4
Problems and Procedures

Exercise 4.1. For each expression, give the value to which the expression evaluates. Assume *fcompose* and *inc* are defined as above.

a. ((*fcompose square square*) 3)

Solution. The application expression (*fcompose square square*) evaluates to a procedure that composes square with square (that is, it multiples its input by itself four times). Hence, applying this procedure to 3 evaluates to 81.

b. (*fcompose* (**lambda** (x) ($*$ x 2)) (**lambda** (x) (/ x 2)))

Solution. This evaluates to an identity procedure for number inputs. It produces a procedure that takes a number as its input, and applies a procedure that multiplies by 2 to the result of a procedure that divides the input number by 2.

c. ((*fcompose* (**lambda** (x) ($*$ x 2)) (**lambda** (x) (/ x 2))) 1120)

Solution. This applies the identity procedure from the previous part to 1120, so the result is 1120.

d. ((*fcompose* (*fcompose inc inc*) *inc*) 2)

Solution. The inner application expression, (*fcompose inc inc*), evaluates to a procedure that takes a number as its input and outputs the result of incrementing it twice (that is, adding 2). The next application expression, (*fcompose* (*fcompose inc inc*) *inc*), composes this with another *inc* procedure, producing a procedure that adds 3 to its input. Applying this procedure to 2 results in the value 5.

Exercise 4.2. Suppose we define *self-compose* as a procedure that composes a procedure with itself:

(**define** (*self-compose f*) (*fcompose f f*))

Explain how (((*fcompose self-compose self-compose*) *inc*) 1) is evaluated.

Solution. The application expression, (*fcompose self-compose self-compose*), produces a procedure that composes *self-compose* with itself. Using the substitution evaluation rules and the definition of *fcompose*, this expression evaluates to

((**lambda** (f g) (**lambda** (x) (g (f x)))) *self-compose self-compose*)

which evaluates to (**lambda** (x) (*self-compose* (*self-compose x*))). Applying this to *inc* results in (*self-compose* (*self-compose inc*)). Substituting the definition of *self-compose*, we get:

```
(fcompose (fcompose inc inc) (fcompose inc inc))
```

Now, we can substitute the definition of *fcompose* for the outer application to get:

```
(lambda (x) ((fcompose inc inc) ((fcompose inc inc) x)))
```

This expression is applied to 1, producing ((*fcompose inc inc*) ((*fcompose inc inc*) 1)). Next, we substitute the definition of *fcompose* in the inner application to get:

```
((fcompose inc inc) (((lambda (f g) (lambda x) (g (f x))) inc inc) 1))
```

Using the application rule, this simplifies to ((*fcompose inc inc*) ((**lambda** (*x*) (*inc* (*inc x*))) 1)). Applying again, substituting 1 for *x*, we get:

```
((fcompose inc inc) (inc (inc 1)))
```

After performing the *inc* applications, this is ((*fcompose inc inc*) 3). The remaining application expressions are evaluated the same way, producing the final value of 5.

Exercise 4.3. Define a procedure *fcompose3* that takes three procedures as input, and produces as output a procedure that is the composition of the three input procedures. For example, ((*fcompose3 abs inc square*) −5) should evaluate to 36. Define *fcompose3* two different ways: once without using *fcompose*, and once using *fcompose*.

Solution. Without using *fcompose*:

```
(define (fcompose3 f1 f2 f3)
   (lambda (x) (f3 (f2 (f1 x)))))
```

Using *fcompose*:

```
(define (fcompose3 f1 f2 f3)
   (fcompose (fcompose f1 f2) f3))
```

Exercise 4.4. The *fcompose* procedure only works when both input procedures take one input. Define a *f2compose* procedure that composes two procedures where the first procedure takes two inputs, and the second procedure takes one input. For example, ((*f2compose* + *abs*) 3 −5) should evaluate to 2.

Solution.

```
(define (f2compose f g)
        (lambda (x y)
           (g (f x y))))
```

Exercise 4.5. How many different ways are there of choosing an unordered 5-card hand from a 52-card deck?

This is an instance of the "n choose k" problem (also known as the binomial coefficient): how many different ways are there to choose a set of k items from n items. There are n ways to choose the first item, $n - 1$ ways to choose the second, …, and $n - k + 1$ ways to choose the k^{th} item. But, since the order does not matter, some of these ways are equivalent. The number of possible ways to order the k items is $k!$, so we can compute the number of ways to choose k items from a set of n items as:

$$\frac{n*(n-1)*\cdots*(n-k+1)}{k!} = \frac{n!}{(n-k)!k!}$$

a. Define a procedure *choose* that takes two inputs, n (the size of the item set) and k (the number of items to choose), and outputs the number of possible ways to choose k items from n.

