Chapter 3

Functions and Prefix Notation

3.1 Testing Software

Suppose you have purchased a piece of software from someone, and you want to take it for a test drive to see if it works. Let's say it's the function that delivers sum of the first n reciprocals, given a number n. (You've seen this function before: page **??**.)

$$reciprocals(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

You could start with a few simple checks.

$$reciprocals(1) = 1 \qquad reciprocals(2) = \frac{3}{2} \qquad reciprocals(2) = \frac{11}{6}$$

That gets old fast. It would be nice to do a whole bunch of tests, not just a few specific ones. One way to describe a lot of tests at once is to find a way to specify a relationship among values of the function that you know will always be true. For example, you know that $reciprocal(n)$ is just like $reciprocal(n-1)$, except that it has one more term in the sum, namely the reciprocal of n.

$$reciprocal(n) = reciprocals(n-1) + 1/n, if n > 1$$

This relationship provides a different test for each possible value of n. You could tell the computer to run thousands of tests using random values

for n and have it report an error any of the tests fail. If the computer encounters errors, you will of course want your money back.

Since you do not expect the function to work unless the input is a non-negative integer, you need to tell the computer to choose random numbers of that kind. In fact, the test as it stands doesn't work when n is zero, so we need to fix that problem, either by avoiding zero in the testing process or by modifying to test to deal with zero.

$$reciprocal(n) = \begin{cases} 0 & \text{if } x = 0 \\ reciprocals(n-1) + 1/n & \text{otherwise} \end{cases}$$

After this modification, the computer can run thousands of tests automatically, just by choosing random, non-negative, integer values for n, computing $reciprocals(n)$ and $reciprocals(n-1) + 1/n$, and checking to make sure they are the same. If zero comes up as the random input, the computer compares $reciprocals(n)$ to zero instead of $reciprocals(n-1) + 1/n$.

It's nice to have an automated testing facility if you're buying software, and you want to make sure it works. It's also nice if you're building the software yourself. Then, you can deliver extensively tested software to your customers, and incur less risk of them asking for their money back.

We will be using a software development environment that facilitates automated testing. To use it, we define tests in the form of equations that that formulas using the software should satisfy, and tell the computer to generate random data and make sure the equations aren't violated.

The environment we will be using for this purpose is known as Dracula, and it's automated testing facility is called "doublecheck". The tool employs a notation that requires formulas to be written in "prefix" form. From our experience with arithmetic formulas, we are more accustomed to "infix" form, where operators come between their operands, as in the formula $(x + y)$. In prefix form, this formula would be $(+xy)$.

That seems simple enough, but it gets gradually more unfamiliar as the formulas get more complicated. For example, the formula sum of x and 3 times y, $(x + 3 * y)$, comes out as $(+x(*3y))$ in prefix form. It takes a while to get used to prefix notation, but not long. Some people end up liking it better than infix.

In prefix notation, the combined property of the *reciprocals* function specified above would take the following form.

Dr ACuLa, or "Dracula" for short, is the software development environment we will be using in to illustrate logic in action. Follow the instructions at the following URL to download and install it on your computer system: `http://www.ccs.neu.edu/home/cce/acl2/`. If that link fails, use look for "Dracula DrRacket" with your web search engine. It's a well-known system and should be easy to find. It's free, too.
Dracula provides a way to develop, test, reason about, and run programs written in the ACL2 dialect of a programming language called Lisp. It also provides a way to use a mechanized logic that is part of the ACL2 system to assist us in reasoning about our programs. At present we are discussing the testing process. Later, we will discuss developing programs, reasoning about them, and running them.

Aside 8: DrACuLa – a Software Development Environment

```
1 (defproperty reciprocals-test
2   (n :value (random-natural))
3   (= (reciprocals n)
4      (if (= n 0)
5          0
6          (+ (reciprocals (- n 1)) (/ 1 n)))))
```

As you can see, all of the formulas are in prefix notation, even the operator that sets up the definition: defproperty. It comes first, then comes the information it needs to carry out the tests. The first part of the data describes the random values to be used. In this case, values for n will be random natural numbers (that is, integers that are zero or bigger).

A specification of the test comes next. This one compares (reciprocals n) with 0, if n is zero, or with (+ (reciprocals (- n 1) (/ 1 n)), if n isn't zero. As you can see, the comparison operator, "=", comes first (it's prefix notation). Its first operand is the (reciprocals n) formula, and its second operand is one of two alternative formulas selected by an "if" operation. The "if" operator takes three operands and selects between the formulas supplied as its last two operands. It selects the second operand if the first operand has the value true and selects the third operand if the first operand has the value false.

Given this property definition, the doublecheck facility will perform many tests using random data and report successes and failures. You can

ask for as many tests as you like. The default is fifty tests, so if you don't say how many you want, you get fifty tests. The following property definition is the same as before, except that it calls for a hundred tests.

```
1 (include-book "doublecheck" :dir :teachpacks)
2
3 (defproperty reciprocals-test  :repeat 100
4   (n :value (random-natural))
5   (= (reciprocals n)
6      (if (= n 0)
7          0
8          (+ (reciprocals (- n 1)) (/ 1 n)))))
```

The "include-book" command at the beginning tells the computer what testing facility to use and where to find it. The tests will not take place without this directive.

