ViW

with Scala

Bob Reynders

Introduction

Hello, all questions go onto the toledo forum, not into my mailbox. You are free to discuss problems with your fellow students on the toledo forum as long as you're not giving away solutions. Do not dump parts of your own code.

You're developing a text editing plugin for the Atom editor and are making a (worse) version of the famous vi editor commands.

You have to implement certain features as well as comply with certain code quality requirements. The assignment is incremental and will be graded both automatically and manually (i.e., I will look at your code).

A flawless implementation of the base assignment gets you a passing grade. Your grades go up as you implement other features. Certain features impact the difficulty of others. Read the entire assignment and think before you act. Naively implementing movement commands make all others more difficult.

Adriaan's guidelines for writing Scala apply:

- Use OO to encapsulate implementation details
- Use FP for small abstractions (e.g., collection API methods, higher-order functions)
- Favor immutability (the entire project can be written with only 1 mutable variable – you don't have to strictly adhere to this 1 mutable variable but keep it readable and clean)

Submission Details

Before you submit check that you:

- completed the FILLME.md file with details about your project
- zip your project so that it contains everything (including the atom directory, and your tests)
- did *not* modify `Runner`, `State` or the header of the `processKey` method.

Getting Started

The project skeleton provides all Atom-specific code and some tests that are easy to write and read . You will implement the entire project through a very simple API given to you in `Viw.scala`:

```scala
def processKey(key: String, state: State): Option[State]
```

The `processKey` method takes two arguments, `key` contains a string that represents the key that was pressed. These are simple ASCII representations, for their names look at `atom/keymaps/viw.cson`. The `state` parameter, contains the state of the text editor at that moment. It contains all you need to know about the **editor**, not necessarily all the state that you will need to implement your application. You return the (optional) new state of the editor as a result of processing the pressed key.

State contains the following fields, representing the text content of the currently opened file, the position of the cursor (for simplification we assume 1 cursor), the currently selected text and whether or not `viw-mode` is enabled:

```scala
trait StateFields {
  def content: String
  def position: State.Position
  def selection: Option[(State.Position, State.Position)]
  def mode: Boolean
}
```

The plugin has two modes, normal mode and viw mode. In normal mode the editor behaves as if the plugin does not exist (contrary to vi(m)). Viw mode should be entered when alt-space is pressed (shows up as just `" "` in `processKey`). viw mode is disabled when i is pressed or whenever a certain command sequence exits.

Running and Developing Scala(.js)

Atom plugins are traditionally written in JavaScript, but in this assignment you're using Scala. We use the Scala.js compiler (a Scala compiler plugin) to generate JavaScript. Do **not** write your code in JavaScript.

I strongly suggest you look at using `sbt` (accessible in IntelliJ as Sbt shell) to run your tests. `sbt` is the scala build tool and through its commandline interface it is possible to run tests whenever a file is saved with `~test`, run only failed tests `~testQuick`, etc.

You can finish the entire assignment without touching Atom. But to get a feel for the plugin and to look at your own results it is possible to actually load the plugin. To test your plugin you have to invoke the Scala.js compiler through `fastOptJS`. It produces a big JavaScript development version that is several times that suffices for our case.

After that, use a shell to navigate to the `atom` directory in the project and use the Atom Package Manager `apm` to execute `apm link`. This creates an Atom-compatible package and links it to the plugin directory of the editor. Start up Atom (or re-start) and your plugin should be available in any text buffer so just open a file, hit alt+space and test away. To hot-load (without restarting Atom) a new version of your plugin you can use Atom's `Window Reload` action available in the `ctrl+shift+p` command palette after recreating the JavaScript through `fastOptJS`.

I did not get to test this on every single operating system or computer setup, if you have any issues, continue developing against your tests while you wait for answers on toledo.

Tests

Being able to run your code properly in Atom is a bonus, the most important thing is to have a proper implementation that works with our tests. I say this explicitly because Atom, being a text editor, already has tons of features itself. It is not alright to ignore certain implementations because default Atom behavior would fill in the gap. Make sure your intented behavior is visible in the `State` to `State` transformation.

You get one test per basic functionality but they are very easy to write yourself (and I encourage you to do so). Your code will be tested on corner cases with plenty of other tests so do not assume a passing grade just because your tests succeed.

Viw related test code is in `test/scala/viw/ViwTest`, the given tests can be found in `test/scala/viw/CommandsTest`. Make new files for your additional tests and name them properly.