Solution.

```
(define (choose n k)
    (/ (factorial n) (* (factorial (- n k)) (factorial k))))
```

b. Compute the number of possible 5-card hands that can be dealt from a 52-card deck.

Solution.

```
> (choose 52 5)
2598960
```

c. [⋆] Compute the likelihood of being dealt a flush (5 cards all of the same suit). In a standard 52-card deck, there are 13 cards of each of the four suits. Hint: divide the number of possible flush hands by the number of possible hands.

Solution. The number of possible flushes for each suit is the number of ways to choose 5 cards from the 13 cards of each suit. So the total number of possible flushes is (∗ 4 (*choose* 13 5)). To compute the probability of being dealt a 5-card flush, we divide the number of ways to make a flush by the number of 5-card hands:

```
> (/ (* 4 (choose 13 5)) (choose 52 5))
33/16660
> (exact->inexact (/ (* 4 (choose 13 5)) (choose 52 5)))
0.0019807923169267707
```

So, you should expect to see a 5-card flush roughly once every 505 hands.

Exercise 4.6. Gauss, Karl Reputedly, when Karl Gauss was in elementary school his teacher assigned the class the task of summing the integers from 1 to 100 (e.g., $1+2+3+\cdots+100$) to keep them busy. Being the (future) "Prince of Mathematics", Gauss developed the formula for calculating this sum, that is now known as the *Gauss sum*. Had he been a computer scientist, however, and had access to a Scheme interpreter in the late 1700s, he might have instead defined a recursive procedure to solve the problem. Define a recursive procedure, *gauss-sum*, that takes a number n as its input parameter, and evaluates to the sum of the integers from 1 to n as its output. For example, (*gauss-sum* 100) should evaluate to 5050.

Solution.

```
(define (gauss-sum n)
    (if (= n 1) 1
        (+ n (gauss-sum (- n 1)))))
```

Exercise 4.7. [⋆] accumulate Define a higher-order procedure, *accumulate*, that can be used to make both *gauss-sum* (from Exercise 4.6) and *factorial*. The *accumulate* procedure should take as its input the function used for accumulation (e.g., ∗ for *factorial*, + for *gauss-sum*). With your *accumulate* procedure, ((*accumulate* +) 100) should evaluate to 5050 and ((*accumulate* ∗) 3) should evaluate to 6. We assume the result of the base case is 1 (although a more general procedure could take that as a parameter).

Hint: since your procedure should produce a procedure as its output, it could start like this:

```
(define (accumulate f)
  (lambda (n)
     (if (= n 1) 1
        ...
```

Solution.

```
(define (accumulate f)
    (lambda (n)
          (if (= n 1) 1
             (f n ((accumulate f) (- n 1))))))
```

Here are a few examples:

```
> ((accumulate +) 100)
5050
> ((accumulate +) 100)
5050
> ((accumulate (lambda (x y) (- x y))) 10)
5
```

Exercise 4.8. To find the maximum of a function that takes a real number as its input, we need to evaluate at all numbers in the range, not just the integers. There are infinitely many numbers between any two numbers, however, so this is impossible. We can approximate this, however, by evaluating the function at many numbers in the range.

Define a procedure *find-maximum-epsilon* that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *epsilon*, and produces as output the maximum value of *f* in the range between *low* and *high* at interval *epsilon*. As the value of *epsilon* decreases, *find-maximum-epsilon* should evaluate to a value that approaches the actual maximum value.

For example,

```
(find-maximum-epsilon (lambda (x) (* x (- 5.5 x))) 1 10 1)
```

evaluates to 7.5. And,

```
(find-maximum-epsilon (lambda (x) (* x (- 5.5 x))) 1 10 0.01)
```

evaluates to 7.5625.

Solution. We start from the *find-maximum* definition from the example, and add an extra parameter:

```
(define (find-maximum-epsilon f low high epsilon)
  (if (>= low high)
    (f low)
    (bigger (f low) (find-maximum-epsilon f (+ low epsilon) high epsilon))))
```

The most important change is replacing = in the if expression predicate with >=. Otherwise, it is possible the exact matching value is skipped and the procedure will continue to evaluate forever without every reaching the base case!

Exercise 4.9. [⋆] The *find-maximum* procedure we defined evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure that finds the input in the range that produces the maximum output value. For example, (*find-maximum-input inc* 1 10) should evaluate to 10 and (*find-maximum-input* (**lambda** (*x*) (∗ *x* (− 5.5 *x*))) 1 10) should evaluate to 3.

