5 Data

Exercise 5.1. Describe the type of each of these expressions.

a. 17

Solution. Number

b. (**lambda** (*a*) (> *a* 0))

Solution. A procedure of type: Number $\rightarrow$ Boolean

c. ((**lambda** (*a*) (> *a* 0)) 3)

Solution. Boolean

d. (**lambda** (*a*) (**lambda** (*b*) (> *a* *b*)))

Solution. A procedure of type: Number $\rightarrow$ (Number $\rightarrow$ Boolean). That is, the result of applying this procedure to a Number would be a procedure that takes a Number as input and outputs a Boolean.

e. (**lambda** (*a*) *a*)

Solution. $T \rightarrow T$. A procedure that takes any type as its input, and produces as its output a value of the same type as the input.

Exercise 5.2. Define or identify a procedure that has the given type.

a. Number $\times$ Number $\rightarrow$ Boolean

Solution. > (the built-in greater-than procedure takes two Numbers as inputs and outputs a Boolean)

b. Number $\rightarrow$ Number

Solution. $-$ (the built-in negation procedure takes one Number input and outputs a Number)

c. (Number $\rightarrow$ Number) $\times$ (Number $\rightarrow$ Number)
$\rightarrow$ (Number $\rightarrow$ Number)

Solution. The *fcompose* function from Section 4.2.1 takes as inputs two functions from Number to Number, and outputs a function that takes a Number as input and outputs a Number.

(**lambda** (*f* *g*) (**lambda** (*x*) (*g* (*f* *x*))))

d. Number $\rightarrow$ (Number $\rightarrow$ (Number $\rightarrow$ Number))

Solution.

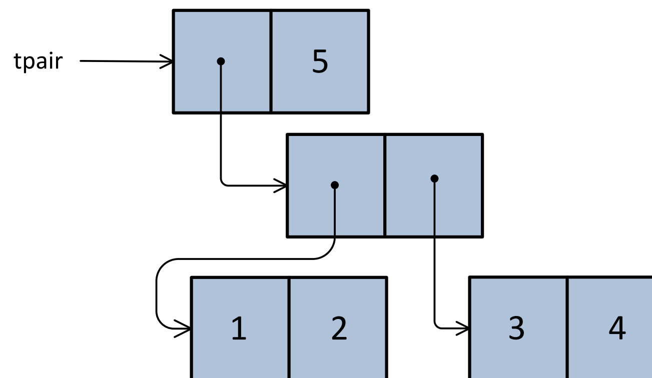
(**lambda** (*a*) (**lambda** (*b*) (**lambda** (*c*) (+ *a* *b* *c*))))

Exercise 5.3. Suppose the following definition has been executed:

```
(define tpair
  (cons (cons (cons 1 2) (cons 3 4))
        5))
```

Draw the structure defined by *tpair*, and give the value of each of the following expressions.

Solution. The expression for *tpair* creates a pair where the first part of the pair is a pair. That pair has the pair (1 . 2) as its first part, and the pair (3 . 4) as its second part. The second part of the main pair is the number 5.



a. (*cdr tpair*)

Solution. 5

b. (*car* (*car* (*car tpair*)))

Solution. 1

c. (*cdr* (*cdr* (*car tpair*)))

Solution. 4

d. (*cdr* (*cdr tpair*))

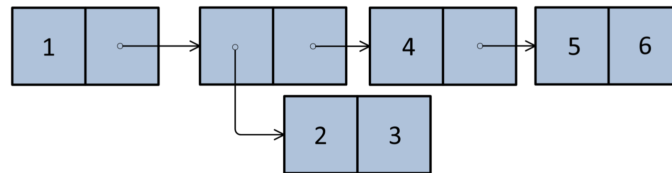
Solution. Error: the result of (*cdr tpair*) is the scalar value 5, so the attempted to apply *cdr* to this value results in an error.

