Chapter 9

Working with strings

9.1 Operations

Computer scientists use the term *string*, or *character string*, to mean a sequence of *characters*, which are basically keys on the keyboard. (There are a few exceptions: the arrow keys, function keys, "page up" and "page down" keys, *etc.* don't produce ordinary characters.) You've already learned how to type a literal string: a string starts and ends with double-quote marks, and in between them, you can put numbers, letters, spaces, parentheses, punctuation — anything except other double-quote marks. (In fact, if you *really* need to put a double-quote mark inside a string, you can do it by preceding it with a backslash, *e.g.* `"He said \"Hello,\" and I replied \"Hi there.\""` We won't need to do this very often.) In this section we'll learn to *operate* on strings just as we've already learned to operate on pictures and numbers.

The simplest imaginable string has no characters at all in between the quotation marks:

```
""
```

This is referred to, for obvious reasons, as the "empty string". Whenever you write a function that works on strings, make sure you include the empty string as one of the test cases.

Here are several of the most common operations on strings:

`string-append`

Contract:

```
; string-append :  string ...-> string
```

It takes in one or more[1] strings, puts them together end to end into a single string, and returns that. For example,

```
(string-append "hello" "there" "friend")
"hellotherefriend"
```

Note that it does *not* automatically put spaces in between: if you want spaces, you have to put them in:

```
(string-append "hello " "there" " " "friend")
"hello there friend"
```

[1] Actually, it even accepts no strings at all; it returns the empty string `""`.

`string-length`
Contract:

```
; string-length :  string -> integer
```

It tells you how many characters (letters, spaces, punctuation marks, *etc.*) are in the given string. For example,

```
(string-length "hellothere")
10
(string-length "Hi there, friend!")
17
```

`substring`
Contract:

```
; substring :  string integer(start) [integer(end)] -> string
```

(The "[integer(end)]" notation means that the third parameter is optional; in other words, the function takes in a string and one or two integers.) If there is only one integer parameter, `substring` chops off that many characters at the beginning. If there are two integer parameters, `substring` chops off everything *after* the first `end` characters, and then chops off the first `start` characters. The result will have length `end-start`, unless `end` is smaller than `start`, in which case you'll get an error message.

`number->string`
Contract:

```
; number->string :  number -> string
```

Converts a number to the sequence of characters used to print it out.

`string->number`
Contract:

```
; string->number :  string -> number
```

If the string can be interpreted as the sequence of characters used to print out a number, returns that number. If not, returns the special value `false` (about which we'll learn more in Chapter 13).

Practice Exercise 9.1.1 ***Play*** *with these.*

9.2 String variables and functions

You can define variables and functions with string values just as you can define them with image or numeric values.

Practice Exercise 9.2.1 ***Define*** *a variable named* `me` *whose value is your full name (first and last, with a space in between).*
Write several expressions *using this variable and the built-in functions* `string-append`*,* `string-length`*, and* `substring`*.*

Exercise 9.2.2 ***Develop a function*** *named* `repeat` *that takes in a string and returns that string appended to itself (*i.e. *the resulting string is twice as long).*

Exercise 9.2.3 ***Develop a function*** `chop-first-char` *that takes in a string and returns all but the first character. (For now, you may assume the string is non-empty; we'll drop this assumption later.)*

Exercise 9.2.4 ***Develop a function*** `first-char` *that takes in a string and returns a string of length 1, containing just the first character of the given string. (For now, you may assume the string is non-empty; we'll drop this assumption later.)*

Exercise 9.2.5 ***Develop a function*** *named* `last-half` *that takes in a string and returns the last half of it.*

Hint: Be sure to test your program on both even-length and odd-length strings. Also try some special cases like the empty string, `""`.

Exercise 9.2.6 ***Develop a function*** *named* `first-half` *that takes in a string and returns the first half of it.*

What happens if you concatenate the `first-half` of a string to the `last-half` of the same string? What *should* happen? Again, be sure to test this on both even-length and odd-length strings, and on the empty string.

Exercise 9.2.7 ***Develop a function*** *named* `number->image` *that takes in a number and returns an image of that number in (say) 18-point blue font.*

Hint: Combine the built-in functions `text` and `number->string`.

Exercise 9.2.8 ***Develop a function*** *named* `digits` *that takes in a positive integer (like 52073; you don't need to deal with fractions or decimals) and tells how many digits long it is, when written in base 10.*

Hint: This doesn't require any arithmetic, only combining functions described in this chapter.

9.3 Review of important words and concepts

Thus far we've seen three important *data types*, or kinds of information: *images*, *strings*, and *numbers* (which can be further broken down into *integers*, *fractions*, *floats*, and *complexes*). Racket provides several built-in functions for working on strings. These functions are used in exactly the same way as functions on images or functions on numbers. Likewise, you can define variables and functions with string values, just as you defined variables and functions with image or number values.

9.4 Reference: Built-in functions on strings

This chapter introduced the following built-in functions:

- `string-append`

- `string-length`
- `substring`
- `number->string`
- `string->number`