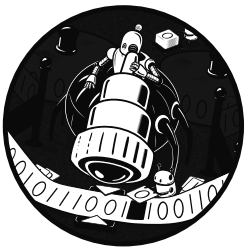
3

FUNCTION FUNDAMENTALS



In the last chapter, we introduced you to Racket's basic numerical operations. In this chapter, we'll explore the core ideas that form the subject of functional programming.

What Is a Function?

A *function* can be thought of as a box with the following characteristics: if you push an object in one side, an object (possibly the same, or not) comes out the other side; and for any given input item, the same output item comes out. This last characteristic means that if you put a triangle in one side and a star comes out the other, the next time you put a triangle in, you will also get a star out (see Figure 3-1). Unfortunately, Racket doesn't have any built-in functions that take geometric shapes as input, so we'll need to settle for more-mundane objects like numbers or strings.

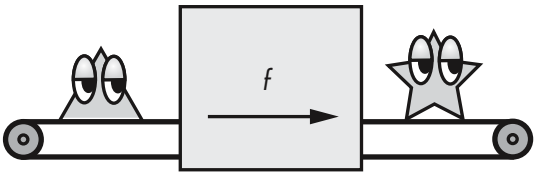


Figure 3-1: How a function works

Lambda Functions

In its most basic form, a function in Racket is something produced by a *lambda expression*, designated by the Greek letter λ. This comes from a mathematical discipline called lambda calculus, an arcane world we won't explore here. Instead, we'll focus on practical applications of lambda expressions. Lambda functions are intended for short simple functions that are immediately applied, and hence, don't need a name (they're anonymous). For example, Racket has a built-in function called `add1` that simply adds 1 to its argument. A Racket lambda expression that does the same thing looks like this:

```
(lambda (x) (+ 1 x))
```

Racket lets you abbreviate `lambda` with the Greek symbol λ, and we'll frequently designate it this way. You can enter λ in DrRacket by selecting it from the Insert menu or using the keyboard shortcut CTRL-\. We could rewrite the code above to look like this:

```
(λ (x) (+ 1 x))
```

To see a lambda expression in action, enter the following in the interactions pane:

```
> ((λ (x y) (+ (* 2 x) y)) 4 5)
13
```

Notice that instead of a function name as the first element of the list, we have the actual function. Here 4 and 5 get passed to the lambda function for evaluation.

An equivalent way of performing the above computation is with a `let` form.

```
> (let ([x 4]
        [y 5])
    (+ x (* 2 y)))
13
```

This form makes the assignment to variables `x` and `y` more obvious.

We can use lambda expressions in a more conventional way by assigning them to an identifier (a named function).

```
> (define foo (λ (x y) (+ (* 2 x) y)))
> (foo 4 5)
13
```

Racket also allows you to define functions using this shortcut:

```
> (define (foo x y) (+ (* 2 x) y))
> (foo 4 5)
13
```

These two forms of function definition are entirely equivalent.

Higher-Order Functions

Racket is a functional programming language. *Functional programming* is a programming paradigm that emphasizes a declarative style of programming without side effects. A *side effect* is something that changes the state of the programming environment, like assigning a value to a global variable.

Lambda values are especially powerful because they can be passed as values to other functions. Functions that take other functions as values (or return a function as a value) are known as *higher-order functions*. In this section, we'll explore some of the most commonly used higher-order functions.

The map Function

One of the most straightforward higher-order functions is the `map` function, which takes a function as its first argument and a list as its second argument, and then applies the function to each element of the list. Here's an example of the `map` function:

```
> (map (λ (x) (+ 1 x)) '(1 2 3))
'(2 3 4)
```

You can also pass a named function into `map`:

```
> (define my-add1 (λ (x) (+ 1 x)))
> (map my-add1 '(1 2 3)) ; this works too
'(2 3 4)
```

In the first example above, we take our increment function and pass it into `map` as a value. The `map` function then applies it to each element in the list `'(1 2 3)`.

It turns out that `map` is quite versatile. It can take as many lists as the function will accept as arguments. The effect is sort of like a zipper, where the list arguments are fed to the function in parallel, and the resulting values is a single list, formed by applying the function to the elements from each list. The example below shows `map` being used to add the corresponding elements of two equally sized lists together:

```
> (map + '(1 2 3) '(2 3 4))
'(3 5 7)
```

As you can see, the two lists were combined by adding the corresponding elements together.

The apply Function

The `map` function lets you apply a function to each item in a list individually. But sometimes, we want to apply all the elements of a list as arguments in a single function call. For example, Racket arithmetical operators can take multiple numeric arguments:

```
> (+ 1 2 3 4)
10
```

But if we try to pass in a list as an argument, we'll get an error:

```
> (+ '(1 2 3 4))
. . +: contract violation
  expected: number?
  given: '(1 2 3 4)
```

The + operator is only expecting numeric arguments. But not to worry. There's a simple solution: the `apply` function:

```
> (apply + '(1 2 3 4))
10
```

The `apply` function takes a function and a list as its arguments. It then *applies* the function to values in the list as if they were arguments to the function.

The foldr and foldl Functions

Yet another way to add the elements of a list together is with the `foldr` function. The `foldr` function takes a function, an initial argument, and a list:

```
> (foldr + 0 '(1 2 3 4))
10
```

Even though `foldr` produced the same result as `apply` here, behind the scenes it worked very differently. This is how `foldr` added the list together: 1 + (2 + (3 + (4 + 0))). The function "folds" the list together by performing its operation in a right-associative fashion (hence the `r` in `foldr`).

Closely associated with `foldr` is `foldl`. The action of `foldl` is slightly different from what you might expect. Observe the following:

```
> (foldl cons '() '(1 2 3 4))
'(4 3 2 1)

> (foldr cons '() '(1 2 3 4))
'(1 2 3 4)
```

One might have expected `foldl` to produce `'(1 2 3 4)`, but actually `foldl` performs the computation `(cons 4 (cons 3 (cons 2 (cons 1 '()))))`. The list arguments are processed from left to right, but the two arguments fed to `cons` are reversed—for example, we have `(cons 1 '())` and not `(cons '() 1)`.

The compose Function

Functions can be combined together, or *composed*, by passing the output of one function to the input of another. In math, if we have $f(x)$ and $g(x)$, they can be composed to make $h(x) = f(g(x))$ (in mathematics text this is sometimes designated with a special composition operator as $h(x) = (f \circ g)(x)$. We can do this in Racket using the `compose` function, which takes two or more functions and returns a new composed function. This new function works a bit like a pipeline. For example, if we want to increment a number by 1 and

square the result (that is, for any n compute $(n + 1)^2$), we could use following function:

```
(define (n+1_squared n) (sqr (add1 n)))
```

But `compose` allows this to be expressed a bit more succinctly:

```
> (define n+1_squared (compose sqr add1))
> (n+1_squared 4)
25
```

Even simpler . . .

```
> ((compose sqr add1) 4)
25
```

Please note that `add1` is performed first and then `sqr`. Functions are composed from right to left—that is, the rightmost function is applied first.

The filter Function

Our final example is `filter`. This function takes a predicate (a function that returns a Boolean value) and a list. The returned value is a list such that only elements of the original list that satisfy the predicate are included. Here's how we'd use `filter` to return the even elements of a list:

```
> (filter even? '(1 2 3 4 5 6))
'(2 4 6)
```

The `filter` function allows you to filter out items in the original list that won't be needed.

As you've seen throughout this section, our description of a function as a box is apt since it is in reality a value that can be passed to other functions just like a number, a string, or a list.

Lexical Scoping

Racket is a lexically scoped language. The Racket Documentation provides the following definition for *lexical scoping*:

> Racket is a lexically scoped language, which means that whenever an identifier is used as an expression, something in the textual environment of the expression determines the identifier's binding.