Solution. This one gets more complicated. We need an extra parameter to keep track of the input that produces the maximum output value found so far. To keep the interface the same, we define a worker procedure that takes the extra parameter, starting with the value of *low*.

```
(define (find-maximum-input f low high)
  (define (find-maximum-input-worker f low high best)
    (if (= low high)
        (if (> (f low) (f best))
            low
            best)
        (find-maximum-input-worker
         f (+ low 1) high
         (if (> (f low) (f best)) low best))))
  (find-maximum-input-worker f low high low))
```

Exercise 4.10. [⋆] Define a *find-area* procedure that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *epsilon*, and produces as output an estimate for the area under the curve produced by the function *f* between *low* and *high* using the *epsilon* value to determine how many regions to evaluate.

Solution. We estimate the area under the curve by summing the areas of each trapezoid formed by the points $(x, 0)$, $(x, f(x))$, $(x+\epsilon, 0)$, $(x+\epsilon, f(x+\epsilon))$. The area of a trapezoid is its length times its average height:

$$\epsilon \times \frac{(f(x+\epsilon) - f(x))}{2}.$$

```
(define (find-area f low high epsilon)
  (if (>= low high)
      0
      (+ (* (/ (+ (f low) (f (+ low epsilon))) 2) epsilon)
         (find-area f (+ low epsilon) high epsilon))))
```

Here are some examples:

```
> (find-area (lambda (x) 5) 0 5 0.001)
24.99999999999931
> (find-area (lambda (x) x) 0 5 0.001)
12.49999999999935
> (find-area (lambda (x) (sin x)) 0 (* 2 pi) 0.0001)
1.0372978290178584e−010
```

The answers are not exact for two reasons. The first is that epsilon is not infinitesimal (and never can be with a finite computation). The second is a minor bug in the code since it does not account for the situation where (+ *low epsilon*) exceeds *high*. Fixing this problem is left as (another) exercise for the reader.

Exercise 4.11. Show the structure of the *gcd-euclid* applications in evaluating (*gcd-euclid* 6 9).

Solution. The first application is (*gcd-euclid* 6 9). Since the predicate is false, the alternate clause is evaluated. It leads to the application (*gcd-euclid* 9 6). In this application, the predicate is still false, and the alernate clause is valuated. Since (*modulo* 9 6) evaluates to 3, this results in the application (*gcd-euclid* 6 3). For this application, the predicate is (= (*modulo* 6 3) 0) which is *true*. Hence, the expression evaluates to the consequence clause, *b* which has the value 3.

Exercise 4.12. [⋆] Provide a convincing argument why the evaluation of (*gcd-euclid a b*) will always finish when the inputs are both positive integers.

Solution.

Exercise 4.13. Provide an alternate definition of *factorial* that is tail recursive. To be tail recursive, the expression containing the recursive application cannot be part of another application expression. (Hint: define a *factorial-helper* procedure that takes an extra parameter, and then define *factorial* as (**define** (*factorial n*) (*factorial-helper n* 1)).)

Solution. Following the hint, we add an extra parameter to keep track of the working result:

```
(define (factorial n)
    (define (factorial-helper n v)
        (if (= n 1) v
            (factorial-helper (− n 1) (∗ v n))))
    (factorial-helper n 1))
```

Exercise 4.14. Provide a tail recursive definition of *find-maximum*.

Solution.

```
(define (find-maximum-tail f low high)
   (define (find-maximum-helper f low high best)
      (if (= low high)
         (bigger (f low) best)
         (find-maximum-helper f (+ low 1) high (bigger (f low) best))))
   (find-maximum-helper f low high (f low)))
```

Exercise 4.15. [⋆⋆] Provide a convincing argument why it is possible to transform any recursive procedure into an equivalent procedure that is tail recursive.

Solution.

Exercise 4.16. This exercise tests your understanding of the (*factorial* 2) evaluation.

a. In step 5, the second part of the application evaluation rule, Rule 3(b), is used. In which step does this evaluation rule complete?

b. In step 11, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

c. In step 25, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

d. To evaluate (*factorial* 3), how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?

e. [$\star$] To evaluate (*factorial n*) for any positive integer *n*, how many times would Evaluation Rule 2 be used to evaluate the name *factorial*?

Exercise 4.17. For which input values *n* will an evaluation of (*factorial n*) eventually reach a value? For values where the evaluation is guaranteed to finish, make a convincing argument why it must finish. For values where the evaluation would not finish, explain why.