Exercise 5.4. Write expressions that extract each of the four elements from *fstruct* defined by (**define** *fstruct* (*cons* 1 (*cons* 2 (*cons* 3 4)))).

Solution.

- (*car fstruct*) evaluates to 1.
- (*car* (*cdr fstruct*)) evaluates to 2.
- (*car* (*cdr* (*cdr fstruct*))) evaluates to 3.
- (*cdr* (*cdr* (*cdr fstruct*))) evaluates to 4.

Exercise 5.5. Give an expression that produces the structure shown below.



Solution. This expression produces the structure shown:

(*cons* 1 (*cons* (*cons* 2 3) (*cons* 4 (*cons* 5 6))))

Exercise 5.6. Convince yourself the definitions of *scons*, *scar*, and *scdr* above work as expected by following the evaluation rules to evaluate

(*scar* (*scons* 1 2))

Solution. The expression, (*scar* (*scons* 1 2)) is an *ApplicationExpression*, so the first step is to evaluate all the subexpressions. The first subexpression, *scar*, is a name expression that evaluates to the procedure, (**lambda** (*pair*) (*pair true*)). The second subexpression, (*scons* 1 2) is an *ApplicationExpression*. After evaluating all its subexpressions, the next step is to apply the value of the first subexpression, (**lambda** (*a b*) (**lambda** (*w*) (**if** *w a b*))), to the values of the other subexpressions which are the numbers 1 and 2. To evaluate the application of the compound procedure, we evaluate its body with the input values bound to the names of the parameters. So, *a* will have value 1, and *b* will have value 2, and the result of evaluating the body expression is (**lambda** (*w*) (**if** *w* 1 2)).

Next, we continue evaluating the outer application expression by applying the value of the first subexpression, *scar*, to the value of the second subexpression. The value of *scar* is (**lambda** (*pair*) (*pair true*)) and the value of the second subexpression is (**lambda** (*w*) (**if** *w* 1 2)). So, we follow the application rule for compound procedures. The name *pair* is associated with the value, (**lambda** (*w*) (**if** *w* 1 2)). Then we evaluate the body expression, (*pair true*). Substituting the value of *pair*, this becomes ((**lambda** (*w*) (**if** *w* 1 2)) *true*).

Following the evaluation rules, we associate *w* with the value *true* and evaluate the body, which is the if expression, (**if** *w* 1 2). This evaluates to 1, which is the first part of the pair created using *scons*, so is consistent with the desired behavior of *scons* and *scar*.

Exercise 5.7. Show the corresponding definitions of *tcar* and *tcdr* that provide the pair selection behavior for a pair created using *tcons* defined as:

(**define** (*tcons a b*) (**lambda** (*w*) (**if** *w b a*)))

Solution. Since *tcons* reverses the order of *a* and *b*, we need to define *tcar* and *tcdr* as:

(**define** (*tcar pair*) (*pair false*))
(**define** (*tcdr pair*) (*pair true*))

Exercise 5.8. Define a procedure that constructs a quintuple and procedures for selecting the five elements of a quintuple.

Solution. The most natural way is to use the procedures we defined for making quadruples:

(**define** (*make-quintuple a b c d e*) (*cons a* (*make-quadruple b c d e*)))
(**define** (*quintuple-first q*) (*car q*))
(**define** (*quintuple-second q*) (*quad-first* (*cdr q*)))
(**define** (*quintuple-third q*) (*quad-second* (*cdr q*)))
(**define** (*quintuple-fourth q*) (*quad-third* (*cdr q*)))
(**define** (*quintuple-fifth q*) (*quad-fourth* (*cdr q*)))

Exercise 5.9. Another way of thinking of a triple is as a Pair where the first cell is a Pair and the second cell is a scalar. Provide definitions of *make-triple*, *triple-first*, *triple-second*, and *triple-third* for this construct.

Solution.