What's important about this definition is the term *textual environment*. A textual environment is one of two things: the *global environment*, or forms where identifiers are bound. As we've already seen, identifiers are bound in the global environment (sometimes referred to as the top level) with `define`. For example

```
> (define ten 10)
> ten
10
```

The values of identifiers bound in the global environment are available everywhere. For this reason, they should be used sparingly. Global definitions should normally be reserved for function definitions and constant values. This, however, is not an edict, as there are other legitimate uses for global variables.

Identifiers bound within a form will *normally* not be defined outside of the form environment (but see "Time for Some Closure" on page 57 for an intriguing exception to this rule).

Let's look at a few examples.

Previously we explored the lambda expression `((λ (x y) (+ (* 2 x) y )) 4 5)`. Within this expression, the identifiers `x` and `y` are bound to 4 and 5. Once the lambda expression has returned a value, the identifiers are no longer defined.

Here again is the equivalent `let` expression.

```
(let ([x 4]
      [y 5])
  (+ (* 2 x) y))
```

You might imagine that the following would work as well:

```
(let ([x 4]
      [y 5]
      [z (* 2 x)])
  (+ z y))
```

But this fails to work. From a syntactic standpoint there's no way to convert this back to an equivalent lambda expression. And although the identifier `x` is bound in the list of binding expressions, the value of `x` is only available inside the body of the `let` expression.

There is, however, an alternative definition of `let` called `let*`. In this case the following would work.

```
> (let* ([x 4]
         [y 5]
         [z (* 2 x)])
    (+ z y))
13
```

The difference is that with `let*` the value of an identifier is available immediately after it's bound, whereas with `let` the identifier values are only available after *all* the identifiers are bound.

Here's another slight variation where `let` *does* work.

```
> (let ([x 4]
        [y 5])
    (let ([z (* 2 x)])
      (+ z y)))
13
```

In this case the second `let` is within the lexical environment of the first `let` (but as we've seen, `let*` more efficiently encodes this type of nested construct). Hence `x` is available for use in the expression `(* 2 x)`.

Conditional Expressions: It's All About Choices

The ability of a computer to alter its execution path based on an input is an essential component of its architecture. Without this a computer cannot compute. In most programming languages this capability takes the form of something called a *conditional expression*, and in Racket it's expressed (in its most general form) as a `cond` expression.

Suppose you're given the task to write a function that returns a value that indicates whether a number is divisible by 3 only, divisible by 5 only, or divisible by both. One way to accomplish this is with the following code.

```
(define (div-3-5 n)
  (let ([div3 (= 0 (remainder n 3))]
        [div5 (= 0 (remainder n 5))])
    (cond [(and div3 div5) 'div-by-both]
          [div3 'div-by-3]
          [div5 'div-by-5]
          [else 'div-by-neither])))
```

The `cond` form contains a list of expressions. For each of these expressions, the first element contains some type of test, which if it evaluates to true, evaluates the second element and returns its value. Note that in this example the test for divisibility by 3 and 5 must come first. Here are trial runs:

```
> (div-3-5 10)
'div-by-5

> (div-3-5 6)
'div-by-3

> (div-3-5 15)
'div-by-both

> (div-3-5 11)
'div-by-neither
```

A simplified version of `cond` is the `if` form. This form consists of a single test (the first subexpression) that returns its second argument (after it's evaluated) if the test evaluates to true; otherwise it evaluates and returns the third argument. This example simply tests whether a number is even or odd.

```
(define (parity n)
  (if (= 0 (remainder n 2)) 'even 'odd))
```

If we run some tests:

```
> (parity 5)
'odd
> (parity 4)
'even
```

Both `cond` and `if` are expressions that return values. There are occasions where one simply wants to conditionally execute some sequence of steps if a condition is true or false. This usually involves cases where some side effect like printing a value is desired and returning a result is not required. For this purpose, Racket provides `when` and `unless`. If the conditional expression evaluates to true, `when` evaluates all the expressions in its body; otherwise it does nothing.

```
> (when (> 5 4)
    (displayln 'a)
    (displayln 'b))
a
b

> (when (< 5 4) ; doesn't generate output
    (displayln 'a)
    (displayln 'b))
```

The `unless` form behaves in exactly the same way as `when`; the difference is that `unless` evaluates its body if the conditional expression is not true.

```
> (unless (> 5 4) ; doesn't generate output
    (displayln 'a)
    (displayln 'b))

> (unless (< 5 4)
    (displayln 'a)
    (displayln 'b))
a
b
```

I'm Feeling a Bit Loopy!

Loops (or iteration) are the bread and butter of any programming language. With the discussion of loops, invariably the topic of *mutability* comes up. Mutability of course implies change. Examples of mutability are assigning values to variables (or worse, changing a value embedded in a data structure such as a vector). A function is said to be *pure* if no mutations (or side effects, like printing out a value or writing to a file—also forms of mutation) occur within the body of a function. Mutations are generally to be avoided if possible. Some languages, such as Haskell, go out of their way to avoid

this type of mischief. A Haskell programmer would rather walk barefoot through a bed of glowing, hot coals than write an impure function.

There are many good reasons to prefer pure functions, such as something called referential transparency (this mouthful simply means the ability to reason about the behavior of your program). We won't be quite so persnickety and will make judicious use of mutation and impure functions where necessary.

Suppose you're given the task of defining a function to add the first *n* positive integers. If you're familiar with a language like Python (an excellent language in its own right), you might implement it as follows.

```
def sum(n):
    s = 0
    while n > 0:
     ❶ s = s + n
     ❷ n = n - 1
    return s
```

This is a perfectly good function (and a fairly benign example of using mutable variables) to generate the desired sum, but notice both the variables `s` and `n` are modified ❶ ❷. While there's nothing inherently wrong with this, these assignments make the implementation of the function `sum` impure.

Purity

Before we get down and dirty, let's begin by seeing how we can implement looping using only pure functions. *Recursion* is the custom when it comes to looping or iteration in Racket (and all functional programming languages). A recursive function is just a function defined in terms of itself. Here's a pure (and simple) recursive program to return the sum of the first *n* positive integers.

```
(define (sum n)
❶ (if (= 0 n) 0
   ❷ (+ n (sum (- n 1)))))
```

As you can see, we first test whether `n` has reached 0 ❶, and if so we simply return the value 0. Otherwise, we take the current value of `n` and *recursively* add to it the `sum` of all the numbers less than `n` ❷. For the mathematically inclined, this is somewhat reminiscent of how a proof by mathematical induction works where we have a base case ❶ and the inductive part of the proof ❷.

Let's test it out.

```
> (sum 100)
5050
```

There's a potential problem with the example we have just seen. The problem is that every time a recursive call is made, Racket must keep track of

where it is in the code so that it can return to the proper place. Let's take a deeper look at this function.

```
(define (sum n)
  (if (= 0 n) 0
   ❶ (+ n (sum (- n 1)))))
```

When the recursive call to `sum` is made ❶, there's still an addition remaining to be done after the recursive call returns. The system must then remember where it was when the recursive call was made so that it can pick up where it left off when the recursive call returns. This isn't a problem for functions that don't have to nest very deeply, but for large depths of recursion, the computer can run out of space and fail in a dramatic fashion.

