changed-stream

Warren Wilkinson

December 17, 2012

Contents

1 Overview **2**
- 1.1 Features . . . 2
- 1.2 Limitations . . . 3

2 Installation **3**
- 2.1 Quick Lisp . . . 3
- 2.2 Gentoo . . . 3
- 2.3 Ubunto . . . 4
- 2.4 Manual Installation . . . 4
- 2.5 Running the Tests . . . 4
- 2.6 Getting Support . . . 5

3 Implementation **5**
- 3.1 diffcase . . . 5
  - 3.1.1 changed-stream class . . . 6
- 3.2 file-position . . . 7
- 3.3 read-char . . . 8
- 3.4 peek-char . . . 8
- 3.5 read-sequence . . . 9
- 3.6 Test Framework . . . 10
  - 3.6.1 Read Patterns . . . 10
  - 3.6.2 Running Tests . . . 14

4 Tests **15**
- 4.1 Deletion Tests . . . 16
- 4.2 Insertion Tests . . . 18
- 4.3 Replacement Tests . . . 18
- 4.4 Replace+Delete Tests . . . 20

4.5 Replace+Insert Tests 20

5 License **21**

1 Overview

Changed-stream lets you apply changes to a stream, without modifying the underlying stream. In this example, stream s remains unmodified except for file-position[1].

```
(with-input-from-string (s "0123456789")
  (let ((d (change-stream s
             :at 3
             :delete 1
             :insert "xyz")))
    (read-line d)))
;; "012xyz456789"
```

. . . and changed-streams are composable, but be aware of the order of operations.

```
(with-input-from-string
  (s "The Cat in the Hat")
  (read-line (change-stream
              (change-stream s
                 :at 2
                 :delete 6)
              :at 12
              :insert "chery")))
;; "Thin the Hatchery"
```

```
(with-input-from-string
  (s "The Cat in the Hat")
  (read-line (change-stream
              (change-stream s
                 :at 12
                 :insert "chery")
              :at 2
              :delete 6)))
;; "Thin tcheryhe Hat"
```

1.1 Features

- read-char
- peek-char

[1] Changed-streams, like regular streams, aren't implicitly thread safe. But also ensure the underlying stream isn't being read concurrently.

- file-position
- read-sequence has been efficiently implemented.
- extending and shrinking the stream (deleting or inserting near the end will shrink and grow the stream).

1.2 Limitations

- The underlying stream must be at file-position 0 when the changed-stream is created. There is an assert to catch this.
- Thread safety is your responsibility – in particular don't change the file-position of the underlying stream, the changed-stream will give incorrect results and could cause errors.
- unread-char has not been implemented (but peek-char has been)
- Read only. There are no writing operations allowed to the stream

2 Installation

2.1 Quick Lisp

Install Quick Lisp and then run:

```
(ql:quickload 'changed-stream)
```

If you have problems, see the support section, and you may want to run the tests.

2.2 Gentoo

As root,

```
emerge changed-stream
```

Once the emerge is finished, the package can be loaded using ASDF:

```
(asdf:operate 'asdf:load-op :changed-stream)
```

If you have problems, see the support section, otherwise you may want to run the tests.

2.3 Ubunto

```
sudo apt-get install changed-stream
```

Once the installation is finished, the package is loadable using ASDF:

```
(asdf:operate 'asdf:load-op :changed-stream)
```

If you have problems, see the support section, otherwise you may want to run the tests.

2.4 Manual Installation

In summary: Untar the .tar package and then symlink the .asd files into a place where ASDF can find them.

1. Untar the files where you want them to be. On windows download the .zip and unzip it instead, it's the same files.

2. ASDF could be looking anywhere – it depends on your setup. Run this in your lisp repl to get a clue as to where ASDF is seeking libraries[2]:

   ```
   (mapcan #'funcall asdf:*default-source-registries*)
   ```

3. Symlink the .asd files to the source directory. If you use windows, these instructions on symlink alternatives apply to you.

Once the files are in place, the package can be loaded with ASDF by:

```
(asdf:operate 'asdf:load-op :changed-stream)
```

If you have problems, see the support section. If you don't have problems you may want to run the tests anyway, because you can.

2.5 Running the Tests

Once the system is loaded, it can be tested with asdf.

```
(asdf:operate 'asdf:test-op :changed-stream)
```

[2] you might need to (require 'asdf) before running this example

This should display something like the following. There should be **zero failures**, if you have failures see the support section of this document.

```
RUNNING CHANGED-STREAM TESTS...
CHANGED-STREAM TEST RESULTS:
     Tests: 124
   Success: 124
  Failures: 0
```

2.6 Getting Support

You can find support on this libraries website and/or github repository. Or you can email Warren Wilkinson.

