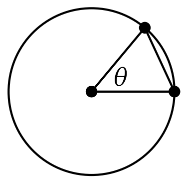
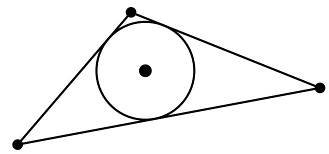


(a) The unit circle and the chord spanned by an angle θ.



(b) The circle inscribed in a triangle.

Figure 1: Diagrams accompanying Problem 1.

1. (**10 points.**) Define SCHEME functions with the following specifications.

   (a) (**2 points.**) The length of the chord spanned by the angle θ in the unit circle is $2\sin(\theta/2)$. Define a SCHEME function `chord-length` so that `(chord-length theta)` returns the length of the chord in the unit circle spanned by this angle. Use the built-in function `sin` which computes the sin.

```
(define (chord-length theta)
  (* 2 (sin (/ theta 2))))
```

   (b) (**3 points.**) The radius R of the circle inscribed in a triangle with edge lengths A, B, and C is given by the formula

$$R = \sqrt{\frac{(S-A)(S-B)(S-C)}{S}} \qquad \text{where} \qquad S = \frac{A+B+C}{2}.$$

   Define a SCHEME function `iradius` which takes three parameters for the side lengths (perhaps call them A, B, and C) and returns the radius as given by the formula above. (You may use the built-in scheme function `sqrt` for this purpose. A `let` construct can save you a lot of ink.)

```
(define (iradius A B C)
  (let ((S (/ (+ A B C) 2)))
    (sqrt (/ (* (- S A)
                (- S B)
                (- S C))
             S))))
```

   (c) (**2 points.**) Define a recursive SCHEME function `harmonic` so that `(harmonic n)` returns the nth harmonic number, equal to

$$\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}.$$

   (Thus `(harmonic 2)` should return $3/2 = 1 + 1/2$.)

```
(define (harmonic n)
  (if (= n 1)
      1
      (+ (/ 1 n) (harmonic (- n 1)))))
```

(d) (**3 points.**) A well-studied sum in number theory is the sum of the reciprocals of the primes:

$$\sum_{\text{prime } p} \frac{1}{p} = \frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \cdots.$$

Give a SCHEME program `prime-reciprocals` so that `(prime-reciprocals n)` returns the sum of the reciprocals of all prime numbers between 2 and n. For example `(prime-reciprocals 5)` should return $1/2 + 1/3 + 1/5$ because 2, 3, and 5 are the only prime numbers less than or equal to 5. Note that `(prime-reciprocals 6)` would actually return exactly the same value.

You may assume in your solution that `prime?` is already defined, so that `(prime? n)` will return `#t` when n is a prime number and `#f` otherwise. You do not need to implement this.

For completeness, here is one method of testing for primality (you did not need to implement this in your solutions):

```
(define (divides a b) (= (modulo b a) 0))
(define (smooth n k)
  (and (>= k 2)
       (or (divides k n)
           (smooth n (- k 1)))))
(define (prime? p)
  (not (smooth p (floor (sqrt p)))))
```

Assuming a procedure for testing primality, one solution:

```
(define (prime-reciprocals n)
  (cond ((< n 2) 0)
        ((prime? n) (+ (/ 1 n)
                       (prime-reciprocals (- n 1))))
        (else (prime-reciprocals (- n 1)))))
```

2. (**10 points.**) We discussed the *golden ratio* $\phi \approx 1.6180\ldots$ in one of our problem sets. In this problem, you will develop a SCHEME program to compute a good approximation of this (irrational) number. Among many other interesting properties, ϕ is the positive root of the simple polynomial $p(X) = X^2 - X - 1$ and, as a result, there is a general method called the "Newton-Raphson method" for giving iterative approximations to ϕ. Specifically, if $g \geq 1$ is a "guess" for the value of ϕ, the value

$$\texttt{improve}(g) = \frac{g^2+1}{2g-1}$$

will always be closer to ϕ than g was (unless $g = \phi$, in which case it is unchanged by `improve`).

How can we measure how well a particular guess g approximates the real value of ϕ? One approach is to use the fact that ϕ is a root of the polynomial $p(X) = X^2 - X - 1$, so $p(\phi) = 0$. To check that g is a good approximation for ϕ we will check to see that $p(g)$ is close to zero.

