \{x\} \qquad \text{(variables)}$$

$$FV(P \texttt{ + } Q) = FV(P) \cup FV(Q) \qquad \text{(and similarly for other binary operators)}$$

$$FV(P\ Q) = FV(P) \cup FV(Q) \qquad \text{(applications)}$$

$$FV(\texttt{fun } x \texttt{ -> } P) = FV(P) - \{x\} \qquad \text{(functions)}$$

$$FV(\texttt{let } x \texttt{ = } P \texttt{ in } Q) = (FV(Q) - \{x\}) \cup FV(P) \qquad \text{(binding)}$$

Definition of substitution of expressions for variables in expressions for a functional language with naming and arithmetic.

$$\begin{aligned}
\overline{m}[x \mapsto P] &= \overline{m} \\
x[x \mapsto P] &= P \\
y[x \mapsto P] &= y \qquad \text{where } x \neq y \\
(Q \texttt{ + } R)[x \mapsto P] &= Q[x \mapsto P] \texttt{ + } R[x \mapsto P] \\
&\quad \text{and similarly for other binary operators} \\
Q\,R[x \mapsto P] &= Q[x \mapsto P]\,R[x \mapsto P] \\
(\texttt{fun } x \texttt{ -> } Q)[x \mapsto P] &= \texttt{fun } x \texttt{ -> } Q \\
(\texttt{fun } y \texttt{ -> } Q)[x \mapsto P] &= \texttt{fun } y \texttt{ -> } Q[x \mapsto P] \\
&\quad \text{where } x \neq y \text{ and } y \notin FV(P) \\
(\texttt{fun } y \texttt{ -> } Q)[x \mapsto P] &= \texttt{fun } z \texttt{ -> } Q[y \mapsto z][x \mapsto P] \\
&\quad \text{where } x \neq y \text{ and } y \in FV(P) \text{ and } z \text{ is a fresh variable} \\
(\texttt{let } x \texttt{ = } Q \texttt{ in } R)[x \mapsto P] &= \texttt{let } x \texttt{ = } Q[x \mapsto P] \texttt{ in } R \\
(\texttt{let } y \texttt{ = } Q \texttt{ in } R)[x \mapsto P] &= \texttt{let } y \texttt{ = } Q[x \mapsto P] \texttt{ in } R[x \mapsto P] \\
&\quad \text{where } x \neq y \text{ and } y \notin FV(P) \\
(\texttt{let } y \texttt{ = } Q \texttt{ in } R)[x \mapsto P] &= \texttt{let } z \texttt{ = } Q[x \mapsto P] \texttt{ in } R[y \mapsto z][x \mapsto P] \\
&\quad \text{where } x \neq y \text{ and } y \in FV(P) \text{ and } z \text{ is a fresh variable}
\end{aligned}$$

Substitution semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

(R_{int})
$$\overline{n} \Downarrow \overline{n}$$

(R_{fun})
$$\texttt{fun } x \texttt{ -> } B \Downarrow \texttt{fun } x \texttt{ -> } B$$

(R_{+})
$$\begin{array}{l} P \texttt{ + } Q \Downarrow \\ \quad \left|\begin{array}{l} P \Downarrow \overline{m} \\ Q \Downarrow \overline{n} \end{array}\right. \\ \quad \Downarrow \overline{m+n} \end{array}$$

($R_{/}$)
$$\begin{array}{l} P \texttt{ / } Q \Downarrow \\ \quad \left|\begin{array}{l} P \Downarrow \overline{m} \\ Q \Downarrow \overline{n} \end{array}\right. \\ \quad \Downarrow \overline{\lfloor m/n \rfloor} \end{array}$$

(R_{let})
$$\begin{array}{l} \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow \\ \quad \left|\begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array}\right. \\ \quad \Downarrow v_B \end{array}$$

(R_{app})
$$\begin{array}{l} P\ Q \Downarrow \\ \quad \left|\begin{array}{l} P \Downarrow \texttt{fun } x \texttt{ -> } B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array}\right. \\ \quad \Downarrow v_B \end{array}$$

5.2. **Environment semantics.**

Dynamic environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

(R_{int})
$$E \vdash \overline{n} \Downarrow \overline{n}$$

(R_{var})
$$E \vdash x \Downarrow E(x)$$

(R_{fun})
$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow \texttt{fun}\ x\ \texttt{->}\ P$$