3 Implementation

A changed stream has five zones:

1. **Before** The area before our changes.
2. **Replace** The range were our changes overwrite the source.
3. **Delete** The range where we are deleting (not emitting) source characters.
4. **Insert** The range where we are outputing new characters, but not consuming source characters.
5. **After** The area after our changes.

We never have both delete and insert, instead we'd represent that as a replace and insert or a replace and delete.

3.1 diffcase

The changed-stream class is designed to support a macro called 'diffcase'. This macro is like a case statement, but supports these five parts. Most of the stream functions use this macro.

```
(defmacro diffcase ((position stream)
                    (before &rest before-case-code)
                    (replace &rest replace-case-code)
                    (delete &rest delete-case-code)
```

```
                  (insert &rest insert-case-code)
                  (after &rest after-case-code))
  (declare (ignore before replace delete insert after))
  (let ((pos (gensym)) (strm (gensym)))
    `(let ((,pos ,position) (,strm ,stream))
       (cond ((< ,pos (last-unchanged-position ,strm)) ,@before-case-code)
             ((< ,pos (last-replacement-position ,strm)) ,@replace-case-code)
             ((< ,pos (last-modified-position ,strm))
              (if (mod-is-delete-p ,strm)
                  (progn ,@delete-case-code)
                  (progn ,@insert-case-code)))
             (t ,@after-case-code)))))
```

3.1.1 changed-stream class

The class support the macros, and the macro cares about character positions: At what position is the first change? At what position is the last replacement? And where is the last modification!

The class stores these precomputed values so we don't have to add our change-position to the minimum of the length of the insert string vs the number of deleted characters – we know this number, it's 'last-replacement-position'.

```
(defclass changed-stream (fundamental-character-input-stream)
  ((stream                    :initarg  :stream
                              :reader stream-of)
   (virtual-position          :initform 0
                              :accessor virtual-position)
   (last-unchanged-position   :initarg  :last-unchanged-position
                              :reader last-unchanged-position)
   (last-replacement-position :initarg  :last-replacement-position
                              :reader last-replacement-position)
   (last-modified-position    :initarg  :last-modified-position
                              :reader last-modified-position)
   (removed-characters        :initarg  :removed-characters
                              :reader removed-characters)
   (insert-string             :initarg  :string
                              :reader insert-string)))

(defun mod-is-delete-p (stream) (> (removed-characters stream) 0))
```

```
(defun change-stream (stream &key (at 0) (insert "") (delete 0))
  (assert (zerop (file-position stream)))
  (let ((last-replace (+ at (min (length insert) delete)))
        (last-mod (+ at (max delete (length insert)))))
    (make-instance 'changed-stream
       :stream stream
       :last-unchanged-position at
       :last-replacement-position last-replace
       :last-modified-position last-mod
       :removed-characters (- delete (length insert))
       :string insert)))
```

3.2 file-position

When we set our own file-position, we have to compute the correct file-position to set the source stream too. This is easy during the before and replacement stages – it's our current position.

If we are repositioning into the delete range, the file-position of the source moved ahead by the number of characters to delete.

If we are repositioning into the insert range, the file position of the source is just after the position of the last replaced character. When we start reading the source again, all the replaced characters will have been skipped.

If we are repositioning after the modification range, the file-position of the source moved ahead by the number of characters to delete. This takes into account the fact that these characters have been skipped and we are further in our source than we'd be otherwise. If our modification was insert+replace, then removed-characters will be negative and will put the file-position backwards. Since extra characters have been emitted, we're further back in the source stream.

```
(defmethod stream-file-position ((stream changed-stream) &optional newval)
  (if newval
      (progn
        (diffcase (newval stream)
           (before  (file-position (stream-of stream) newval))
           (replace (file-position (stream-of stream) newval))
           (delete  (file-position (stream-of stream)
                                   (+ newval (removed-characters stream))))
           (insert  (file-position (stream-of stream)
```

```
                                    (last-replacement-position stream)))
          (after   (file-position (stream-of stream)
                                    (+ newval (removed-characters stream)))))
        (setf (virtual-position stream) newval))
      (virtual-position stream)))
```

3.3 read-char

Real characters are characters from the source stream. Replacement characters skip a source stream character and return the next character from the insert stream (a virtual character).