Racket (and virtually all Scheme variants) implement something called *tail call optimization* (the Racket community says this is simply the proper way to handle tail calls rather than an optimization, but *tail call optimization* is generally used elsewhere). What this means is that if a recursive call is the very last call being made, there's no need to remember where to return to since there are no further computations to be made within the function. Such functions in effect behave as a simple iterative loop. This is a basic paradigm for performing looping computations in the Lisp family of languages. You do, however, have to construct your functions in such a way as to take advantage of this feature. We can rewrite the summing function as follows.

```
(define (sum n)
  (define (s n acc)
 ❶ (if (= 0 n) acc
     ❷ (s (- n 1) (+ acc n))))
  (s n 0))
```

Notice that `sum` now has a local function called `s` that takes an additional argument called `acc`. Also notice that `s` calls itself recursively ❷, but it's the last call in the local function; hence tail call optimization takes place. This all works because `acc` accumulates the sum and passes it along as it goes. When it reaches the final nested call ❶, the accumulated value is returned.

Another way to do this is with a named `let` form as shown here.

```
(define (sum n)
  (let loop ([n n] [acc 0])
    (if (= 0 n) acc
        (loop (- n 1) (+ acc n)))))
```

The named `let` form, similar to the normal `let`, has a section where local variables are initialized. The expression `[n n]` may at first appear puzzling, but what it means is that the first `n`, which is local to the `let`, is initialized with the `n` that the `sum` function is called with. Unlike `define`, which simply binds an identifier with a function body, the named `let` binds the identifier (in this case `loop`), evaluates the body, and returns the value resulting from calling the function with the initialized parameter list. In this example the

function is called recursively (which is the normal use case for a named `let`) as indicated by the last line in the code. This is a simple illustration of a side-effect-free looping construct favored by the Lisp community.

The Power of the Dark Side

Purity is good, as far as it goes. The problem is that staying pure takes a lot of work (especially in real life). It's time to take a closer look at the dreaded `set!` form. Note that an exclamation point at the end of any built-in Racket identifier is likely there as a warning that it's going to do something impure, like modify the program state in some fashion. A programming style that uses statements to change a program's state is said to use *imperative programming*. In any case, `set!` reassigns a value to a previously bound identifier. Let's revisit the Python `sum` function we saw a bit earlier. The equivalent Racket version is given below.

```
(define (sum n)
  (let ([s 0])   ; initialize s to zero
    (do ()       ; an optional initializer statement can go here
      ((< n 1))  ; do until this becomes true
      (set! s (+ s n))
      (set! n (- n 1)))
    s))
```

Racket doesn't actually have a `while` statement (this has to do with the expectation within the Lisp community that recursion *should* be the go-to method for recursion). The Racket `do` form functions as a `do-until`.

If you're familiar with the C family of programming languages, then you will see that the full form of the `do` statement actually functions much like the C `for` statement. One way to sum the first *n* integers in C would be as follows:

```
int sum(int n)
{
  int s = 0;
  for (i=1; i<= n; i++) // initialize i=1, set i = i+1 at each iteration
                        // do while i<= n
  {
      s = s + i;
  }
  return s;             // return s
}
\end{lstlisting

Here's the Racket equivalent:
\begin{lstlisting}[]
(define (sum n)
❶ (let ([s 0])
  ❷ (do ([i 1 (add1 i)])   ; initialize i=1, set i = i+1 at each iteration
```

```
❸ ((> i n) s)          ; do until i>n, then return s
❹ (set! s (+ s i)))))
```

In the above code we first initialize the local variable `s` (which holds our sum) to 0 ❶. The first argument to `do` ❷ initializes `i` (`i` is local to the `do` form) to 1 and specifies that `i` is to be incremented by 1 at each iteration of the loop. The second argument ❸ tests whether `i` has reached the target value and if so returns the current value of `s`. The last line ❹ is where the sum is actually computed by increasing the value of `s` with the current value of `i` via the `set!` statement.

The value of forms such as `do` with the `set!` statement is that many algorithms are naturally stated in a step-by-step fashion with variables mutated by equivalents to the `set!` statement. This helps to avoid the mental gymnastics needed to convert such constructs to pure recursive functions.

In the next section, we examine the `for` family of looping variants. Here we will see that Racket's `for` form provides a great deal of flexibility in how to manage loops.

The for Family

Racket provides the `for` form along with a large family of `for` variants that should satisfy most of your iteration needs.

A Stream of Values

Before we dive into `for`, let's take a look at a couple of Racket forms that are often used in conjunction with `for`: `in-range` and `in-naturals`. These functions return something we haven't seen before called a *stream*. A stream is an object that's sort of like a list, but whereas a list returns all its values at once, a stream only returns a value when requested. This is basically a form of *lazy evaluation*, where a value is not provided until asked for. For example, `(in-range 10)` will return a stream of 10 values starting with 0 and ending with 9. Here are some examples of `in-range` in action.

```
> (define digits (in-range 10))
> (stream-first digits)
0

> (stream-first (stream-rest digits))
1

> (stream-ref digits 5)
5
```

In the code above, `(in-range 10)` defines a sequence of values 0, 1, . . . , 9, but `digits` doesn't actually contain these digits. It basically just contains a specification that will allow it to return the numbers at some later time. When `(stream-first digits)` is executed, `digits` gives the first available value, which in this case is the number 0. Then `(stream-rest digits)` returns the stream containing the digits after the first, so that `(stream-first (stream-rest`

digits)) returns the number 1. Finally, stream-ref returns the *i*-th value in the stream, which in this case is 5.

The function in-naturals works like in-range, but instead of returning a specific number of values, in-naturals returns an infinite number of values.

```
> (define naturals (in-naturals))
> (stream-first naturals)
0

> (stream-first (stream-rest naturals))
1

> (stream-ref naturals 1000)
1000
```

How the stream concept is useful will become clearer as we see it used within some for examples. We'll also met some useful additional arguments for in-range.

for in the Flesh

Here's an example of for in its most basic form. The goal is to print each character of the string "Hello" on a separate line.

```
> (let* ([h "Hello"]
       ❶ [l (string-length h)])
    ❷ (for ([i (in-range l)])
      ❸ (display (string-ref h i))
       (newline)))
H
e
l
l
o
```

We capture the string-length ❶ and use this length with the in-range function ❷. for then uses the resulting stream of values to populate the identifier i, which is used in the body of the for form to extract and display the characters ❸. In the prior section it was pointed out that in-range produces a sequence of values, but it turns out that in the context of a for statement, a positive integer can also produce a stream as the following example illustrates.

```
> (for ([i 5]) (display i))
01234
```

The for form is quite forgiving when it comes to the type of arguments that it accepts. It turns out that there's a much simpler way to achieve our goal.

```
> (for ([c "Hello"])
    (display c)
    (newline))
H
e
l
l
o
```

Instead of a stream of indexes, we have simply provided the string itself. As we'll see, `for` will accept many built-in data types that consist of multiple values, like lists, vectors, and sets. These data types can also be converted to streams (for example, by `in-list`, `in-vector`, and so on), which in some cases can provide better performance when used with `for`. All expressions that provide values to the identifier that `for` uses to iterate over are called *sequence expressions*.

It's time to see how we can make use of the mysterious `in-naturals` form introduced above.

```
> (define (list-chars str)
    (for ([c str]
          [i (in-naturals)])
      (printf "~a: ~a\n" i c)))

> (list-chars "Hello")
0: H
1: e
2: l
3: l
4: o
```

The `for` form inside the `list-chars` function now has *two* sequence expressions. Such sequence expressions are evaluated in parallel until one of the expressions runs out of values. That is why the `for` expression eventually terminates, even though `in-naturals` provides an infinite number of values.