Let's look at another example. Suppose the function "mod" delivers the remainder when dividing one number by another. Again, imagine that you have downloaded the function from your software supplier, and you are test driving it to decide whether to pay for the download or discard it. You might do some simple checks, such as using mod to compute the remainder when dividing by two. You expect this to be zero for even numbers and one for odd numbers.

```
1 (include-book "testing" :dir :teachpacks)
2
3 (check-expect (mod 12 2) 0)
4 (check-expect (mod 27 2) 1)
```

Here, we have used the "testing" system, which is a facility provided by our computing system to help with simple tests. It doesn't generate random data and run lots of tests, like the doublecheck facility, but is handy for building packages of sanity checks to make sure a function delivers the right answers for a few special cases.

You might also include some divisions by 3 in your package of tests.

```
1 (check-expect (mod 14 3) 2)
2 (check-expect (mod  7 3) 1)
3 (check-expect (mod 18 3) 0)
```

Probably, though, you will want to put "mod" through its paces on a large number of tests using doublecheck. For that, you will need to come up with a relationship that expresses an important property of the function. Since you know that the remainder doesn't change when you subtract the divisor from the dividend, at least when the divisor isn't zero and the dividend is at least as big as the divisor, you use that as a basis for some automated testing.

```
1 (defproperty mod-test
2   (divisor    :value (+ 1 (random-natural))
3    dividend   :value (+ divisor (random-natural)))
4   (= (mod dividend divisor)
5      (mod (- dividend divisor) divisor)))
```

Generating random data is an art. In this example, we have made sure the divisor isn't zero by adding one to a natural number. Since negative numbers aren't "natural" numbers, we know that adding one to a natural number ensures that the sum is non-zero. Using a similar trick, we make sure the dividend is at least as large as the divisor be choosing a random natural number and adding it too the divisor. That way, all of the random inputs will meet the constraints that our tests are based on.

To see the testing facility in action, define the "mod-test" property in a .lisp file, and use Dracula to run the tests. To put a .lisp file into operation with Dracula, just start the file running as you would any other program. When the Dracula window is ready, press the "Run" button in the tool bar on the upper right.

We might want to do some additional testing of the mod function. Another of its properties that we know from our experience with division as an arithmetic operation is that the remainder is always smaller than the divisor. That is, (mod dividend divisor) ¡ divisor. The following property definition makes it possible to use the doublecheck testing facility to check to make sure the mod function delivers values in this range.

```
1 (defproperty mod-upper-limit-test
2   (divisor    :value (+ 1 (random-natural))
3    dividend   :value (+ divisor (random-natural)))
4   (< (mod dividend divisor) divisor))
```

In this test, the property is not expressed as an equation, as in the previous case, but as an inequality based on the less-than operator ("¡").

As always, the formula puts the operation in the prefix position, in front of its operands. For practice, add this property to the .lisp file with the other tests and run it.

Another well-known fact about remainders in division is that they are non-negative integers (that is, natural numbers). We can use the logical-and operator ("and") combine the upper-limit test with the natural-number test ("natp") in one property definition.

```
1 (defproperty mod-range-test
2   (divisor    :value (+ 1 (random-natural))
3    dividend   :value (+ divisor (random-natural)))
4   (and (natp (mod dividend divisor))
5        (< (mod dividend divisor) divisor))
```

As you know, there are two parts to the result of dividing one number by another: the quotient and the remainder. The mod operator delivers the remainder, and an operator called "floor" delivers the quotient. From our experience with division, we known that the quotient is always strictly smaller than the dividend when the divisor is bigger than one and the dividend is a non-zero, natural number. The following test checks for that property. The random-value generator for the divisor makes sure the divisor exceeds one by adding two to a random natural number. A similar trick (which we also used in previous tests) ensures that the divisor is not zero.

```
1 (defproperty quotient-upper-limit-test
2   (divisor    :value (+ 2 (random-natural))
3    dividend   :value (+ 1 (random-natural)))
4   (= (+ (* divisor (floor dividend divisor))
5         (mod dividend divisor))
6      dividend))
```

Checking the result of a division is a matter of multiplying the quotient by the divisor and adding the remainder. If this fails to reproduce the dividend, something has gone wrong in the division process. The following property tests this relationship between the mod and floor operators. It needs to use the multiplication operator, which is denoted by an asterisk.

```
1 (defproperty division-test
2   (divisor    :value (+ 1 (random-natural))
3    dividend   :value (+ divisor (random-natural)))
```

```
4   (= (+ (* divisor (floor dividend divisor))
5         (mod dividend divisor))
6      dividend))
```

We hope by now you are starting to get comfortable with prefix notation. The exercises will give you a chance to practice.

Ex. 12 — Define a test of the floor operator the checks to make sure its value is a natural number when its operands are natural numbers, and the divisor (second operand) is not zero.

Ex. 13 — The "max" operator chooses the larger of two numbers: (max 2 7) is 7, (max 9 3) is 9. Define a property that tests to make sure (max x y) is greater than or equal to x and greater than or equal to y.