

Specifying the Behavior of Expressions

None of the exercises were tested.

Exercise 3.1

$\lfloor (\text{value-of } \langle\langle x \rangle\rangle \rho) \rfloor = 10$

$\lfloor (\text{value-of } \langle\langle 3 \rangle\rangle \rho) \rfloor = 3$

$\lfloor (\text{value-of } \langle\langle v \rangle\rangle \rho) \rfloor = 5$

$\lfloor (\text{value-of } \langle\langle i \rangle\rangle \rho) \rfloor = 1$

Exercise 3.2

A $val \in ExpVal$ must be that which is in $Int + Bool$. Then a $val \in ExpVal$ for which $\lfloor [val] \rfloor \neq val$ is where $val \in Bool$, such as $val = true$.

Exercise 3.3

We are able to describe the arithmetic operations in terms of subtraction. We cannot do so if we chose addition.

Exercise 3.4

Let $\rho = [x=[33], y=[22]]$.

$$\frac{\frac{\text{(value-of-program } \langle\langle \text{if zero? } (-(x, 11)) \text{ then } -(y, 2) \text{ else } -(y, 4) \rangle\rangle)}{\text{(value-of } \langle\langle \text{if zero? } (-(x, 11)) \text{ then } -(y, 2) \text{ else } -(y, 4) \rangle\rangle \rho)} = (\text{bool-val } \#f)}{\text{(value-of } \langle\langle -(y, 4) \rangle\rangle \rho)} \\ \text{[18]}$$

Exercise 3.5

$$\frac{
\begin{array}{l}
(\text{value-of } \langle\langle \text{let } x = 7 \\
\quad \text{in let } y = 2 \\
\text{in let } y = \text{let } x = -(x, 1) \text{ in } -(x, y) \\
\quad \text{in } -(-(x, 8), y) \rangle\rangle \rho_0)
\end{array}
}{
\textit{skibidi - too - hard}
}$$

Exercise 3.6

For this exercise only, I expressed the operation in terms of others. Continuing this trend would have been a mistake.

```

(minus-exp (exp1 expression?))

(minus-exp (exp1)
  (value-of (diff-exp (const-exp 0)
    exp1)
    env))

```

Exercise 3.7

```

(add-exp
  (exp1 expression?)
  (exp2 expression?))
(mul-exp
  (exp1 expression?)
  (exp2 expression?))
(div-exp
  (exp1 expression?)
  (exp2 expression?))

(add-exp (exp1 exp2)
  (num-val (+ (expval->num (value-of exp1 env))
    (expval->num (value-of exp2 env))))))

```

```

(mul-exp (exp1 exp2)
  (num-val (* (expval->num (value-of exp1 env))
              (expval->num (value-of exp2 env)))))

(div-exp (exp1 exp2)
  (let ((val2 (expval->num (value-of exp2 env))))
    (if (= 0 val2)
        (report-division-by-zero)
        (num-val (/ (expval->num (value-of exp1 env))
                     val2)))))

```

Exercise 3.8

```

(equal?-exp
  (exp1 expression?)
  (exp2 expression?))

(greater?-exp
  (exp1 expression?)
  (exp2 expression?))

(less?-exp
  (exp1 expression?)
  (exp2 expression?))

(equal?-exp (exp1 exp2)
  (bool-val (= (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env)))))

(greater?-exp (exp1 exp2)
  (bool-val (> (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env)))))

(less?-exp (exp1 exp2)
  (bool-val (< (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env)))))

```

Exercise 3.9

We express a `list-val` as two expressed values. To select from the

cons we use `expval->list` to return a Scheme cons, and apply either `car` or `cdr` to the cons.

```
(emptylist-val)
(list-val
 (first expval?)
 (rest expval?))

(define expval->list
  (lambda (val)
    (cases expval val
      (list-val (first rest) (cons first rest))
      (else (report-expval-extractor-error 'list val))))))

(cons-exp
 (exp1 expression?)
 (exp2 expression?))
(car-exp
 (exp1 expression?))
(cdr-exp
 (exp1 expression?))
(null?-exp
 (exp1 expression?))
(emptylist-exp)

(cons-exp (exp1 exp2)
          (list-val (value-of exp1 env)
                    (value-of exp2 env)))
(car-exp (exp1)
          (car (expval->list (value-of exp1 env))))
(cdr-exp (exp1)
          (cdr (expval->list (value-of exp1 env))))
(null?-exp (exp1))
```

```

      (let ((val1 (value-of exp1 env)))
        (cases expval val1
          (emptylist-val () (bool-val #t))
          (else (bool-val #f)))))
(emptylist-exp ())
(emptylist-val))