There is, in fact, a version of `for` that *does not* evaluate its sequence expressions in parallel: it's called `for*`. This version of `for` evaluates its sequence expressions in a nested fashion as the following example illustrates.

```
> (for* ([i (in-range 2 7 4)]
         [j (in-range 1 4)])
    (display (list i j (* i j)))
    (newline))
(2 1 2)
(2 2 4)
(2 3 6)
(6 1 6)
(6 2 12)
```

```
(6 3 18)
```

In this example we also illustrate the additional optional arguments that `in-range` can take. The sequence expression `(in-range 2 7 4)` will result in a stream that starts with the number 2, and increment that value by 4 with each iteration. The iteration will stop once the streamed value reaches one less than 7. So in this expression, `i` is bound to 2 and 6. The expression `(in-range 1 4)` does not specify a step value, so the default step size of 1 is used. This results in `j` being bound to 1, 2, and 3.

Ultimately, `for*` takes every possible combination of `i` values and `j` values to form the output shown.

Can You Comprehend This?

There is a type of notation in mathematics called set-builder notation. An example of set-builder notation is the expression $\{x^2 \mid x \in \mathbb{N}, x \leq 10\}$. This is just the set of squares of all the natural numbers between 0 and 10. Racket provides a natural (pun intended) extension of this idea in the form of something called a *list comprehension*. A direct translation of that mathematical expression in Racket would appear as follows.

```
> (for/list ([x (in-naturals)] #:break (> x 10)) (sqr x))
'(0 1 4 9 16 25 36 49 64 81 100)
```

The `#:break` keyword is used to terminate the stream generated by `in-naturals` once all the desired values have been produced. Another way to do this, without having to resort to using `#:break`, would be with `in-range`.

```
> (for/list ([x (in-range 11)]) (sqr x))
'(0 1 4 9 16 25 36 49 64 81 100)
```

If you only wanted the squares of even numbers, you could do it this way:

```
> (for/list ([x (in-range 11)] #:when (even? x)) (sqr x))
'(0 4 16 36 64 100)
```

This time the `#:when` keyword was brought into play to provide a condition to filter the values used to generate the list.

An important difference of `for/list` over `for` is that `for/list` does not produce any side effects and is therefore a pure form, whereas `for` is expressly for the purpose of producing side effects.

More Fun with for

Both `for` and `for/list` share the same keyword parameters. Suppose we wanted to print a list of squares, but don't particularly like the number 5. Here's how it could be done.

```
> (for ([n (in-range 1 10)] #:unless (= n 5))
    (printf "~a: ~a\n" n (sqr n)))
1: 1
```

```
2: 4
3: 9
4: 16
6: 36
7: 49
8: 64
9: 81
```

By using `#:unless` we've produced an output for all values, $1 \leq n < 10$, unless $n = 5$.

Sometimes it's desirable to test a list of values to see if they all meet some particular criteria. Mathematicians use a fancy notation to designate this called the universal quantifier, which looks like this $\forall$ and means "for all." An example is the expression $\forall x \in \{2, 4, 6\}, x \bmod 2 = 0$, which is literally interpreted as "for all x in the set $\{2, 4, 6\}$, the remainder of x after dividing by 2 is 0." This just says that the numbers 2, 4, and 6 are even. The Racket version of "for all" is `for/and`.

Feed the `for/and` form a list of values and a Boolean expression to evaluate the values. If each value evaluates to true, the entire `for/and` expression returns true; otherwise it returns false. Let's have a go at it.

```
> (for/and ([x '(2 4 6)]) (even? x))
#t

> (for/and ([x '(2 4 5 6)]) (even? x))
#f
```

Like `for`, `for/and` can handle multiple sequence expressions. In this case, the values in each sequence are compared in parallel.

```
> (for/and ([x '(2 4 5 6)]
            [y #(3 5 9 8)])
    (< x y))
#t
> (for/and ([x '(2 6 5 6)]
            [y #(3 5 9 8)])
    (< x y))
#f
```

Closely related to `for/and` is `for/or`. Not to be outdone, mathematicians have a notation for this as well: it's called the existential quantifier, $\exists$. For example, they express the fact that there *exists* a number in the set $\{2, 7, 4, 6\}$ greater than 5 with the expression $\exists x \in \{2, 7, 4, 6\}, x > 5$.

```
> (for/or ([x '(2 7 4 6)]) (> x 5))
#t

> (for/or ([x '(2 1 4 5)]) (> x 5))
#f
```

Suppose now that you not only want to know whether a list contains a value that meets a certain criterion, but you want to extract the first value that meets the criterion. This is a job for `for/first`:

```
> (for/first ([x '(2 1 4 6 7 1)] #:when (> x 5)) x)
6

> (for/first ([x '(2 1 4 5 2)] #:when (> x 5)) x)
#f
```

The last example demonstrates that if there is no value that meets the criterion, `for/first` returns false.

Correspondingly, if you want the last value, you can use `for/last`:

```
> (for/last ([x '(2 1 4 6 7 1)] #:when (> x 5)) x)
7
```

The `for` family of functions is fertile ground for exploring parallels between mathematical notation and Racket forms. Here is yet another example. To indicate the sum of the squares of the integers from 1 to 10, the following notation would be employed:

$$S = \sum_{i=1}^{10} i^2$$

The equivalent Racket expression is:

```
> (for/sum ([i (in-range 1 11)]) (sqr i))
385
```

The equivalent mathematical expression for products is

$$p = \prod_{i=1}^{10} i^2$$

which in Racket becomes

```
> (for/product ([i (in-range 1 11)]) (sqr i))
13168189440000
```

Most of the `for` forms discussed above come in a starred version (for example `for*/list`, `for*/and`, `for*/or`, and so on). Each of these works by evaluating their sequence expressions in a nested fashion as described for `for*`.

Time for Some Closure

Suppose you had $100 in the bank and wanted to explore the effects of compounding with various interest rates. If you're not familiar with how compound interest works (and you very well should be), it works as follows: if

you have n_0 in a bank account that pays i periodic interest, at the end of the period you would have this:

$$n_1 = n_0 + n_0 i = n_0(1 + i)$$

Using your $100 deposit as an example, if your bank pays 4 percent ($i = 0.04$) interest per period (good luck getting that rate at a bank nowadays), you would have the following at the end of the period:

$$100 + 100 \cdot 4\% = 100(1 + 0.04) = 104$$

One way to do this is to create a function that automatically updates the balance after applying the interest rate. A clever way to compute this in Racket is with something called a *closure*, which we use in the following function:

```
(define (make-comp bal int)
  (let ([rate (add1 (/ int 100.0))])
❶ (λ () (set! bal (* bal rate))  (round bal))))
```

Notice that this function actually returns another function—the lambda expression (λ . . .) ❶—and that the lambda expression contains variables from the defining scope. We shall explain how this works shortly.

In the code above, we've defined a function called `make-comp` which takes two arguments: the starting balance and the interest rate percentage. The `rate` variable is initialized to (1 + i). Rather than return a number, this function actually returns another function. The returned function is designed in such a way that every time it's called (without arguments) it updates the balance by applying the interest and returns the new balance. You might think that once `make-comp` returns the lambda expression, the variables `bal` and `rate` would be undefined, but not so with closures. The lambda expression is said to *capture* the variables `bal` and `rate`, which are available within the lexical environment where the lambda expression is defined. The fact that the returned function contains the variables `bal` and `rate` (which are defined outside of the function) is what makes it a closure.

Let's try this out and see what happens.

```
> (define bal (make-comp 100 4))

> (bal)
104.0

> (bal)
108.0

> (bal)
112.0

> (bal)
117.0
```

As you can see, the balance is updated appropriately.

Another use for closures is in a technique called *memoization*. What this means is that we store prior computed values and if a value has already been computed, return the remembered value; otherwise go ahead and compute the value and save it for when it's needed again. This is valuable in scenarios where a function may be called repeatedly with arguments that have already been computed.

