

Input for Interactive Programs

Summary: We consider mechanisms for interactive (and non-interactive) programs to get input in ways other than as simple parameters.

Contents:

- Introduction
- `read`
- An Example
- Sentinels

Introduction

In the programs we've written so far this semester, we've assumed that all the data that a program needs can be either included in the source code, generated automatically within the program, or (at worst) supplied in the interactions window as an argument in a call to one of the program's procedures.

Unfortunately, this simplifying assumption doesn't always hold. In many cases, we'd like our program to take over the job of interacting with users, reading in values and displaying results. To support programs of this kind, Scheme provides several primitive procedures that perform interactive input or output as "side effects". You've already seen the three primary output procedures, `display`, `newline`, and `write`. There's one key input procedure, `read`.

`read`

The `read` procedure takes no arguments and returns one value. When it is invoked, it pauses and waits for the user to supply a representation of a Scheme value -- a numeral, a string literal (enclosed in double-quotation marks, as if in a Scheme program), a Boolean or character literal, a symbol (which need not be preceded by a single quotation mark), or a list (which again need not be quoted). The `read` procedure returns the value represented.

Under DrScheme, the `read` procedure's interaction with the user takes place in an *interaction box*, visually separated from the rest of the Interactions window. The user of the program types into this box a text representation of the value that she wants to send to the program -- the number 25, say:

25

When the user presses the <Enter> key to end the line, DrScheme releases the value that she has entered to the `read` procedure, which returns it.

An Example

Here's a small illustration of the use of the `read` procedure. The `square-root-computer` procedure asks the user to supply a number, computes the square root of the number that the user supplies, and prints out the result, appropriately labelled, all within the interaction box:

```
;;; Procedure:
;;;   square-root-computer
;;; Parameters:
;;;   [None, the input is read interactively]
;;; Purpose:
;;;   Reads in a number and displays its square.
;;; Produces:
;;;   [Nothing]
;;; Preconditions:
;;;   [None]
;;; Postconditions:
;;;   The program's user has been prompted for a number.
;;;   The program's user's reply has been read in.
;;;   If the program's user's reply is a number, its square
;;;   root has been printed out, appropriately labelled.
(define square-root-computer
  (lambda ()
    (display "Give me a number, and I'll compute its square root.")
    (newline)
    (let ((proposed-number (begin
                            (display "Number: ")
                            (read))))
      (begin
        (display "The square root of ")
        (display proposed-number)
        (display " is ")
        (display (sqrt proposed-number))
        (display ".")
        (newline))))))
```

The following sample calls demonstrate the working of the `square-root-computer` procedure. Notice that the value of `proposed-number` is not supplied as an argument to `square-root-computer`, but is read in as the program is being executed. The green printing shows where the user typed it in.

```
> (square-root-computer)
Give me a number, and I'll compute its square root.
Number: 4225
The square root of 4225 is 65.
```

Sentinels

If one wants the procedure to compute many square roots instead of just one, prompting the user each time for a new number, one can set up a recursion in which the completion of each exchange initiates another:

```

;;; Procedure:
;;; multi-square-root-computer
;;; Purpose:
;;; prompts the user for numbers and
;;; and outputs the square root of each one
;;; Parameters:
;;; [None]
;;; Produces:
;;; [Nothing; Called for its side effects]
;;; Preconditions:
;;; [None]
;;; Postconditions:
;;; The program user has been prompted at least once for a number.
;;; The program user's reply has been read in.
;;; If the program user's reply is a number, its square root has
;;; been printed out, appropriately labelled, and the prompt
;;; has been repeated.
;;; If the program user's reply is the symbol STOP, a
;;; cheerful salutation of farewell has been printed out
;;; and the prompt has not been repeated.
(define multi-square-root-computer
  (lambda ()
    (display "Give me one number at a time.")
    (newline)
    (display "I'll compute its square root and ask you for another number.")
    (newline)
    (display "Type STOP when you're done.")
    (newline)
    (let kernel ((proposed-number (begin
                                   (display "Number: ")
                                   (read))))
      (cond ((eq? proposed-number 'stop)
             (begin
              (display "Goodbye!")
              (newline)))
            ((number? proposed-number)
             (begin
              (display "The square root of ")
              (display proposed-number)
              (display " is ")
              (display (sqrt proposed-number))
              (display ".")
              (newline)
              (kernel (begin
                       (display "Number: ")
                       (read))))))
            (else
             (error 'multi-square-root-computer
                    "The input must be a number."))))))

```

Let's walk through the body of this procedure definition. When `multi-square-root-computer` is invoked, it begins by printing out three lines of instructions, then enters the recursive kernel, reading in the first user input as it enters and associating the parameter `proposed-number` with it.

The `cond`-expression first checks to see whether the user has submitted the symbol `stop`, which it interprets as a *sentinel* -- a conventional signal of the end of the input, indicating that the user is ready to leave the program. If the sentinel is detected, `multi-square-root-computer` prints out “Goodbye!” and returns.

If the user’s input is not `stop`, however, the second `cond`-clause is activated. If the user has submitted a number, `multi-square-root-computer` figures its square root and displays the result, embedded in a complete English sentence.

On the other hand, if the user’s input is neither the symbol `stop` nor a number, it is erroneous, and the procedure signals that a precondition has failed by invoking the `error` procedure to halt execution.

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.