For deletes, we seek the end of the deleted characters and then emit the next.

```
(defmethod stream-read-char ((stream changed-stream))
  (diffcase ((if (mod-is-delete-p stream)
                 (file-position (stream-of stream))
                 (virtual-position stream))
             stream)
    (before  (real-char stream))
    (replace (replacement-char stream))
    (delete  (internal-seek-end-of-delete stream) (real-char stream))
    (insert (virtual-char stream))
    (after  (real-char stream))))
```

3.4 peek-char

Similar in structure read-char, peek-char does the same operation but with peeks. The replace region just takes the next virtual-char (rather than the replacement-char which would read and drop the next character from the source stream).

```
(defmethod stream-peek-char ((stream changed-stream))
    (diffcase ((if (mod-is-delete-p stream)
                   (file-position (stream-of stream))
                   (virtual-position stream)) stream)
      (before  (peek-char nil (stream-of stream) nil :eof))
      (replace (virtual-char stream t))
      (delete  (internal-seek-end-of-delete stream)
               (peek-char nil (stream-of stream) nil :eof))
      (insert  (virtual-char stream t))
```

```
    (after   (peek-char nil (stream-of stream) nil :eof))))
```

3.5 read-sequence

Read sequence is three functions, implemented as one for simplicity. But essentially:

1. Read the sequence before the changes by calling read-sequence on the source stream.
2. Read the sequence during the changes by call-next-method (which will do it by calls to read-char)
3. Read the sequence after the changes by calling read-sequence on the source stream.

The complexity of this function comes from computing the correct start and end ranges for these three steps.

```
(defmethod stream-read-sequence ((stream changed-stream) seq &optional start end)
  (let ((position (virtual-position stream)))
    (let ((write-length (- (or end (length seq)) (or start 0))))
      (let ((before-length (min write-length
                            (max 0 (- (last-unchanged-position stream) position)))))
        (read-sequence seq (stream-of stream)
                       :start start
                       :end (+ start before-length))
        (incf start before-length)
        (incf position before-length)
        (decf write-length before-length)
        (file-position stream position))

      (let ((during-length (min write-length
                            (max 0 (- (last-modified-position stream) position)))))
        (call-next-method stream seq start (+ start during-length))
        (incf start during-length)
        (incf position during-length)
        (decf write-length during-length))

      (let ((after-length write-length))
        (read-sequence seq (stream-of stream)
```

```
                    :start start
                    :end (+ start after-length))
        (incf start after-length)
        (incf position after-length)
        (decf write-length after-length)
        (file-position stream position)))))
```

3.6 Test Framework

Tests are just functions, that are pushed onto a list when defined. Within do-testing are functions that read and verify the changed-stream using some pattern of reads, peeks, file-positions and read-sequences.

```
(defvar *all-tests* nil)

(defun do-testing (at delete insert is)
  (with-input-from-string (input "0123456789ABCDEF")
    (let ((diff (change-stream input :at at :insert insert :delete delete)))
      (and (pattern-sequential diff is)
           (pattern-sequential+peek diff is)
           (pattern-file-position-forward diff is)
           (pattern-file-position-backward diff is)
           (pattern-file-position-backward-with-reads diff is)
           (pattern-read-sequences diff is)))))

(defmacro deftest (name &key (at 0) (delete 0) (insert "") is)
  `(progn
     (defun ,name () (do-testing ,at ,delete ,insert ,is))
     (pushnew ',name *all-tests*)))
```

3.6.1 Read Patterns

All the various pattern reading tests use defpattern to wrap a few common variables. It's basically a fixture.

- Output the pattern readers name.
- Set file position to 0
- Create a local variable called 'success'
- Create a local variable called 'storage' that can be read into.

Meanwhile, the macro problem combines outputting an error message, and setting success to false into one operation.

```
(defmacro defpattern (name (diff expected) &rest code)
  `(defun ,name (,diff ,expected)
     (format t ,(concatenate 'string "~% " (string-downcase (symbol-name name))))
     (file-position ,diff 0)
     (let ((success t)
           (storage (make-string (length ,expected) :initial-element #\_)))
       ,@code (format t "~%") success)))

(defmacro problem (msg &rest args)
  `(progn
     (setf success nil)
     (format t ,(concatenate 'string "~%    * " msg)
             ,@args)))
```