To facilitate this capability, something called a *hash table* or dictionary is typically used. A hash table is a mutable set of key-value pairs. A hash table is constructed with the function `make-hash`. Items can be stored to the hash table via `hash-set!` and retrieved from the table with `hash-ref`. We test whether the table already contains a key with `hash-has-key?`.

The standard definition for the factorial function is $n! = n(n - 1)!$. The obvious way to implement this in Racket is with the following.

```
(define (fact n)
  (if ( = 0 n) 1
      (* n (fact (- n 1)))))
```

This works, but every time you call `(fact 100)`, Racket has to perform 100 computations. With memoization, executing `(fact 100)` still requires 100 computations *the first time*. But the next time you call `(fact 100)` (or call `fact` for any value less than 100), Racket only has to look up the value in the hash table, which happens in a single step. Here's the implementation.

```
(define fact
  (let ([h (make-hash)]) ; hash table to contain memoized values
 ❶ (define (fact n)
     (cond [(= n 0) 1]
        ❷ [(hash-has-key? h n) (hash-ref h n)]
           [else
           ❸ (let ([f (* n (fact (- n 1)))])
             ❹ (hash-set! h n f)
              f)]))
 ❺ fact))
```

It's important to note that the outer `fact` function actually returns the inner `fact` function ❶. This is ultimately what gets executed when we call `fact 100`. It's this inner `fact` function, which captures the hash table, that constitutes the closure. First, it checks to see whether the argument to `fact` is one that is already computed ❷ and if so, returns the saved value. We still have to compute the value if it hasn't been computed yet ❸, but then we save it in case it's needed later ❹. The local `fact` function is returned as the value of the global `fact` function (sorry about using the same name twice).

Applications

Having introduced the basic programming constructs available in Racket, let's take a look at some applications spanning computer science, mathematics, and recreational puzzles.

I Don't Have a Queue

In this section we touch on Racket's *object-oriented programming* capability. Objects are like a deluxe version of the structures we met in Chapter 1.

Imagine early morning at a small-town bank with a single teller. The bank has just opened, and the teller is still trying to get set up, but a customer, Tom, has already arrived and is waiting at the window. Shortly, two other customers show up: Dick and Harry. The teller finally waits on Tom, then Dick and Harry in that order. This situation is a classic example of a *queue*. Formally, a queue is a first-in, first-out (FIFO) data structure. Racket comes with a built-in queue (several, in fact), but let's explore building one from scratch.

We can model a queue with a list. For example the line of folks waiting to see the teller can be represented by a single list: `(define q (list 'tom 'dick 'harry))`. But there's a problem. It's clearly easy to remove Tom from the head of the list and get the remainder of the list by using `car` (or `first`) and `cdr` (or `rest`):

```
> (car q)
'tom

> (set! q (cdr q))
> q
'(dick harry)
```

But what happens when Sue comes along? We could do the following:

```
> (set! q (append q (list 'sue)))
> q
'(dick harry sue)
```

But consider what happens if the list is very long, say 10,000 elements. The `append` function will create an entire new list containing all the elements from q and the one additional value `'sue`. One way to do this efficiently is to maintain a pointer to the last element in the list and instead of creating a new list change the `cdr` of the last node of the list to point to the list `(list 'sue)` (see Figure 3-2). About now alarm bells should be going off in your head. You should have an uneasy feeling that modifying a list structure is somehow wrong. And you'd be right. It's not even possible to do this with the normal Racket list structure since the `car` and `cdr` cells in a list pair are immutable and cannot be changed.

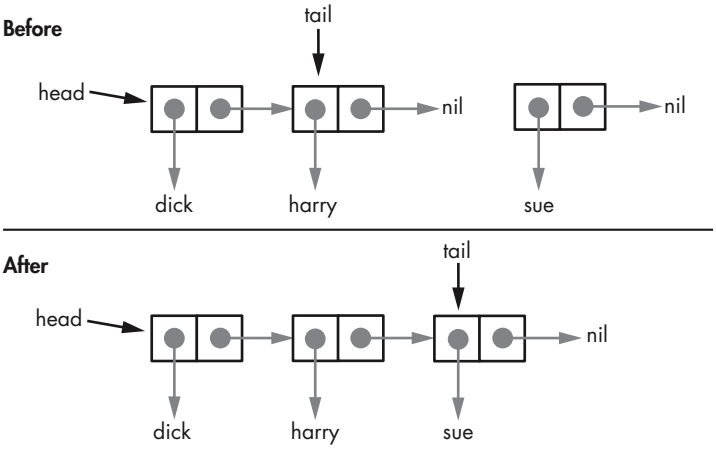


Figure 3-2: Mutable list

The traditional version of Scheme allow the elements of a cons node to be modified via `set-car!` and `set-cdr!` methods. Since these aren't defined in Racket, Racket guarantees that any identifier bound to a Racket list will have the same value for the life of the program.

There are still valid reasons why this capability may be needed. As we've seen, this functionality is needed for queues to ensure efficient operation. To accommodate this need, Racket provides a mutable cons cell that can be created with the `mcons` function. Each component of the mutable cons cell can be modified with `set-mcar!` and `set-mcdr!`. The functions `mcar` and `mcdr` are the corresponding accessor functions.

The reason modifying a list structure is bad is because if some other identifier is bound to the list, it will now have the modified list as its value, and maybe that's not what was intended. Observe the following.

```
> (define a (mcons 'apple 'orange))
> (define b a)
> a
(mcons 'apple 'orange)
> b
(mcons 'apple 'orange)

> (set-mcdr! a 'banana)
> a
(mcons 'apple 'banana)
> b
(mcons 'apple 'banana)
```

Although we only *seemed* to be changing the value of `a`, we also changed the value of `b`.

To avoid this potentially disastrous situation, we'll *encapsulate* the list in such a way that the list itself is not accessible, but we'll still be able to remove elements from the front of the list and add elements to the end of the list to implement our queue. Encapsulation is a fundamental component of object-

oriented programming. We'll dive right in by creating a class that contains all the functionality we need to implement our queue:

```
❶ (define queue%

  ❷ (class object%

    ❸ (init [queue-list '()])

    ❹ (define head '{})
      (define tail '{})

    ❺ (super-new)

    ❻ (define/public (enqueue val)
        (let ([t (mcons val '())])
          (if (null? head)
              (begin
                (set! head t)
                (set! tail t))
              (begin
                (set-mcdr! tail t)
                (set! tail t)))))

    ❼ (define/public (dequeue)
        (if (null? head) (error "Queue is empty.")
            (let ([val (mcar head)])
           ❽ (set! head (mcdr head))
              (when (null? head) (set! tail '()))
              val)))

      (define/public (print-queue)
        (define (prt rest)
          (if (null? rest)
              (newline)
              (let ([h (mcar rest)]
                    [t (mcdr rest)])
                (printf "~a " h)
                (prt t))))
        (prt head))

    ❾ (for ([v queue-list]) (enqueue v))))
```

Our class name is `queue%` (note that, by convention, Racket class names end with `%`). We begin with the class definition ❶. All classes must inherit from some parent class. In this case we're using the built-in class `object%` ❷. Once we've specified the class name and parent class, we specify the initialization parameters for the class ❸. This class takes a single, optional list

argument. If supplied, this list is used to initialize the queue ❾. Our class uses `head` and `tail` pointer identifiers, which we have to define ❹. Within the body of a class, `define` statements are not accessible from outside the class. This means that there is no way for the values of `head` or `tail` to be bound to an identifier outside of the class.

After a required call to the super class (in this case `object%`) ❺, we get into the real meat of this class: its methods. First we define a *public* class method called `enqueue` ❻. Public methods are accessible from outside the class. This method takes a single value, which is added to the end of the queue in a manner similar to our apple and banana example. If the queue is empty, then it initializes the `head` and `tail` identifiers with the mutable cons cell `t`.