```

Exercise 3.10

Expression ::= list({*Expression*}^(*))

We take advantage of using a list in the defining language to construct a list in the defined language. The Expressions are captured in a Scheme list. We map `value-of` to each *Expression* then transform the Scheme list into a nesting of `list-vals`, ending with `(emptylist-val)` when the Scheme list is null.

```

(list-exp
 (exps (list-of expression?)))

(list-exp (exps)
  (list->expval (map (lambda (expr)
                     (value-of expr env))
                    exps)))

(define list->expval
  (lambda (lst)
    (if (null? lst)
        (emptylist-val)
        (list-val (car lst)
                  (list->listval (cdr lst))))))

```

Exercise 3.11

We take the consequent expression of each variant in the cases and make a procedure out of it. Now `value-of` need only be modified by adding a

variant and procedure call, and the possibly large procedure can be made elsewhere.

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (const-op num))
      (var-exp (var) (var-op var env))
      (diff-exp (exp1 exp2) (diff-op exp1 exp2 env))
      (zero?-exp (exp1) (zero?-op exp1 env))
      (if-exp (exp1 exp2 exp3) (if-op exp1 exp2 exp3 env))
      (let-exp (var exp1 body) (let-op var exp1 body env))
      (minus-exp (exp1) (minus-op exp1 env))
      (add-exp (exp1 exp2) (add-op exp1 exp2 env))
      (mul-exp (exp1 exp2) (mul-op exp1 exp2 env))
      (div-exp (exp1 exp2) (div-op exp1 exp2 env))
      (equal?-exp (exp1 exp2) (equal?-op exp1 exp2 env))
      (greater?-exp (exp1 exp2) (greater?-op exp1 exp2 env))
      (less?-exp (exp1 exp2) (less?-op exp1 exp2 env))
      (cons-exp (exp1 exp2) (cons-op exp1 exp2 env))
      (car-exp (exp1) (car-op exp1))
      (cdr-exp (exp1) (cdr-op exp1))
      (null?-exp (exp1) (null?-op exp1 env))
      (emptylist-exp () (emptylist-op))
      (list-exp (exps) (list-op exps env))))))

(define const-op
  (lambda (num)
    (num-val num)))

(define var-op
  (lambda (var env)
    (apply-env env var)))
```

```

(define diff-op
  (lambda (exp1 exp2 env)
    (let ((val1 (value-of exp1 env))
          (val2 (value-of exp2 env)))
      (let ((num1 (expval->num val1))
            (num2 (expval->num val2)))
        (num-val (- num1 num2))))))

(define zero?-op
  (lambda (exp1 env)
    (let ((val1 (value-of exp1 env)))
      (let ((num1 (expval->num val1)))
        (if (zero? num1)
            (bool-val #t)
            (bool-val #f))))))

(define if-op
  (lambda (exp1 exp2 exp3 env)
    (let ((val1 (value-of exp1 env)))
      (if (expval->bool val1)
          (value-of exp2 env)
          (value-of exp3 env)))))

(define let-op
  (lambda (var exp1 body env)
    (let ((val1 (value-of exp1 env)))
      (value-of body
                  (extend-env var val1 env)))))

(define minus-op
  (lambda (exp1 env)

```

```

(value-of (diff-exp (const-exp 0) exp1)
  env)))

(define add-op
  (lambda (exp1 exp2 env)
    (num-val (+ (expval->num (value-of exp1 env))
      (expval->num (value-of exp2 env))))))

(define mul-op
  (lambda (exp1 exp2 env)
    (num-val (* (expval->num (value-of exp1 env))
      (expval->num (value-of exp2 env))))))

(define div-op
  (lambda (exp1 exp2 env)
    (let ((val2 (expval->num (value-of exp2 env))))
      (if (= 0 val2)
        (report-division-by-zero)
        (num-val (/ (expval->num (value-of exp1 env))
          val2))))))

(define equal?-op
  (lambda (exp1 exp2 env)
    (bool-val (= (expval->num (value-of exp1 env))
      (expval->num (value-of exp2 env))))))

(define greater?-op
  (lambda (exp1 exp2 env)
    (bool-val (> (expval->num (value-of exp1 env))
      (expval->num (value-of exp2 env))))))

(define less?-op

```



```

(lambda (exp1 exp2 env)
  (bool-val (< (expval->num (value-of exp1 env))
              (expval->num (value-of exp2 env))))))

(define cons-op
  (lambda (exp1 exp2 env)
    (list-val (value-of exp1 env)
              (value-of exp2 env))))

(define car-op
  (lambda (exp1)
    (car (expval->list exp1))))

(define cdr-op
  (lambda (exp1)
    (cdr (expval->list exp1))))

(define null?-op
  (lambda (exp1 env)
    (let ((vall (value-of exp1 env)))
      (cases expval vall
        (emptylist-val () (bool-val #t))
        (else (bool-val #f))))))

(define emptylist-op
  (lambda ()
    (emptylist-val)))

(define list-op
  (lambda (exps env)
    (list->expval (map (lambda (expr) (value-of expr env))
                      exps))))

```

Exercise 3.12

```
(cond-exp
  (clauses (list-of (list-of expression?))))