(**define** (*make-triple a b c*) (*cons* (*cons a b*) *c*))
(**define** (*triple-first t*) (*car* (*car t*)))
(**define** (*triple-second t*) (*cdr* (*car t*)))
(**define** (*triple-third t*) (*cdr t*))

Exercise 5.10. For each of the following expressions, explain whether or not the expression evaluates to a List. Check your answers with a Scheme interpreter by using the *list?* procedure.

a. *null*

Solution. Yes, *null* is a List by the direct definition of a List.

b. (*cons* 1 2)

Solution. No, (*cons* 1 2) is **not** a List, since it is a Pair but the second part of the Pair is 2 which is not a List.

c. (*cons null null*)

Solution. Yes, (*cons null null*) is a List, since it is a Pair where the second part of the Pair is *null* which is a List.

d. (*cons* (*cons* (*cons* 1 2) 3) *null*)

Solution. Yes, (*cons* (*cons* (*cons* 1 2) 3) *null*) is a List, since it is a Pair where the second part of the Pair is *null* which is a List.

e. (*cdr* (*cons* 1 (*cons* 2 (*cons null null*))))

Solution. Yes, (*cdr* (*cons* 1 (*cons* 2 (*cons null null*)))) is a List. The *cdr* application evlauates to the second part of the input Pair, (*cons* 2 (*cons null null*)). This is a List, since it is a Pair, where the second part of the Pair, (*cons null null*) is a List.

f. (*cons* (*list* 1 2 3) 4)

Solution. No, (*cons* (*list* 1 2 3) 4) is **not** a List since the second part of the Pair is the scalar value 4, which is not a List.

Exercise 5.11. Define a procedure *is-list?* that takes one input and outputs true if the input is a List, and false otherwise. Your procedure should behave identically to the built-in *list?* procedure, but you should not use *list?* in your definition.

Solution. Here's a straightforward we do define *is-list?*:

```
(define (is-list? p)
   (if (null? p)
      true
      (if (pair? p)
         (is-list? (cdr p))
         false)))
```

We can make a simpler definition, that follows closely the way we defined a List, by using the *and* and *or* special forms:

```
(define (is-list? p)
   (or (null? p) (and (pair? p) (is-list? (cdr p)))))
```

Note that this relies on an important property of the *or* and *and* special forms, which is that unlike an *ApplicationExpression*, they do not evaluate all the subexpressions. Instead, the *or* special form expression (*or p q*) behaves like, (**if** *p true q*), so *q* is not evaluated if the first expression evaluates to *true*. The *and* special form expression (*and p q*) behaves like, (**if** *p q false*). In the *is-list?* definition, this is why (*and* (*pair? p*) (*is-list?* (*cdr p*))) evaluates correctly. If (*pair? p*) evaluates to *false*, the second subexpression is never evaluated. If it were evaluated when *p* is not a Pair, this would produce an error when the expression (*cdr p*) is evaluated on an input that is not a Pair.

Exercise 5.12. Define a procedure *list-max* that takes a List of non-negative numbers as its input and produces as its result the value of the greatest element in the List (or 0 if there are no elements in the input List). For example, (*list-max* (*list* 1 1 2 0)) should evaluate to 2.

Solution. Here is a straightforward definition:

```
(define (list-max p)
  (if (null? p)
     0
     (if (> (list-max (cdr p)) (car p))
        (list-max (cdr p))
        (car p))))
```

This is correct, but isn't a great way to define *list-max* since it requires lost of duplicate work. The subexpression, (*list-max* (*cdr p*)) may involve a lot of work for a long list, but if the maximum element of the list is at the end of the list, this expression is evaluated *twice* for each element.

A better definition would avoid this duplicate work, by using the *bigger* procedure (from Chapter 3):

```
(define (bigger a b) (if (> a b) a b))

(define (list-max p)
  (if (null? p)
     0
     (bigger (car p) (list-max (cdr p)))))
```

This definition is better than the previous one in at least three ways: it is shorter, it is easier to

understand and be convinced that it is correct, and it is more efficient.