The `dequeue` method ❼ returns the value at the head of the queue, but generates an error if the queue is empty. The `head` pointer is updated to point to the next value in the queue ❽.

To see all the values in the queue, we've also defined the method `print-queue`.

Let's see it in action.

```
> (define queue (new queue% [queue-list '(tom dick harry)]))

> (send queue dequeue)
'tom

> (send queue enqueue 'sue)
> (send queue print-queue)
dick harry sue

> (send queue dequeue)
'dick

> (send queue dequeue)
'harry

> (send queue dequeue)
'sue

> (send queue dequeue)
. . Queue is empty.
```

Class objects are created with the `new` form. This form includes the class name and any parameters defined by the `init` form in the class definition (see the class definition code ❸).

Unlike normal Racket functions and methods, an object method must be invoked with a `send` form. The `send` identifier is followed by the object name (`queue`), the method name, and any arguments for the method.

This example was just meant to expose the basics of Racket's object-oriented capabilities, but we'll be seeing much more of Racket's object prowess in the remainder of the text.

The Tower of Hanoi

The Tower of Hanoi is a puzzle that consists of three pegs embedded in a board, along with eight circular discs, each with a hole in the center. No two discs are the same size, and they are arranged on one of the pegs so that the largest is on the bottom and the rest are arranged such that a smaller disc is always immediately above a larger disc (See Figure 3-3).

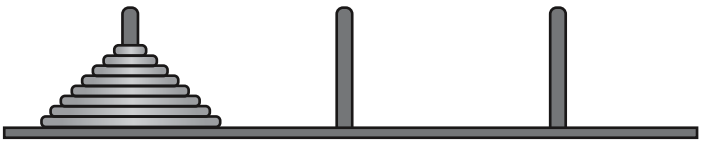


Figure 3-3: The Tower of Hanoi

W. W. Rouse Ball tells the following entertaining story about how this puzzle came about (see [3] and [8]).

> In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disc resting on the brass plate and the others getting smaller and smaller up to the top one. This is the tower of Brahma. Day and night unceasingly, the priest on duty transfers the disks from one diamond needle to another, according to the fixed and immutable laws of Brahmah, which require that the priest must move only one disk at a time, and he must place these discs on needles so that there never is a smaller disc below a larger one. When all the sixty-four discs shall have been thus transferred from the needle on which, at the creation, God placed them, to one of the other needles, tower, temple, and Brahmans alike will crumble into dust, and with a thunderclap the world will vanish.

This would take 2^{64} - 1 moves. Let's see how much time we have left until the world comes to an end. We assume one move can be made each second.

```
> (define moves (- (expt 2 64) 1))
> moves
18446744073709551615

> (define seconds-in-a-year (* 60 60 24 365.25))
> seconds-in-a-year
31557600.0

> (/ moves seconds-in-a-year)
```

```
584542046090.6263
```

This last number is about 5.84×10^{11} years. The universe is currently estimated to be a shade under 14×10^9 years old. If the priests started moving disks at the beginning of the universe, there would be about 570 billion years left, so you probably have at least enough time to finish reading this book.

As interesting as this is, our main objective is to use Racket to show how to actually perform the moves. We'll of course begin with a more modest number of disks, so let's start with just one disk. We'll number the pegs 0, 1, and 2. Suppose our goal is to move the disks from peg 0 to peg 2. With only one disk, we just move the disk from peg 0 to peg 2. If we have $n > 1$ disks, we designate the peg we're moving all the disks from as f, the peg we are moving to as t and the remaining peg we designate as u. The steps to solve the puzzle can be stated thusly:

1. Move n - 1 disks from f to u.
2. Move a single disk from f to t.
3. Move n - 1 disks from u to t.

While simple, this process is sufficient to solve the puzzle. Steps 1 and 3 imply the use of recursion. Here is the Racket code that implements these steps.

```
❶ (define (hanoi n f t)
  ❷ (if (= 1 n) (list (list f t))        ; only a single disk to move
     ❸ (let* ([u (- 3 (+ f t))]          ; determine unused peg
            ❹ [m1 (hanoi (sub1 n) f u)] ; move n-1 disks from f to u
            ❺ [m2 (list f t)]           ; move single disk from f to t
            ❻ [m3 (hanoi (sub1 n) u t)]); move disks from u to t
       ❼ (append m1 (cons m2 m3)))))
```

We pass `hanoi` the number of disks, the peg to move them from, and the peg to move to. Then we compute the moves required to implement steps one ❸, two ❹, and three ❺. Can you see why the `let` expression ❸ determines the unused peg? (Hint: think of the possible combinations. For example if $f = 1$ and $t = 2$, the `let` expression ❸ would give $u = 3 - (1 + 2) = 0$, the unused peg number.) The `hanoi` function returns a list of moves ❻. Each element of the list is a list of two elements that designate the peg to move from and the peg to move to. Here's an example of the output for three disks:

```
> (hanoi 3 0 2)
'((0 2) (0 1) (2 1) (0 2) (1 0) (1 2) (0 2))
```

Note that we have $2^3 - 1 = 7$ moves.

As can be seen from the comments in the code, the `hanoi` function is essentially a direct translation of the three-step solution process given earlier. Further, it provides a practical application of recursion where the function calls itself with a simpler version of the problem.

Fibonacci and Friends

The *Fibonacci sequence* of numbers is defined as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

where the next term in the sequence is always the sum of the two preceding terms. In some cases the initial zero is not considered part of the sequence. This sequence has a ton of interesting properties. We will only touch on a few of them here.

Some Interesting Properties

One interesting property of the Fibonacci sequence is that it's always possible to create a rectangle tiled with squares whose sides have lengths generated by the sequence, as seen in Figure 3-4. We'll see how to generate this tiling in Chapter 4.

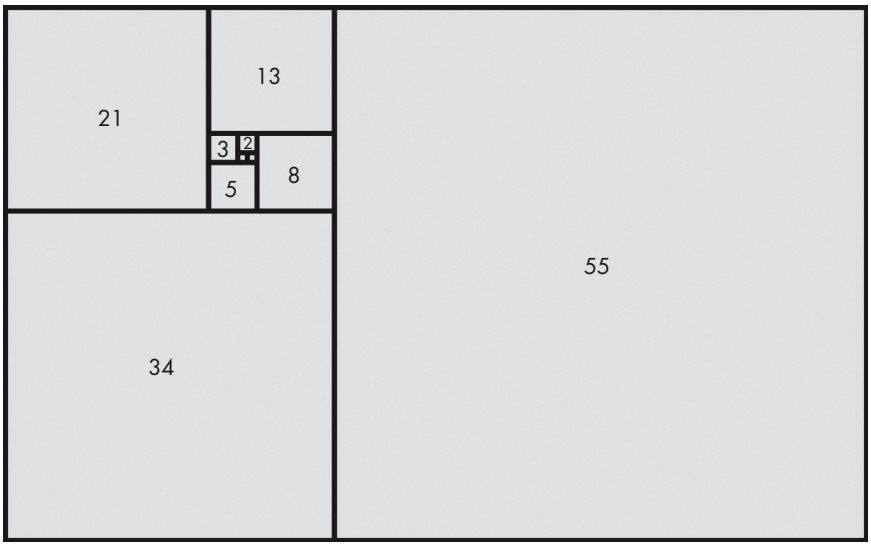


Figure 3-4: Fibonacci tiling

Johannes Kepler pointed out that the ratio of consecutive Fibonacci numbers approaches a particular number, designated by ϕ, which is known as the *golden ratio*:

$$\lim_{n \to \infty} \frac{F_{n+1}}{F_n} = \phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180339887\ldots$$

If you're not familiar with that $\lim n \to \infty$ business, it just means this is what you get when n gets bigger and bigger.