- pattern-sequential
  Uses read-char to read the entire thing. Checks to ensure that you are not allowed to read more characters than should actually exist (called 'overreading' here).

  ```
  (defpattern pattern-sequential (s expected)
    (dotimes (i (length expected))
      (unless (eq i (file-position s))
        (problem "Bad file-position, got ~a instead of ~d"
                 (file-position s) i))
      (let ((char (read-char s nil :eof)))
        (if (eq char :eof)
            (problem "EOF at ~d" i)
            (setf (char storage i) char))))
    (let ((char (read-char s nil :eof)))
      (unless (eq char :eof) (problem "Allowed overread of ~a" char)))
    (unless (string= storage expected)
      (problem "Wrong result (got ~s instead of ~s)" storage expected)))
  ```

- pattern-sequential+peek
  Like pattern-sequential, but peeks characters before reading to ensure that peeking doesn't mess things up.

```
(defpattern pattern-sequential+peek (s expected)
  (dotimes (i (length expected))
    (unless (eq i (file-position s))
      (problem "Bad file-position, got ~a instead of ~d"
               (file-position s) i))
    (let* ((peek (peek-char nil s nil :eof))
           (char (read-char s nil :eof)))
      (unless (eq peek char)
        (problem "Peek return ~a, char return ~a at ~a" peek char i))
      (if (eq char :eof)
          (problem "Peek caused EOF character at ~d" i)
          (setf (char storage i) char))))
  (let ((peek (peek-char nil s nil :eof))
        (char (read-char s nil :eof)))
    (unless (eq peek char)
      (problem "Peek overread ~a and char overread ~a." peek char))
    (unless (eq char :eof)
      (problem "Peek allowed overread of ~a" char))
    (unless (string= storage expected)
      (problem "Wrong result (got ~s instead of ~s)"
               storage expected))))
```

- pattern-file-position-forward
  Like pattern-sequential, but does a file-position before each read. This test is to ensure that file-positions, followed by reads, always produce the correct result.

```
(defpattern pattern-file-position-forward (s expected)
  (dotimes (i (length expected))
    (file-position s i)
    (let ((char (read-char s nil :eof)))
      (if (eq char :eof)
          (problem "~%EOF at ~2d" i)
          (setf (char storage i) char))))

  (unless (string= storage expected)
    (problem "Wrong result (got ~s instead of ~s)"
             storage expected)))
```

- pattern-file-position-backward
  Like pattern-file-position-forward, but in reverse. We read the last character, then the last-1 character... This test is because pattern-file-position-forward isn't difficult enough to be a good test of the file-position operation on it's own.

```
(defpattern pattern-file-position-backward (s expected)
  (loop for i from (1- (length expected)) downto 0
        do (progn
             (file-position s i)
             (let ((char (read-char s nil :eof)))
               (if (eq char :eof)
                   (problem "EOF at ~2d" i)
                   (setf (char storage i) char)))))
  (unless (string= storage expected)
    (problem "Wrong result (got ~s instead of ~s)"
             storage expected)))
```

- pattern-file-position-backward-with-reads
  Like pattern-file-position-backward, but each file-position is followed by reading up to the end of the string again. This test is to ensure that file-position produces correct results for **all** subsequent read-chars – not just the next one.

```
(defpattern pattern-file-position-backward-with-reads (s expected)
  (loop for i from (1- (length expected)) downto 0
        do (fill storage #\_)
        do (file-position s i)
        do (loop for j from i upto (1- (length expected))
                 as char = (read-char s nil :eof)
                 if (eq char :eof)
                 do (problem "EOF reading from ~2d at ~2d: ~s"
                             i j storage)
                 do (setf (char storage j) char))
        if (not (string= (subseq storage i) (subseq expected i)))
        do (problem "Wrong result (reading from ~2d got ~s instead of ~s)"
                    i storage expected)))
```

- pattern-read-sequences
  Uses read-sequence on every valid combination of (start, end). Ensures

every read-sequence no matter where it starts or where it ends produces the correct result.

```
(defpattern pattern-read-sequences (s expected)
  (loop for start from 0 upto (1- (length expected))
        do (loop for end from start upto (1- (length expected))
                 do (file-position s start)
                 do (fill storage #\_)
                 do (read-sequence storage s :start start :end end)
                 if (not (string= (subseq storage start end)
                                  (subseq expected start end)))
                 do (problem "Sequence (~2d, ~2d) got ~a instead of ~s"
                             start end storage expected))))
```