Exercise 5.13. Use *list-accumulate* to define *list-max* (from Exercise **??**).

Solution.

```
(define (list-max p)
  (list-accumulate bigger 0 p))
```

If *bigger* is not already defined,

```
(define (list-max p)
  (list-accumulate (lambda (a b) (if (> a b) a b)) 0 p))
```

Exercise 5.14. [⋆⋆] Use *list-accumulate* to define *is-list?* (from Exercise **??**).

Solution. This isn't really possible! The problem is *list-accumulate* assumes its input is a List. Its alternate clause, (*f* (*car p*) (*list-accumulate f base* (*cdr p*))) includes (*car p*) which will produce an error if *p* is not a Pair. But, if the input to *is-list?* is not a List, then at some point the error must be produced. So, we could use *list-accumulate* to define a procedure that evaluates to *true* when its input is a List, but produces an error otherwise:

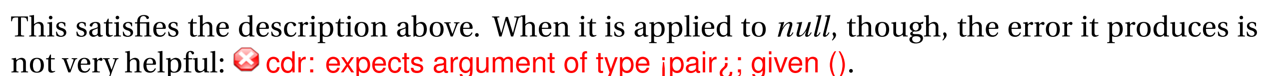
```
(define (is-list-or-error? p)
  (list-accumulate (lambda (a b) true) true p))
```

A procedure that evaluates to *ture* on valid Lists and produces an error for non-list inputs, though, isn't very useful.

Exercise 5.15. Define a procedure *list-last-element* that takes as input a List and outputs the last element of the input List. If the input List is empty, *list-last-element* should produce an error.

Solution. A natural way to define *list-last-element* is:

```
(define (last-list-element p)
  (if (null? (cdr p))
     (car p)
     (last-list-element (cdr p))))
```


This satisfies the description above. When it is applied to *null*, though, the error it produces is not very helpful: cdr: expects argument of type ¡pair¿; given ().

A better definition of *list-last-element* would produce a more helpful error message:

```
(define (last-list-element p)
  (if (null? p)
     (error "Cannot apply last-list-element to empty list.")
     (if (null? (cdr p))
        (car p)
        (last-list-element (cdr p)))))
```

Exercise 5.16. Define a procedure *list-ordered?* that takes two inputs, a test procedure and a List. It outputs true if all the elements of the List are ordered according to the test procedure. For example, (*list-ordered?* < (*list* 1 2 3)) evaluates to true, and (*list-ordered?* < (*list* 1 2 3 2)) evaluates

to false. Hint: think about what the output should be for the empty list.

Solution.

```
(define (list-ordered? cf p)
  (if (null? p)
     true
     (if (null? (cdr p))
        true
        (if (cf (car p) (car (cdr p)))
           (list-ordered? cf (cdr p))
           false))))
```

We can create a more concise definition using the *or* and *and* special forms:

```
(define (list-ordered? cf p)
  (or (null? p)
      (null? (cdr p))
      (and (cf (car p) (car (cdr p))) (list-ordered? cf (cdr p)))))
```

Exercise 5.17. Define a procedure *list-increment* that takes as input a List of numbers, and produces as output a List containing each element in the input List incremented by one. For example, (*list-increment* 1 2 3) evaluates to (2 3 4).

Solution. Without using *list-map*:

```
(define (list-increment p)
  (if (null? p)
     null
     (cons (+ 1 (car p)) (list-increment (cdr p)))))
```

Using *list-map*:

```
(define (list-increment p)
  (list-map (lambda (n) (+ n 1)) p))
```

Exercise 5.18. Use *list-map* and *list-sum* to define *list-length*:

```
(define (list-length p) (list-sum (list-map _____ p)))
```

Solution. We need to replace each element in the list with the value 1:

```
(define (list-length p) (list-sum (list-map (lambda (v) 1) p)))
```

Exercise 5.19. Define a procedure *list-filter-even* that takes as input a List of numbers and produces as output a List consisting of all the even elements of the input List.