The number ϕ has many interesting properties as well. One example is the *golden spiral*. A golden spiral is a logarithmic spiral whose growth factor is ϕ, which means that it gets wider (or further from its origin) by a factor of ϕ for every quarter-turn. A golden spiral with initial radius 1 has the following polar equation:

$$r = \phi^{\theta \frac{2}{\pi}} \tag{3.1}$$

A plot of the golden spiral is shown in Figure 3-5. We'll show how this plot was produced in Chapter 4.

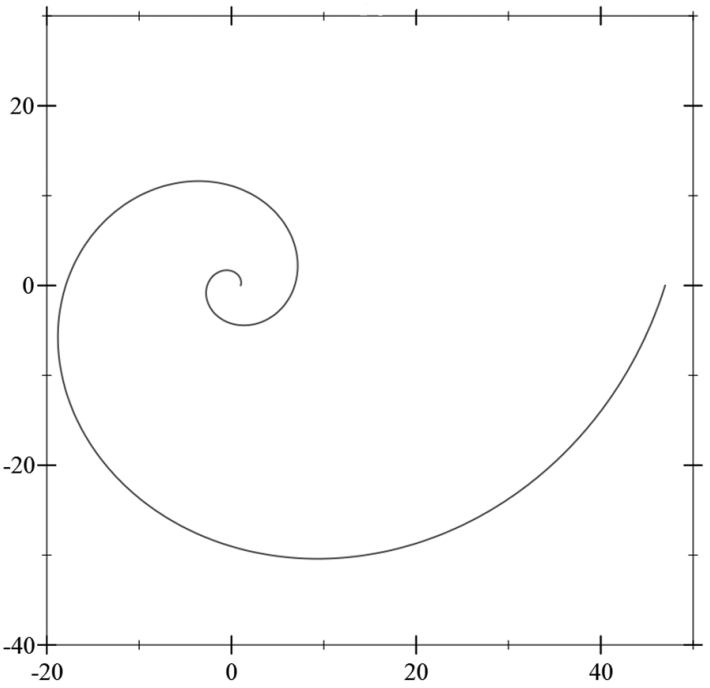


Figure 3-5: The golden spiral

Figure 3-6 illustrates an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling (in Chapter 4 we'll see how to superimpose this spiral onto a Fibonacci tiling).

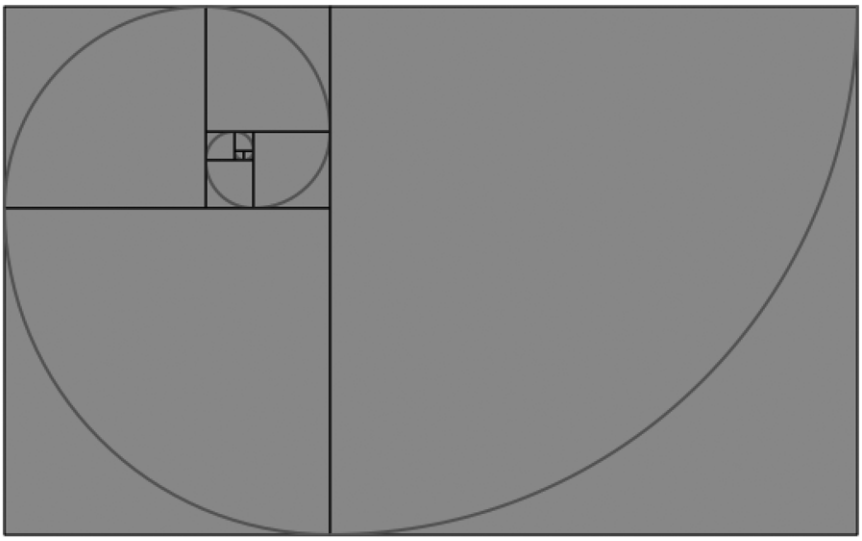


Figure 3-6: A golden spiral approximation

While these two versions of the golden spiral appear quite similar, mathematically they're quite different. This has to do with a concept called *curvature*. This has a precise mathematical definition, but for now, just think of it as the curviness of the path. The tighter the curve, the larger the curviness. The path described by Equation (3.1) has continuous curvature, while the Fibonacci spiral has discontinuous curvature. Figure 3-7 demonstrates the distinct difference in curvature these two paths possess.

We will make use of these properties in the following sections and in Chapter 4.

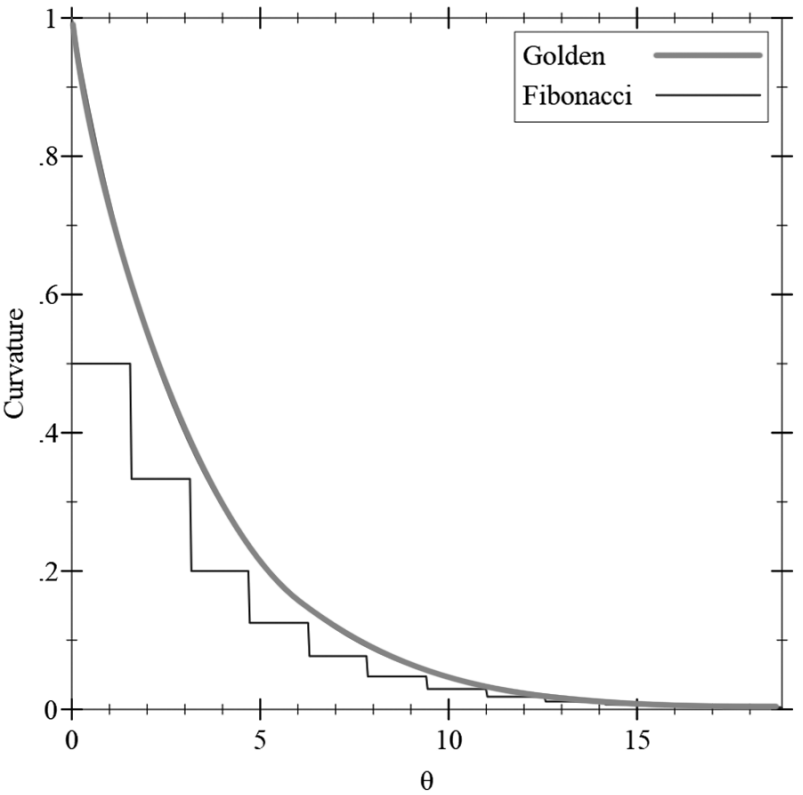


Figure 3-7: Curvature: golden vs. Fibonacci

Computing the Sequence

Mathematically the sequence F_n of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

In this section we'll explore three different methods of computing this sequence.

1. The No-brainer Approach. With the recurrence relation definition for the Fibonacci sequence, our first version practically writes itself. It's literally an exact translation from the definition to a Racket function.

```
(define (F n)
  (if (<= n 1) n
    (+ (F (- n 1)) (F (- n 2)))))
```

This code has the virtue of being extremely clear and simple. The only problem with this code is that it's terribly inefficient. The two nested calls cause the same value to be computed over and over. The end result is that the amount of computation grows exponentially with the size of n.

