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 → Boolean

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

Solution. Boolean

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

Solution. A procedure of type: Number → (Number → 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. $T \to T$. 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 × Number → Boolean

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

b. Number → Number

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

c. (Number → Number) × (Number → Number)
→ (Number → 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 → (Number → (Number → 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.

a. (*cdr tpair*)

Solution.

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

Solution.

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

Solution.

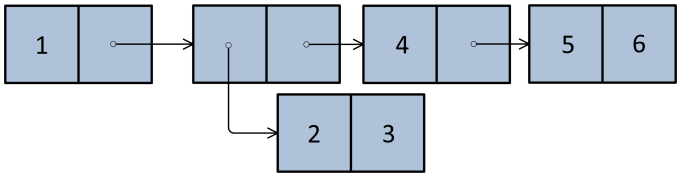
d. (*car* (*cdr* (*cdr tpair*)))

Solution.

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.

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



Solution.

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.

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.

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

Solution.

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.

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*

b. (*cons* 1 2)

c. (*cons null null*)

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

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

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

Solution.

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.

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.

Exercise 5.13. Use *list-accumulate* to define *list-max* (from Exercise 5.12).

Solution.

Exercise 5.14. [⋆] Use *list-accumulate* to define *is-list?* (from Exercise 5.11).

Solution.

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.

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.

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.

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

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

Solution.

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.

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.

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.

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

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

Solution.

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 output a List with the same structure as the input List with each value mapped using the mapping procedure.

Solution.

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

Solution.