A Scheme Word Count Program

A tutorial introduction to literate programming in Scheme

Raja Sooriamurthi

The Unix `wc` program has been used a didactic example to illustrate literate programming [2, 3]. In continuing that tradition we present below a Scheme version of the `wc` program written using `noweb`.

The calling convention of the program is: `(wc 'l 'c 'w "file-1" "file-2" ... )` . Where `c`, `w`, and `l` are counter parts of the Unix `wc` options specifying whether we want a count of the number of characters (bytes), words (white space separated character sequences), or lines (newline characters). If no options are explicitly specified then it is assumed to be `(c l w)`. Files are identified by strings following the options. If no files are specified then input is read from the `(current-input-port)`.

Output is returned as a list of the various counts. For instance

```
> (wc "/etc/motd")
(("/etc/motd" (chars 92) (words 13) (lines 4)))

> (wc 'l 'c 'w "/etc/motd" "/etc/passwd")
(("/etc/motd" (chars 92) (words 13) (lines 4))
 ("/etc/passwd" (chars 1629) (words 59) (lines 28))
 (total (chars 1721) (words 72) (lines 32)))

> (wc)
> (wc)
Literate
Programming
can be
fun
(<stdio> (chars 33) (words 5) (lines 5))
```

1 The Idea

The `wc` is a simple *Mealy machine* as given below:

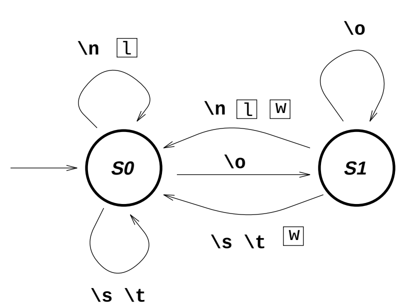

Figure 1: A Mealy machine that represents the behaviour of the `wc` program.

The arc transitions occur on a newline (`\n`), a space or tab (`\s, \t`) or any other character (`\o`). Associated with each arc transition we have an associated action. All of these actions increment a counter.

$\boxed{l}$ and $\boxed{w}$ are counters for the number of lines and words in the input. There is another counter for characters $\boxed{c}$ whose increment is associated with all the arcs and hence has not been explicitly shown.

2 The Program

Given the specifications above we have the outline of the program as:

2a ⟨* 2a⟩≡

```
⟨Some help routines 4c⟩
⟨Scheme wc 2b⟩
⟨Process an input 3b⟩
```

Root chunk (not used in this document).

The main `wc` program consists of two phases (1) processing the input option argument to determine the format of the output [1] and (2) processing the individual files. (In this code we make use of Chez Scheme's multiple values [1].)

2b ⟨*Scheme wc* 2b⟩≡

```
(define wc
  (lambda args
    (call-with-values
     ⟨Split input args 4b⟩
     ⟨Process all of the input files 2c⟩)))
```

Defines:
`wc`, never used.
This code is used in chunk 2a.

To process the input we have to check if no files have been specified and if so we receive input from the `(current-input-port)` or else we process the specified files.

2c ⟨*Process all of the input files* 2c⟩≡

```
(lambda (options all-files)
  (if (null? all-files)
      ;; then process the (current-input-port)
      (call-with-values
       (lambda () (process-input (current-input-port)))
       (lambda (chars words lines)
         `(<stdio> (chars ,chars) (words ,words) (lines ,lines))))
      ;; else process the list of the input files
      ;; keeping track of a running total
      (let loop
          ([files all-files]
           [total_chars 0]
           [total_words 0]
           [total_lines 0])
        (if (null? files)
            (if (null? (cdr all-files))
                ;; there was only 1 file
                '()
                ;; else return a summary
                (list `(Total (chars ,total_chars) (words ,total_words)
                              (lines ,total_lines))))
            ⟨Process a single file 3a⟩))))
```

Defines:
`total_chars`, used in chunk 3a.
`total_lines`, used in chunk 3a.

[1] This program isn't complete: it only processes the full set of options and not a particular subset.

`total_words`, used in chunk 3a.
Uses `chars` 3b, `lines` 3b, `process-input` 3b, and `words` 3b.
This code is used in chunk 2b.

To process a single file we first open it, count the number of characters, words and lines, and then update the running total.

3a ⟨*Process a single file* 3a⟩≡

```
(call-with-input-file (first files)
  (lambda (port)
    (call-with-values
     (lambda () (process-input port))
     (lambda (chars words lines)
       (cons `(,(first files) (chars ,chars) (words ,words) (lines ,lines))
             (loop (rest files)
                   (+ total_chars chars)
                   (+ total_words words)
                   (+ total_lines lines)))))))
```

Uses `chars` 3b, `first` 4c, `lines` 3b, `process-input` 3b, `rest` 4c, `total_chars` 2c, `total_lines` 2c, `total_words` 2c, and `words` 3b.
This code is used in chunk 2c.

To process an input we just traverse the aforementioned two state Mealy machine.

3b ⟨*Process an input* 3b⟩≡

```
(define process-input
  (lambda (port)
    (let loop
       ([chars 0]
        [words 0]
        [lines 0]
        [state 's0]
        [in (read-char port)])
      (if (eof-object? in)
          (values chars words lines)
          (case state
            ⟨State 0 3c⟩
            ⟨State 1 4a⟩)))))
```

Defines:
`chars`, used in chunks 2–4.
`in`, used in chunks 3c and 4a.
`lines`, used in chunks 2–4.
`process-input`, used in chunks 2c and 3a.
`state`, never used.
`words`, used in chunks 2–4.
This code is used in chunk 2a.

The first state of the FSA is defined by:

3c ⟨*State 0* 3c⟩≡

```
[(s0)
 (case in
   [(#\newline)
    (loop (add1 chars) words (add1 lines) 's0 (read-char port))]
   [(#\space #\tab)
    (loop (add1 chars) words lines 's0 (read-char port))]
   [else (loop (add1 chars) words lines 's1 (read-char port))])]
```

Uses `chars` 3b, `in` 3b, `lines` 3b, and `words` 3b.
This code is used in chunk 3b.

The second state of the FSA is:

4a ⟨*State 1* 4a⟩≡
```
[(s1)
 (case in
   [(#\newline)
    (loop (add1 chars) (add1 words) (add1 lines) 's0 (read-char port))]
   [(#\space #\tab)
    (loop (add1 chars) (add1 words) lines 's0 (read-char port))]
   [else (loop (add1 chars) words lines 's1 (read-char port))])]
```
Uses `chars` 3b, `in` 3b, `lines` 3b, and `words` 3b.
This code is used in chunk 3b.

To determine what are the options to be used while producing the final output we split the input `args` into a list of options and a list of the file names.

4b ⟨*Split input args* 4b⟩≡
```
(lambda ()
  (let loop
      ([args args]
       [options '()])
      (cond
        [(null? args) (values options args)]
        [(string? (first args))
         ;; the options have been processed
         (values options args)]
        [else (loop (rest args) (cons (first args) options))])))
```
Uses `first` 4c and `rest` 4c.
This code is used in chunk 2b.

Finally some simple help routines.

4c ⟨*Some help routines* 4c⟩≡
```
(define first car)
(define rest cdr)
```
Defines:
`first`, used in chunks 3a and 4b.
`rest`, used in chunks 3a and 4b.
This code is used in chunk 2a.

Chunks:

Index:

References

[1] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 140–149, 1994.

[2] Donald E. Knuth. *Literate Programming*. CSLI, Stanford University, Stanford, CA, 1992.

[3] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, Sept 1994.