(cond-exp (clauses) (cond-op clauses env))

(define cond-op
  (lambda (clauses env)
    (if (null? clauses)
        (error-all-false-predicates)
        (let* ((clause (car clauses))
                (predicate (car clause))
                (val (value-of predicate env)))
          (if (expval->bool val)
              (let ((consequent (cadr clause)))
                (value-of consequent env))
              (cond-op (cdr clauses) env)))))))
```

Exercise 3.13

```
(define-datatype expval expval?
  (num-val
    (num number?)))

(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
          (num-val 1)
          (num-val 0))))))

(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
```

```

      (if (zero? (expval->num val1))
          (value-of exp3 env)
          (value-of exp2 env))))

```

Exercise 3.14

We introduce a new nonterminal *Bool-exp*. We represent it as a data type having its variants be predicate operations. We declare that **value-of-bool-exp** : *Bool-exp* × *Env* → *ExpVal*, and that these *ExpVals* are *Bools*. Thus all the variants are predicates. Lastly, we add a production to the grammar of *Expressions* to include *Bool-exps*. This allows a predicate operation to be written by itself without needing to be in an *if-exp*.

```

(define-datatype bool-exp bool-exp?
  (zero?-exp
    (exp expression?))
  (equal?-exp
    (exp1 expression?)
    (exp2 expression?))
  (greater?-exp
    (exp1 expression?)
    (exp2 expression?))
  (less?-exp
    (exp1 expression?)
    (exp2 expression?))

(define value-of-bool-exp
  (lambda (exp env)
    (cases bool-exp exp
      (zero?-exp (exp)
        (let ((val (value-of exp env)))
          (let ((num (expval->num val)))
            (if (zero? num)
                (bool-val #t)
                (bool-val #f))))))

```

```

(equal?-exp (exp1 exp2)
  (bool-val (= (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env)))))
(greater?-exp (exp1 exp2)
  (bool-val (> (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env))))
(less?-exp (exp1 exp2)
  (bool-val (< (expval->num (value-of exp1 env))
               (expval->num (value-of exp2 env)))))

(bool-expr
  (bool-exp1 bool-exp?)))

(if-exp (bool-exp1 exp2 exp3)
  (let ((val1 (value-of-bool-exp bool-exp1 env)))
    (if (expval->bool val1)
        (value-of exp2 env)
        (value-of exp3 env))))

(bool-expr (bool-exp1)
  (value-of-bool-exp bool-exp1 env))

```

Exercise 3.15

```

(print-exp
  (exp expression?))

(print-exp (exp)
  (display (value-of exp env))
  (num-val 1))

```

Exercise 3.16

```

(let-exp
  (vars (list-of identifier?))
  (exps (list-of expression?))

```

```

(body expression?))

(let-exp (vars exps body)
  (value-of body
    (extend-env* vars
      (map (lambda (exp)
              (value-of exp env))
            exps)
      env)))

```

Exercise 3.17

```

(let*-exp
  (vars (list-of identifier?))
  (exps (list-of expression?))
  (body expression?))

(let*-exp (vars exps body)
  (let*-op vars exps body env))

(define let*-op
  (lambda (vars exps body env)
    (if (null? vars)
        (value-of body env)
        (let*-op (cdr vars)
                  (cdr exps)
                  body
                  (extend-env (car vars)
                              (value-of (car exps) env)
                              env))))))

```

Exercise 3.18

```

(unpack-exp
  (vars (list-of identifier?))

```

```

(exp expression?)
(body expression?))

(unpack-exp (vars exp body)
            (unpack-op vars
                        (expand-expval->list (value-of exp env))
                        body
                        env))

(define unpack-op
  (lambda (vars lst body env)
    (if (null? vars)
        (if (null? lst)
            (value-of body env)
            (report-too-many-elements lst))
        (unpack-op (cdr vars)
                    (cdr lst)
                    body
                    (extend-env (car vars)
                                (car lst)
                                env)))))

(define expand-expval->list
  (lambda (val)
    (cases expval val
      (emptylist-val () '())
      (list-val (first rest)
                (cons first (expand-expval->list rest)))
      (else (report-not-a-list val)))))

```