2. Efficiency Is King. Here we explore an ingenious method presented in the computer science classic *Structure and Interpretation of Computer Programs* [2]. The idea is to use a pair of integers initialized such that $a = F_1 = 1$ and $b = F_0 = 0$, and repeatedly apply the transformations:

$$a \leftarrow a + b$$
$$b \leftarrow a$$

It can be shown that after applying these transformations n times, we'll have $a = F_{n+1}$ and $b = F_n$. The proof is not difficult, and I've left it as an exercise for you. Here's the code to implement this solution:

```
(define (F n)
  (define (f a b c)
    (if (= c 0) b
        (f (+ a b) a (- c 1))))
  (f 1 0 n))
```

Due to tail call optimization, `f` recursively calls itself without the need to keep track of a continuation point. This works as an iterative process and only grows linearly with the size of n.

3. Memory Serves. In this version we use the memoization technique introduced in "Time for Some Closure" on page 57. To facilitate this, the code below uses a *hash table*. Recall that a hash table is a mutable set of key-value pairs, and it's constructed with the function `make-hash`. Items can be stored to the hash table via `hash-set!` and retrieved from the table with `hash-ref`. We test whether the table already contains a key with `hash-has-key?`.

```
(define F
  (let ([f (make-hash)]) ; hash table to contain memoized F values
    (define (fib n)
      (cond [(<= n 1) n]
            [(hash-has-key? f n) (hash-ref f n)]
            [else
              (let ([fn (+ (fib (- n 1)) (fib (- n 2)))])
                (hash-set! f n fn)
                fn)]))
    fib))
```

This code should be fairly easy to decipher. It's a straightforward application of memoization as seen in the `fact` example presented earlier.

And the Winner Is? It depends. You should definitely never use the first approach. Here's something to consider when comparing the second and the third: the second approach *always* requires n computations every time `F` is called for n. The third approach also requires n computations *the first time*

`F` is called for n. If you call it a second (or subsequent) time for n (or for any number less than n), it returns almost instantly since it simply has to look up the value in the hash table. There's a small space penalty for the third approach, but it's likely to be insignificant in most cases.

Binet's Formula. Before we leave the fascinating world of Fibonacci numbers and how to compute them, let's take a look at *Binet's Formula*:

$$F_n = \frac{\phi^n - \psi^n}{\phi - \psi} = \frac{\phi^n - \psi^n}{\sqrt{5}}$$

In this formula, the following is true:

$$\psi = \frac{1 - \sqrt{5}}{2} = 1 - \phi = -\frac{1}{\phi}$$

This formula provides us with yet another way of computing F_n. The following applies to all n:

$$\left| \frac{\psi^n}{\sqrt{5}} \right| < \frac{1}{2}$$

So the number F_n is the closest integer to $\frac{\psi^n}{\sqrt{5}}$. Therefore, if we round to the nearest integer, F_n can be computed by the following:

$$F_n = \left[\frac{\psi^n}{\sqrt{5}} \right]$$

The square brackets are used to designate the rounding function. In Racket, this becomes:

```
(define (F n)
  (let* ([phi (/ (add1 (sqrt 5)) 2)]
         [phi^n (expt phi n)])
    (round (/ phi^n (sqrt 5)))))
```

While Binet's formula is quite fast (since it does not require looping or recursion), the downside is that it only gives an approximate answer, where the other versions give an exact value.

Continued Fractions. The expression below is an example of a *continued fraction*. In this case, the fractional portion is repeated indefinitely. As we shall see, continued fractions have a surprising relationship to the Fibonacci sequence.

$$f = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}$$

Since the fraction does repeat infinitely, we may make the following substitution.

$$f = 1 + \frac{1}{f}$$

This substitution simplifies to the quadratic equation:

$$f^2 - f - 1 = 0$$

That equation has a couple of solutions. This is true:

$$f = \frac{1 \pm \sqrt{5}}{2}$$

Or, these are true:

$$\phi = \frac{1 + \sqrt{5}}{2} \text{ and } \psi = \frac{1 - \sqrt{5}}{2}$$

The question remains: which of these is the right value for f? Since ψ is negative, the answer must be ϕ. Thus . . .

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}$$

Bet you didn't see that coming.

The Insurance Salesman Problem

This problem is adapted from Flannery's *In Code* [7]. It's an example of a problem that could be solved by hand, but we can take advantage of Racket to do some of the tedious calculations. The problem is stated as follows.

A door-to-door insurance salesman stops at a woman's house and the following dialog ensues:

salesman: How many children do you have?
woman: Three.
salesman: And what are their ages?
woman: Take a guess.
salesman: How about a hint?
woman: Okay, the product of their ages is 36 and all the ages are whole numbers.
salesman: That's not much to go on. Can you give me another hint?
woman: The sum of their ages is equal to the number on the house next door.

The salesman immediately runs off, jumps over the fence, looks at the number on the house next door, scratches his head, and goes back to the woman.

salesman: Could you give me just one more hint?
woman: The oldest one plays the piano.

The salesman thinks for a bit, does some calculations, and figures out the children's ages. What are they?

At first blush, the hints seem a bit incongruous. Let's take them one at a time. First, we know that the product of the three ages is 36. Here is a program that generates all the unique combinations of three positive integers that have a product of 36.

```
#lang racket
(require math/number-theory)

❶ (define triples '())
  (define (gen-triples d1)
 ❷ (let* ([q (/ 36 d1)]
          [divs (divisors q)])
   ❸ (define (try-div divs)
       (when (not (null? divs))
      ❹ (let* ([d2 (car divs)] [d3 (/ q d2)])
        ❺ (when (<= d3 d2 d1)
          ❻ (set! triples (cons (list d3 d2 d1) triples)))
           (try-div (cdr divs)))))
     (try-div divs)))

❼ (for ([d (divisors 36)]) (gen-triples d))

  triples
```

While this code will not win any awards for efficiency, it is relatively simple and it gets the job done. We first define the variable `triples` which will contain the list of generated triples ❶. The processing actually begins when we call `gen-triples` ❺ for each divisor of 36 (provided by the `divisors` function defined in the *math/number-theory* library). This function then defines the quotient `q` ❷ of the divisor `d1` into 36. Following this we generate a list of divisors of `q` (`divs`, which of course also divide 36). We now come to the function `try-div` ❸, which does the bulk of the work. Then we get the first divisor (`d2`) of `q` ❹ and generate the third divisor (`d3`) by dividing `q` by `d2`. These divisors (`d1`, `d2`, and `d3`) are tested to see whether a satisfactory triple is formed (that is to ensure uniqueness, we make sure that they form an ordered sequence ❺) . If so, it's added to the list of triples ❻. Testing other divisors resumes on the following line. Running this program produces the following sets of triples: {1,1,36}, {1,2,18}, {1,3,12}, {1,4,9}, {2,2,9}, {1,6,6}, {2,3,6}, {3,3,4}.

This alone, of course, does not allow the salesman to determine the ages of the children. The second hint is that the sum of the ages equals the number on the house next door. Again we make use of Racket to generate the required sums.

```
(for ([triple triples]) (printf "~a: ~a\n" triple (apply + triple)))
```

From this, we have the following.

$$1 + 1 + 36 = 38$$
$$1 + 2 + 18 = 21$$
$$1 + 3 + 12 = 16$$
$$1 + 4 + 9 = 14$$
$$2 + 2 + 9 = 13$$
$$1 + 6 + 6 = 13$$
$$2 + 3 + 6 = 11$$
$$3 + 3 + 4 = 10$$

After looking at the number of the house next door, the salesman still does not know the ages. This means the ages must have been one of the two sets of numbers that sum to 13 (otherwise he would have known which set to select). Since the woman said "The oldest *one* plays the piano," the only possibility is the set of ages $\{2, 2, 9\}$, since the set $\{1, 6, 6\}$ would imply *two* oldest.

Summary

In this chapter, we introduced Racket's basic programming constructs and applied them to a variety of problem domains. So far our explorations have been confined to getting output in a textual form. Next, we will see how to add some bling to our applications by generating some graphical output.