This provides us a way to generate increasingly good approximations to ϕ and a way to tell if we are close to ϕ; taken together, we can use these tools to compute a good approximation for ϕ.

(a) (**1 points.**) Write a SCHEME definition for the function `improve` which, given a value g, returns the value $(g^2 + 1)/(2g - 1)$.

```
(define (improve g)
  (/ (+ (* g g) 1)
     (- (* 2 g) 1)))
```

(b) (**2 points.**) In light of the discussion above, one way to find a good approximation to ϕ is to consider the sequence:

$$a_0 = 1, \qquad \text{and} \qquad a_{n+1} = \texttt{improve}(a_n) = \frac{a_n^2 + 1}{2a_n - 1} \quad \text{for } n \geq 1.$$

For large values of n, the quantity a_n is very close to ϕ. Give a SCHEME function `phi-approximant` so that `(phi-approximant n)` returns the value a_n given above. (You can save yourself some ink by using your `improve` function from the previous problem.)

```
(define (phi-approximant n)
  (if (= n 0)
      1
      (improve (phi-approximant (- n 1)))))
```

(c) (**5 points.**) We would prefer to compute an approximation to ϕ with *guaranteed accuracy*. Specifically, now we consider the problem of computing a guess g for ϕ so that $|p(g)| \leq$ `accuracy`, where `accuracy` is a parameter set by the user (and $p(X) = X^2 - X - 1$ is the polynomial above).

Write a function `phi-approx` so that `(phi-approx accuracy)` returns an approximation g to ϕ so that $|g^2 - g - 1| \leq$ `accuracy`. (A typical value for `accuracy` might be .001.)

(Hint: I suggest that you write a function `phi-iterate` which takes two arguments: a current guess g for the value of ϕ and the error tolerance `accuracy`. The function should return a value v for which $|p(v)| \leq$ `accuracy`. If the guess g that is provided to `phi-iterate` is already a good enough approximation, it can be returned. Otherwise, this guess needs to be improved...)

```
(define (phi-approx accuracy)
  (define (p x) (+ (* x x)
                   (* -1 x)
                   -1))
  (define (iterate g)
    (if (< (abs (p g)) accuracy)
        g
        (iterate (improve g))))
  (iterate 1))
```

(d) (**2 points.**) Explain what it means for a SCHEME program to be *tail recursive*. If your solution to (**??**) is tail recursive, indicate why. If not, explain how it can be restructured so that it is tail recursive.

A procedure is tail recursive if all recursive calls generate values that are directly returned by the procedure, without further computation. Indeed the solution above to (**??**) is tail recursive because the recursive call to `iterate` generates a value that is returned without further processing.

3. (**10 points.**) Scoping, recursion, and environment semantics.

(a) (**3 points.**) Recall the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for larger n. Consider the following standard version of `fibonacci` which computes these numbers:

```
(define (fibonacci n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fibonacci (- n 1))
                 (fibonacci (- n 2))))))
```

One "problem" with this formulation is that the interpreter ends up repeating a lot of work. For example, a call to `(fibonacci 5)` will eventually generate two different calls to `(fibonacci 3)`, so that the interpreter computes this twice. Note that, for $n \geq 3$,

$$F_n = F_{n-1} + F_{n-2} = (F_{n-2} + F_{n-3}) + F_{n-2} = 2F_{n-2} + F_{n-3},$$

which we obtain by using the rule $F_{n-1} = F_{n-2} + F_{n-3}$. This gives a different recursive relationship between the Fibonacci numbers. This recursive rule looks awesome—it automatically reflects the fact that F_{n-2} is going to be computed twice, and suggests a new program for the Fibonacci numbers:

```
(define (awesome-fibonacci n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (* 2 (awesome-fibonacci (- n 2)))
                 (awesome-fibonacci (- n 3))))))
```

This is a good idea, but this program doesn't correctly compute large Fibonacci numbers. In fact, even `(awesome-fibonacci 5)` doesn't work. Describe what goes wrong, and explain how to fix it. This is an interesting situation–even though the program only makes two different recursive calls, it really needs three bases cases to operate correctly. Note, in particular, that the program does not correctly work for $n = 2$, because this generates a recursive call to $n = -1$, which will never terminate. Moreover, it turns out that this is not an "isolated phenomenon—calls to larger values (e.g., 5) can also generate calls for the case $n = 3$. To fix it, you can add a further case to the `cond` statement to handle $n = 3$.

