

## Characters and Strings in Scheme

**Summary:** In these exercises, you will explore a number of the standard Scheme procedures for handling characters and strings. You will also explore an application of these procedures for marking up text.

### Contents:

- Exercises
  - Exercise 0: Preparation
  - Exercise 1: Collating Sequences
  - Exercise 2: Character Predicates
  - Exercise 3: String Basics
  - Exercise 4: Creating Questions
  - Exercise 5: Referencing Lengths
  - Exercise 6: Building Simple Sentences
  - Exercise 7: Building Sentences, Revisited
  - Exercise 8: A Simple Form Letter
  - Exercise 9: Adding Quotation Marks
  - Exercise 10: More Form Letters
- For Those with Extra Time

### Useful Procedures and Notation:

- Constant notation: `#\ch` (character constants) "*string*" (string constants).
- Character constants: `#\a` (lowercase a) ... `#\z` (lowercase z); `#\A` (uppercase A) ... `#\Z` (uppercase Z); `#\0` (zero) ... `#\9` (nine); `#\space` (space); `#\newline` (newline); and `#\?` (question mark).
- Character conversion: `char->integer`, `integer->char`, `char-downcase`, and `char-upcase`
- Character predicates: `char?`, `char-alphabetic?`, `char-numeric?`, `char-lower-case?`, `char-upper-case?`, `char-whitespace?`, `char<?`, `char<=i?`, `char=?`, `char>=?`, `char>?`, `char-ci<?`, `char-ci<=i?`, `char-ci=?`, `char-ci>=?`, and `char-ci>?`.
- String predicates: `string?`
- String constructors: `make-string`, `string`, `string-append`
- String extractors: `string-ref`, `substring`
- String conversion: `list->string`, `number->string`, `string->list`
- String analysis: `string-length`,
- String comparison (links may not work): `string<?`, `string<=?`, `string=?`, `string>=?`, `string>?`, `string-ci<?`, `string-ci<=?`, `string-ci=?`, `string-ci>=?`, `string-ci>?`

## Exercises

### Exercise 0: Preparation

- a. If you have not done so already, you may also want to open separate tabs with the reading on characters and the reading on strings.
- b. If you have not done so already, you may want to skim Section 6.3.5 of the Scheme Report.
- c. Start DrScheme.

### Exercise 1: Collating Sequences

As you may recall, Scheme uses a *collating sequence* for the letters, assigning a sequence number to each letter. DrScheme uses the ASCII collating sequence.

- a. Determine the ASCII collating-sequence numbers for the capital letter A and for the lower-case letter a.
- b. Find out what ASCII character is in position 38 in the collating sequence.
- c. Do the digit characters *precede* or *follow* the capital letters in the ASCII collating sequence?
- d. If you were designing a character set, where in the collating sequence would you place the space character? Why?
- e. What position does the space character occupy in ASCII?

### Exercise 2: Character Predicates

- a. Determine whether our implementation of Scheme considers `#\newline` a whitespace character.
- b. Determine whether our implementation of Scheme indicates that capital B precedes lower-case a.
- c. Determine whether our implementation of Scheme indicates that lower-case a precedes capital B.
- d. Verify that the case-insensitive comparison operation (`char-ci<?`) gives the expected result for the previous two tests.
- e. Determine whether our implementation of Scheme indicates that `#\a` and `#\A` are the same letter. (It should not.)
- f. Find an equality predicate that returns `#t` when given `#\a` and `#\A` as parameters.

### Exercise 3: String Basics

- Write a Scheme expression to determine whether the symbol `'hyperbola` is a string.
- Write a Scheme expression to determine whether the character `#\A` is a string.
- Does the empty string (represented as `" "`) count as a string?

### Exercise 4: Creating Questions

Develop three ways of constructing the string `"???"` -- one using a call to `make-string`, one a call to `string`, and one a call to `list->string`.

### Exercise 5: Referencing Lengths

Here are two opposing views about the relationship between `string-length` and `string-ref`:

- “No matter what string `str` is, provided that it's not the empty string, `(string-ref str (string-length str))` will return the last character in the string.”
- “No matter what string `str` is, `(string-ref str (string-length str))` is an error.”

Which, if either, of these views is correct? Why?

### Exercise 6: Building Simple Sentences

Consider the definition

```
(define like
  (string-append
    "I like "
    person
    " because "
    person
    " is "
    adjective
    "."))
```

- What other values must be defined in order for this definition to work?
- What type must those values have?
- Suppose you had previously defined `person` as `"Daniel"` and `adjective` as `"cheerful"`. What do you expect the value of `like` to be?
- Confirm your previous answer experimentally.

## Exercise 7: Building Sentences, Revisited

One criticism of the `like` definition in the previous exercise is that it takes a lot of lines. We could define a similar sentence as follows:

```
(define tunes
  (string-append "I listen to " band " because their music is " adjective "."))
```

- What are the comparative advantages and disadvantages of the single-line sentence-building definition?
- Define `band` and `adjective` in such a way that `tunes` can be successfully defined.

## Exercise 8: A Simple Form Letter

We can, of course, use a similar technique to build longer form letters. Consider the following definitions

```
(define cr (string #\newline))
(define letter
  (string-append
    "Dear " recipient ", " cr
    cr
    "Thank you for your submission to " magazine ". Unfortunately, we " cr
    "consider the subject of your article, " article ", inappropriate for our" cr
    "readership. In fact, it is probably inappropriate for any readership." cr
    "Please do not contact us again, and do not bother other magazines with" cr
    "this inappropriate material or we will be forced to contact the " cr
    "appropriate authorities." cr
    cr
    "Regards," cr
    "Ed I. Tor" cr))
```

- What must be defined for the definition of `letter` to succeed?
- Confirm that the definition of `letter` works by using the following sub-definitions

```
(define recipient "Professor Schneider")
(define magazine "College Trustee News")
(define article "Using Grinnell's Endowment to Eliminate Tuition")
```

- You may note that the output is fairly ugly when you simply ask for `letter`. You can get nicer output by using the `display` procedure, as in `(display letter)`. Try doing so.

## Exercise 9: Adding Quotation Marks

- What changes are necessary to `letter` so that name of the article appears in quotation marks?
- Confirm your answer experimentally.

## Exercise 10: More Form Letters

a. Create a file, `sam.ss`, with the following lines:

```
(define recipient "Mr. Rebelsky")  
(define magazine "Liberal Arts Letters")  
(define article "Why Every Faculty Member Should Take Introductory Computer Science")
```

b. Create a separate file, `letter.ss` that contains the definition of `cr` and the updated definition of `letter` from the previous exercises.

c. In the definitions window, type the following

```
(load "sam.ss")  
(load "letter.ss")  
(display letter)
```

d. What do you expect to happen when you click Run?

e. Confirm your answer experimentally.

f. What ideas does this exercise suggest to you?

## For Those with Extra Time

Create another form of form letter.

---

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.