(R_{+})
$$\frac{\begin{array}{l}E \vdash P \Downarrow \overline{m} \\ E \vdash Q \Downarrow \overline{n}\end{array}}{E \vdash P\ \texttt{+}\ Q \Downarrow \overline{m+n}}$$

(and similarly for other binary operators)

(R_{let})
$$\frac{\begin{array}{l}E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B\end{array}}{E \vdash \texttt{let}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow v_B}$$

(R_{app})
$$\frac{\begin{array}{l}E \vdash P \Downarrow \texttt{fun}\ x\ \texttt{->}\ B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B\end{array}}{E \vdash P\ \ Q \Downarrow v_B}$$

Lexical environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

(R_{int})
$$E \vdash \overline{n} \Downarrow \overline{n}$$

(R_{var})
$$E \vdash x \Downarrow E(x)$$

(R_{fun})
$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow [E \vdash \texttt{fun}\ x\ \texttt{->}\ P]$$

(R_{+})
$$\frac{\begin{array}{l}E \vdash P \Downarrow \overline{m} \\ E \vdash Q \Downarrow \overline{n}\end{array}}{E \vdash P\ \texttt{+}\ Q \Downarrow \overline{m+n}}$$

(and similarly for other binary operators)

(R_{let})
$$\frac{\begin{array}{l}E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B\end{array}}{E \vdash \texttt{let}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow v_B}$$

(R_{app})
$$\frac{\begin{array}{l}E_d \vdash P \Downarrow [E_l \vdash \texttt{fun}\ x\ \texttt{->}\ B] \\ E_d \vdash Q \Downarrow v_Q \\ E_l\{x \mapsto v_Q\} \vdash B \Downarrow v_B\end{array}}{E_d \vdash P\ \ Q \Downarrow v_B}$$

Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

(R_{int}) $$E, S \vdash \overline{n} \Downarrow \overline{n}, S$$

(R_{var}) $$E, S \vdash x \Downarrow E(x), S$$

(R_{fun}) $$E, S \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow [E \vdash \texttt{fun}\ x\ \texttt{->}\ P], S$$

(R_{+})
$$\begin{array}{l} E, S \vdash P\ \texttt{+}\ Q \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash P \Downarrow \overline{m}, S' \\ E, S' \vdash Q \Downarrow \overline{n}, S'' \end{array} \right. \\ \quad \Downarrow \overline{m+n}, S'' \end{array}$$

(and similarly for other binary operators)

(R_{let})
$$\begin{array}{l} E, S \vdash \texttt{let}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash D \Downarrow v_D, S' \\ E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S'' \end{array} \right. \\ \quad \Downarrow v_B, S'' \end{array}$$

(R_{app})
$$\begin{array}{l} E_d, S \vdash P\ \ Q \Downarrow \\ \quad \left| \begin{array}{l} E_d, S \vdash P \Downarrow [E_l \vdash \texttt{fun}\ x\ \texttt{->}\ B], S' \\ E_d, S' \vdash Q \Downarrow v_Q, S'' \\ E_l\{x \mapsto v_Q\}, S'' \vdash B \Downarrow v_B, S''' \end{array} \right. \\ \quad \Downarrow v_B, S''' \end{array}$$

(R_{ref})
$$\begin{array}{l} E, S \vdash \texttt{ref}\ P \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash P \Downarrow v_P, S' \end{array} \right. \\ \quad \Downarrow l, S'\{l \mapsto v_P\} \qquad \text{(where } l \text{ is a new location)} \end{array}$$

(R_{deref})
$$\begin{array}{l} E, S \vdash \texttt{!}\ P \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \end{array} \right. \\ \quad \Downarrow S'(l), S' \end{array}$$

(R_{assign})
$$\begin{array}{l} E, S \vdash P\ \texttt{:=}\ Q \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \\ \quad \Downarrow (), S''\{l \mapsto v_Q\} \end{array}$$

(R_{seq})
$$\begin{array}{l} E, S \vdash P\ \texttt{;}\ Q \Downarrow \\ \quad \left| \begin{array}{l} E, S \vdash P \Downarrow (), S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \\ \quad \Downarrow v_Q, S'' \end{array}$$