(b) (**2 points.**) Consider the following declaration:

```
(define (f x)
  (define (g y) (+ x y))
  (let ((x 100)
        (y 200))
    (g (+ x 10))))
```

After this, what would `(f 1000)` return?

It would return 1110.

(c) (**2 points.**) To what does the following expression evaluate?

```
(let ((x 10)
      (y 20)
      (z 40))
  (let ((x (+ y 1))
        (y (+ x 2))
        (z (lambda (z) (* z y))))
    (z y)))
```

It would evaluate to 240.

(d) (**3 points.**) Each of the following three functions, given the positive integer k, was designed with the intention to return the value 3^k using the fact that $3^k = 3 \times 3^{k-1} = 3^{k-1} + 3^{k-1} + 3^{k-1}$.

```
(define (three-power k)
  (if (= k 0) 1
      (+ (three-power (- k 1))
         (three-power (- k 1))
         (three-power (- k 1)))))
```

and

```
(define (three-power-let k)
   (let ((prev (three-power-let (- k 1))))
     (if (= k 0) 1
         (+ prev prev prev))))
```

and

```
(define (three-power-nested-let k)
  (if (= k 0) 1
      (let ((prev (three-power-nested-let (- k 1))))
        (+ prev prev prev))))
```

For each program, decide for which of the three values for k in the set $\{1, 5, 500\}$ you would expect the program to return the correct value 3^k in an hour. Explain.

The first one will generate approximately 3^k recursive calls, so it can handle $k = 1$ and 5 but not $k = 500$. The last one will generate approximately k recursive calls, and is efficient enough to compute even 3^{500}. The second one will not terminate for any value of k.

4. (**10 points.**) The *Pell equation* has the form

$$x^2 - ny^2 = 1,$$

where n is a positive integer (so $n \in \{1, 2, 3, \ldots\}$). A famous question in number theory is the following: fixing a positive integer $n \in \{1, 2, \ldots\}$, are there *integer* values for x and y that solve the Pell equation? For example, if $n = 2$, we can solve the Pell equation by taking $x = 3$ and $y = 2$ because $3^2 - 2 \cdot 2^2 = 1$. As another example, if $n = 5$, the Pell equation is solved by $x = 9$ and $y = 4$ because $9^2 - 5 \cdot 4^2 = 1$. In this problem, you will write a scheme program to find solutions to the Pell equation.

(a) (**3 points.**) Write a scheme function `perfect-square?` so that `(perfect-square? n)` returns `#t` if n is an integer perfect square and `#f` otherwise. (Hint: A good way to get started is to write a function `test-upto` so that `(test-upto n k)` returns `#t` if there is a square root of n in the set $\{1, \ldots, k\}$, and `#f` otherwise.)

```
(define (perfect-square? n) (let ((int-root (round (sqrt n))))
                              (= n (* int-root int-root))))
```

or

```
(define (perfect-square? n)
  (define (test k)
    (and (> k 0)
         (or (= n (* k k))
             (test (- k 1)))))
  (test n))
```

(b) (**2 points.**) If we fix particular values for n and x, then there is an easy way to tell if there is a solution for Pell's equation in the remaining variable y. You can check that for fixed choices of n and x, Pell's equation can be solved precisely when $(x^2 - 1)/n$ is a perfect square.

Use this to write a SCHEME function `pell-solution` so that `(pell-solution n x)` returns `#t` if there is an integer value of y that solves Pell's equation for the given values of n and x, and `#f` otherwise. (You may use the `perfect-square?` function above above even if you did not solve it.)

```scheme
(define (pell-solution n x)
  (perfect-square? (/ (- (* x x) 1) n)))
```

(c) (**5 points.**) Define a SCHEME function `pell-solve` so that `(pell-solve n)` returns the smallest positive integer value of x for which there is a solution to the Pell equation (for this value n).

```scheme
(define (pell-solve n)
  (define (try k)
    (if (pell-solution n k)
        k
        (try (+ k 1))))
  (try 1))
```