Solution. Here's a definition without using *list-filter*:

```
(define (list-filter-even p)
  (if (null? p)
      null
      (if (even? (car p))
          (cons (car p) (list-filter-even (cdr p)))
          (list-filter-even (cdr p)))))
```

Using *list-filter*, a shorter definition is possible:

```
(define (list-filter-even p)
  (list-filter even? p))
```

Exercise 5.20. Define a procedure *list-remove* that takes two inputs: a test procedure and a List. As output, it produces a List that is a copy of the input List with all of the elements for which the test procedure evaluates to true removed. For example, (*list-remove* (**lambda** (*x*) (= *x* 0)) (*list* 0 1 2 3)) should evaluates to the List (1 2 3).

Solution. One way to define *list-remove* is similar to how we defined *list-filter* but switching the clauses:

```
(define (list-remove test p)
  (list-accumulate
    (lambda (el rest) (if (test el) rest (cons el rest)))
    null
    p))
```

A more clever approach would use *list-filter*:

```
(define (list-remove test p)
  (list-filter (lambda (n) (not (test n))) p))
```

Or, using *fcompose* from Chapter 2:

```
(define (list-remove test p)
  (list-filter (fcompose test not) p))
```

Exercise 5.21. [★★] Define a procedure *list-unique-elements* that takes as input a List and produces as output a List containing the unique elements of the input List. The output List should contain the elements in the same order as the input List, but should only contain the first appearance of each value in the input List.

Solution. First, we define a procedure *list-contains?* that takes as inputs a value and a list and evaluates to true if and only if the list contains at least one element equal to the input value:

```
(define (list-contains? el p)
  (if (null? p)
      false
      (if (equal? el (car p))
          true
          (list-contains? el (cdr p)))))
```

Using *list-contains?*, we define *list-unique-elements*:

```
(define (list-unique-elements p)
  (if (null? p)
      null
      (if (list-contains? (car p) (cdr p))
          (list-unique-elements (list-filter (lambda (el) (not (equal? el (car p)))) p))
          (cons (car p) (list-unique-elements (cdr p))))))
```

The trickiest part is using *list-filter* in the consequence clause of the second if expression. This is necessary since we need to remove all instances of (*car p*) from the list for the recursive call. Otherwise, a non-unique element would still be included once the last instance of that element is reached!

Exercise 5.22. Define the *list-reverse* procedure using *list-accumulate*.

Solution.

```
(define (list-reverse p)
  (list-accumulate (lambda (el lst) (list-append lst (list el))) null p))
```

Exercise 5.23. Define *factorial* using *intsto*.

Solution.

```
(define (factorial n)
  (apply * (intsto n)))
```

Exercise 5.24. [⋆] Define a procedure *deep-list-map* that behaves similarly to *list-map* but on deeply nested lists. It should take two parameters, a mapping procedure, and a List (that may contain deeply nested Lists as elements), and outputs a List with the same structure as the input List with each value mapped using the mapping procedure.

Solution.

```
(define (deep-list-map f p)
  (if (null? p)
      null
      (cons
       (if (list? (car p))
           (deep-list-map f (car p))
           (f (car p)))
       (deep-list-map f (cdr p)))))
```

Exercise 5.25. [⋆] Define a procedure *deep-list-filter* that behaves similarly to *list-filter* but on deeply nested lists.

Solution.

```
(define (deep-list-filter f p)
  (if (null? p)
      null
      (if (list? (car p))
          (cons (deep-list-filter f (car p))
                (deep-list-filter f (cdr p)))
          (if (f (car p))
              (cons (car p) (deep-list-filter f (cdr p)))
              (deep-list-filter f (cdr p))))))
```