3.6.2 Running Tests

The bulk of the test code just has to do with collecting results and making pretty output.

```
(defstruct results
  (tests 0)
  (failures nil))
(defun results-failure-count (results)
  (length (results-failures results)))
(defun results-successes (results)
  (- (results-tests results)
     (results-failure-count results)))

(defun runtest (fun results)
  (let* ((success t)
         (output (with-output-to-string (*standard-output*)
                   (setf success (funcall fun)))))
    (make-results
     :tests (1+ (results-tests results))
     :failures (if success
                   (results-failures results)
                   (acons fun output (results-failures results))))))

(defun present-failures (results)
  (format t "~%CHANGED-STREAM FAILURES:~%")
```

```
  (loop for (fn . problems) in (results-failures results)
        do (format t "~%~a~a~%" fn problems)))
(defun present-results (results)
  (format t "~%CHANGED-STREAM TEST RESULTS:")
  (format t "~%    Tests: ~a~%   Success: ~a~%  Failures: ~a"
          (results-tests results)
          (results-successes results)
          (results-failure-count results))
  (when (results-failures results)
    (present-failures results)))

(defun run-tests ()
  (format t "~%RUNNING CHANGED-STREAM TESTS...")
  (present-results
   (reduce #'(lambda (test results) (runtest test results))
           *all-tests* :from-end t :initial-value (make-results))))
```

4 Tests

This package is tested by changing the string '01234567890ABCDEF' in a known way and then verifying that the results are correct no matter how it's read.

we test these reading patterns:

Scanning read-char the entire stream.

Peeking read-char the entire stream, but peek-char before reading.

Seeking file-position to the next character then read it, then next char, then ...

Backwards Seeking File-position to the last character, then read it, then next-to-last, then ...

Backwards Seeking with Forward Scanning File-position to the last character, then scan stream, then next-to-last, then ...

With read-sequence For i from 0 to end, and j from i to end, use file-position and read-sequence to read the range i to j.

4.1 Deletion Tests

at	delete	is	Comments
0	0	0123456789ABCDEF	No changes returns original stream.
0	1	123456789ABCDEF	
0	2	23456789ABCDEF	
0	3	3456789ABCDEF	
0	8	89ABCDEF	
0	9	9ABCDEF	
0	15	F	
0	16		
0	17		
0	18		
1	1	023456789ABCDEF	
1	2	03456789ABCDEF	
1	3	0456789ABCDEF	
1	7	089ABCDEF	
1	13	0EF	
1	14	0F	
1	15	0	
1	16	0	
1	17	0	
2	1	013456789ABCDEF	
2	2	01456789ABCDEF	
2	7	019ABCDEF	
2	13	01F	
2	14	01	
2	15	01	
8	1	012345679ABCDEF	
8	2	01234567ABCDEF	
8	7	01234567F	
8	8	01234567	
8	9	01234567	
14	1	0123456789ABCDF	
14	2	0123456789ABCD	
14	3	0123456789ABCD	
15	1	0123456789ABCDE	
15	2	0123456789ABCDE	
16	1	0123456789ABCDEF	
16	2	0123456789ABCDEF	
17	1	0123456789ABCDEF	

4.2 Insertion Tests

at	insert	is	Comments
0	x	x0123456789ABCDEF	
0	xy	xy0123456789ABCDEF	
0	)!@#$%^&*(abcdef	)!@#$%^&*(abcdef0123456789ABCDEF	
0	)!@#$%^&*(abcdef+	)!@#$%^&*(abcdef+0123456789ABCDEF	
0	)!@#$%^&*(abcdef+-	)!@#$%^&*(abcdef+-0123456789ABCDEF	
1	x	0x123456789ABCDEF	
1	xy	0xy123456789ABCDEF	
1	)!@#$%^&*(abcdef	0)!@#$%^&*(abcdef123456789ABCDEF	
1	)!@#$%^&*(abcdef+	0)!@#$%^&*(abcdef+123456789ABCDEF	
2	x	01x23456789ABCDEF	
2	xy	01xy23456789ABCDEF	
2	)!@#$%^&*(abcdef	01)!@#$%^&*(abcdef23456789ABCDEF	
2	)!@#$%^&*(abcdef+	01)!@#$%^&*(abcdef+23456789ABCDEF	
8	x	01234567x89ABCDEF	
8	xy	01234567xy89ABCDEF	
8	)!@#$%^&*(abcdef	01234567)!@#$%^&*(abcdef89ABCDEF	
8	)!@#$%^&*(abcdef+	01234567)!@#$%^&*(abcdef+89ABCDEF	
14	x	0123456789ABCDxEF	
14	xy	0123456789ABCDxyEF	
14	)!@#$%^&*(abcdef	0123456789ABCD)!@#$%^&*(abcdefEF	
14	)!@#$%^&*(abcdef+	0123456789ABCD)!@#$%^&*(abcdef+EF	
15	x	0123456789ABCDExF	
15	xy	0123456789ABCDExyF	
15	)!@#$%^&*(abcdef	0123456789ABCDE)!@#$%^&*(abcdefF	
15	)!@#$%^&*(abcdef+	0123456789ABCDE)!@#$%^&*(abcdef+F	
16	x	0123456789ABCDEFx	
16	xy	0123456789ABCDEFxy	
16	)!@#$%^&*(abcdef	0123456789ABCDEF)!@#$%^&*(abcdef	
16	)!@#$%^&*(abcdef+	0123456789ABCDEF)!@#$%^&*(abcdef+	
17	x	0123456789ABCDEF	
18	x	0123456789ABCDEF	
20	x	0123456789ABCDEF	

