

6 Strings, printf, load, the *Run* Button, and Comments

This section introduces the following practicalities:

- Scheme's *string* data type. From a conceptual perspective, it would be nice to postpone our discussion of strings; however, from a practical perspective, we cannot do that. Strings are simply too useful for testing, debugging and so on.
- The built-in `printf` function. This function, which takes a *string* as one of its inputs, can be used to display information in DrScheme's Interactions Window. Its functionality is similar to that of the format/print operators found in many programming languages.
- The built-in `load` function. This function causes the Scheme expressions in a specified file to be evaluated in the Interactions Window. In this way, a library of useful Scheme definitions can be incorporated into your own program quite easily. The name of the file is specified by a *string*.
- The *Run* button. This button is located at the top-right of DrScheme's main window. When pressed, it causes the Scheme expressions in the Program Definitions Window to be evaluated, just as if they had been typed into a *fresh* Interactions Window.
- Comments. A comment is a piece of syntax that DrScheme completely ignores. Comments are used by programmers to help clarify (for people) what the program/code/function is supposed to do.

6.1 Strings

Syntactically, strings in Scheme are character sequences delimited by double-quotes. For example, `"hi"` and `"Howdy!"` are character sequences that represent string entities.

The following Interactions Window session demonstrates that the *evaluation* function behaves like the *identity* function when applied to string entities.

```
> "hi"
"hi"
> "Howdy!"
"Howdy!"
```

Scheme also includes a type-checker predicate for the string data type: `string?`, as demonstrated below.

```
> (string? "abc")
#t
> (string? ("a" "b" "c"))
#f
> (string? #t)
#f
```

6.2 The printf Function

Scheme includes a built-in `printf` function that operates on *format* strings as in many other languages. The following examples demonstrate the use of the `printf` function. The `printf` function interprets the character sequences, `\n`, `~%` and `~A`, in a special way that is discussed below. Such character sequences are commonly called *escape sequences*.

```
> (printf "Hi there!\n")
Hi there!
> (printf "Oh, I get it!~%")
Oh, I get it!
> (printf "First thing: ~A, second thing: ~A~%" (+ 2 3) (* 6 7))
First thing: 5, second thing: 42
```

The `printf` function causes the format string (i.e., its first argument) to be displayed in the Interactions Window, except that:

- the quotation marks are omitted;

- each instance of the *escape sequences*, `\n` or `~%`, is interpreted as a new-line character, and thus causes a *carriage return* in the Interactions Window; and
- each instance of the escape sequence, `~A`, is replaced by a character sequence representing the *value* of the corresponding input expression.

Notice that if the format string contains n instances of `~A`, then there must be n input expressions following the format string, as follows:

```
(printf format-string expr1 ... exprn).
```

Here are a few more examples:

```
> (printf "Line One!~% Line Two!!~% Line Three!!!~%")
Line One!
Line Two!!
Line Three!!!
> (printf "First ==> ~A, Second ==> ~A, Third ==> ~A~%"
      (+ 4 2) (- 9 6.3) (* 4 100))
First ==> 6, Second ==> 2.7, Third ==> 400
> (printf "A symbol: ~A, a string: ~A, a boolean: ~A~%"
      'I-am-a-symbol
      "I am a String!"
      (> 4 2))
A symbol: I-am-a-symbol, a string: I am a String!, a boolean: #t
```

Notice that expressions involving an application of the `printf` function evaluate to ... *nothing!* That's because the whole point of a `printf` function is its side effect. In reality, Scheme provides a special datum that is interpreted as "no value". This "no value" datum belongs to the *void* data type. In fact, the "no value" datum is the *only* datum belonging to the *void* data type. Like any other data type, there is a corresponding type-checker predicate for the *void* data type. It is called `void?`. Its use is demonstrated below.

```
> (void? (printf "hi\n"))
hi
#t
```

In this example, the Default Rule for evaluating non-empty lists is used to evaluate the expression, `(void? (printf "hi\n"))`. In the process, the `void?` symbol evaluates to the built-in `void?` function and `(printf "hi\n")` evaluates to the special "no value" datum belonging to the *void* data type. The "no value" datum is fed into the `void?` function, resulting in the output value `#t`, as reported by drScheme. The character sequence, `hi`, was printed out as a side-effect of the evaluation of the expression, `(printf "hi\n")`. DrScheme uses different colors to distinguish side-effect printing (e.g., `hi`) from output values (e.g., `#t`). Of course, in a black-and-white handout such as this, it's sort of hard to see the different colors!

Defining a useful tester function. The following example defines a `tester` function that can be used to demonstrate the evaluation of Scheme expressions. The `tester` function takes a Scheme entity as its input, prints out a character sequence representing that entity, and then evaluates that entity. The output value generated by the `tester` function is the result of evaluating the input entity.

```
> (define tester
      (lambda (expr)
        (printf "~A ==> " expr)
        (eval expr)))
> (tester '+)
+ ==> #<primitive:+>
> (tester +)
#<primitive:+> ==> #<primitive:+>
> (tester '(+ 1 2))
(+ 1 2) ==> 3
> (tester (+ 1 2))
3 ==> 3
```

Notice how the `quote` special form is used to shield the input expression from evaluation. Thus, the *unevaluated* Scheme entity is fed as input to the `tester` function. Then, when desired, inside the body of the tester function, the `eval` function is used to explicitly evaluate the test expression.

Question... How would you change the definition of the `tester` function so that it *printed out* the result of evaluating the Scheme entity instead of returning it as the output value? In this case, the `tester` function would return the “no value” datum.

6.3 The load Function and the *Run* Button

Scheme includes a built-in load function that causes all of the Scheme expressions in a specified file to be evaluated in an Interactions Window session. For example, suppose the file `"test.scm"` contains the following expressions:

```
(printf "Loading test.scm!!")

(define tester
  (lambda (expr)
    (printf "~A ===> ~A%" expr (eval expr))))

(define x 34)
```

Then the following Interactions Window session could ensue:

```
> x
BUG! reference to undefined identifier: x
> (load "test.scm")
Loading test.scm!!
> x
34
> (tester 'x)
x ===> 34
```

Thus, function definitions can be conveniently stored in a file, to be loaded whenever needed.

⇒ The *Run* button on DrScheme’s toolbar is similar to the load function, except that it causes the Scheme expressions currently residing in the Definitions Window to be evaluated within a fresh Interactions Window session.

6.4 Comments, Indentation, and Choosing Names for Variables, etc.

In Scheme, the semi-colon character is used to signal comments, as illustrated by the following example.

```
;;; TESTER
;;; =====
;;; Takes an arbitrary Scheme entity as its only input.
;;; Displays that entity in the Interactions Window.
;;; Then returns as its output, the result of evaluating
;;; that entity.
;;; -----
;;; NOTE: In typical usage, the desired input expression
;;; must be quoted to shield it from evaluation.

(define tester
  (lambda (expr)
    (printf "~A ==> " expr) ;; Notice, no newline character!
    (eval expr) ;; A rare example of explicitly calling
                ;; the EVAL function!
  ))
```

```
;;; Examples
;;; -----

(tester '(+ 2 3))
(tester (+ 2 3))
```

Evaluating the above code in the Interactions Window would have the same result as evaluating the following, uncommented code:

```
(define tester
  (lambda (expr)
    (printf "~A ==> " expr)
    (eval expr)))

(tester '(+ 2 3))
(tester (+ 2 3))
```

The comments make the code easier for people to understand. DrScheme ignores the comments completely.