4.3 Replacement Tests

at	delete	insert	is	Comments
0	1	)	)123456789ABCDEF	
0	2	)!	)!23456789ABCDEF	
0	8	)!@#$%^&	)!@#$%^&89ABCDEF	
0	14	)!@#$%^&*(abcd	)!@#$%^&*(abcdEF	
0	15	)!@#$%^&*(abcde	)!@#$%^&*(abcdeF	
0	16	)!@#$%^&*(abcdef	)!@#$%^&*(abcdef	
0	17	)!@#$%^&*(abcdefg	)!@#$%^&*(abcdefg	
0	18	)!@#$%^&*(abcdefgh	)!@#$%^&*(abcdefgh	
1	1	!	0!23456789ABCDEF	
1	8	!@#$%^&*	0!@#$%^&*9ABCDEF	
1	14	!@#$%^&*(abcde	0!@#$%^&*(abcdeF	
1	15	!@#$%^&*(abcdef	0!@#$%^&*(abcdef	
1	16	!@#$%^&*(abcdefg	0!@#$%^&*(abcdefg	
1	17	!@#$%^&*(abcdefgh	0!@#$%^&*(abcdefgh	
8	1	*	01234567*9ABCDEF	
8	7	*(abcde	01234567*(abcdeF	
8	8	*(abcdef	01234567*(abcdef	
8	9	*(abcdefg	01234567*(abcdefg	
14	1	e	0123456789ABCDeF	
14	2	ef	0123456789ABCDef	
14	3	efg	0123456789ABCDefg	
15	1	f	0123456789ABCDEf	
15	2	fg	0123456789ABCDEfg	
16	1	g	0123456789ABCDEFg	
16	2	gh	0123456789ABCDEFgh	

4.4 Replace+Delete Tests

at	delete	insert	is	Comments
0	2	)	)23456789ABCDEF	
0	4	)!	)!456789ABCDEF	
0	8	)!@#	)!@#89ABCDEF	
0	15	)!	)!F	
0	16	)!	)!	
0	17	)!	)!	
1	2	)	0)3456789ABCDEF	
1	14	)	0)F	
1	15	)	0)	
1	16	)	0)	
13	2	)	0123456789ABC)F	
14	2	)	0123456789ABCD)	
15	2	)	0123456789ABCDE)	
16	2	)	0123456789ABCDEF)	
17	2	)	0123456789ABCDEF	

4.5 Replace+Insert Tests

at	delete	insert	is	Comments
0	1	)!	)!123456789ABCDEF	
0	2	)!@#	)!@#23456789ABCDEF	
0	4	)!@#$%^&	)!@#$%^&456789ABCDEF	
0	8	)!@#$%^&*(abcdef	)!@#$%^&*(abcdef89ABCDEF	
0	15	)!@#$%^&*(abcdefxyz	)!@#$%^&*(abcdefxyzF	
0	16	)!@#$%^&*(abcdefxyz	)!@#$%^&*(abcdefxyz	
0	17	)!@#$%^&*(abcdefxyz	)!@#$%^&*(abcdefxyz	
1	1	)!	0)!23456789ABCDEF	
1	2	)!@#	0)!@#3456789ABCDEF	
13	1	)!	0123456789ABC)!EF	
14	1	)!	0123456789ABCD)!F	
15	1	)!	0123456789ABCDE)!	
16	1	)!	0123456789ABCDEF)!	
17	1	)!	0123456789ABCDEF	

5 License

Changed-stream is distributed under LGPL2 License.