The tests are very easy to read and write and consist of three parts, a command stack, a source text and an expected text. These strings are representations of the editor state and assume the absence of # signs in the content since these values are used to denote cursor position. For example, an editor state with a position of line 1 and character 2 can be represented as:

```
hello this is a test
go#o#d luck on your implementation
```

There are two main test functions implemented on top of the core test function (`viwTest`): `viwTrue` and `viwFalse` – these correspond to the exit mode of the command stack, i.e., should the editor be in viw-mode or not.

```
viwTrue(
  "l",
  """Lorem ipsum dolor sit ame#t#, consectetur adipiscing elit.
  |Cras quis massa eu ex commodo imperdiet.
  |Curabitur auctor tellus at justo malesuada, at ornare mi tincidunt.""".stripMargin,
  """Lorem ipsum dolor sit amet#,# consectetur adipiscing elit.
  |Cras quis massa eu ex commodo imperdiet.
  |Curabitur auctor tellus at justo malesuada, at ornare mi tincidunt.""".stripMargin
)
```

Means that after pressing “l” (this goes from left to right in case there are more commands), the cursor moves from ame[t], to amet[,] and that viw-mode stays activated. It assumes the activation of viw-mode at the start of the test.

Detailed tests that test more than just cursor position should use `viwTest` directly, e.g., testing on selection.

Viw Functionality

The appropriate hotkeys and their functionality are described here, for more details look at the tests. Each test describes the basic functionality. Write your own tests for corner cases and for expansions! Keybinds are case-sensitive!

Base (normal / viw mode, passing grade)

Extra requirements:

- Minimize duplicate code, several movements have very similar behavior and it would be silly to have separate implementations for all of these without sharing code.
- Keeping a history can get messy. You do not want to go overboard with shared mutable state.
- Movements are bounded by the text that is available, you cannot move out of bounds, e.g., spaces will not be added just so that you can keep moving right. This applies to all movements, e.g., j at the 80th character will not pad the line under it, it simply moves to the last character of the line below. Note though that a cursor can move 1 space past the final character (to allow editing).

Cursor Movement (by one):

h (left) cursor left by one
j (down) cursor down by one
k (up) cursor up by one
l (right) cursor right by one

Cursor Movement (by word):

w (next word) cursor to the next word
b (back word) cursor to the start of a word (if its already at the start, to the start of the previous word)
e (end of word) cursor to the end of a word

Cursor Movement (other):

$ move to the end of the line
0 move to the start of the line
% (match brackets) move to the matching brackets when cursor is on one of: (), [] or {} if the cursor is not on a bracket, go to the first bracket on that line, if there is none on that line, do nothing. This should work with nested braces, e.g., if braces are nested and the outer brace is selected, go to the corresponding brace and not just the most inner counterpart.

Modify text:

x (delete) delete a character and move to the position right to it
X (delete backwards) delete a character and move to the position left to it
D (delete line) delete from the current position to the end of the line
J (join line) joins the current line with the next line, that is, move the line below to the current line

Exit viw mode:

i (insert) exits viw mode
a (append) cursor moves one to the right and exits viw mode immediately (allows editing after the current character)
o (open) enter a newline under the current line and exit viw mode
s (substitute) delete character under cursor and exit viw mode
G (go) go to the final line in the text
I (insert in line) go to the first character of the line and exit viw mode
A (insert after line) go to the last character of the line and exit viw mode
C (change line) delete from the current position to the end of the line and exit viw mode

Misc:

. (repeat text changing command) repeat the last text modifying command e.g., x, X, D, J later this should still work with other text modifying commands

True vi(m) text modification

One of the powers of vi(m) is the ability to combine text editing commands with movements.

Extra requirements:

- Minimize duplicate code, program your movements in such a way that you can use their implementation in combination with change or by themselves to move the cursor.

Modify text:

d (delete) + movement d while moving deletes everything that was moved over. Does not exit viw mode, see c for more information.
d (delete) + d deletes the entire line

Exit viw mode:

c (change) + movement c while moving deletes everything that was moved over, e.g., c + l deletes the character to the right. c + j deletes everything right of the cursor up to the character straight down. (This does not follow vi(m) behavior to make it easier on you.) The same applies for all other movement (including more complicated ones such as match bracket), it exits viw mode.

c (change) + c deletes the entire line and exits viw mode.

Indentation

Extra requirements:

- Minimize duplicate code, program your movements in such a way that you can use their implementation in combination with indent, change, etc. or by themselves to move the cursor.

Modify text:

< or > (indent or dedent) + movement indents the code by two spaces. > + j will indent two lines, > + l or h has the same behavior and indents the entire line. Indents work strictly with lines, not with regions of lines.

< + < or > + > same but on the entire line

Pasting

p (paste) paste whatever was deleted previously using a command after the current cursor position, e.g., deleting something with x and using p will paste the deleted character. Deleting with D and pasting with p will paste the entirely deleted sequence.

P (paste behind) exactly the same as p but pasting behind the current position

Yanking

y (yank) + movement yanks text into the copybuffer, this has the same behavior as d or c and requires movement to define what should be yanked. This changes the behavior of p into "paste whatever was deleted previously OR what was yanked".

Count

Extra requirements:

- Minimize duplicate code, moving left or right are movements by 1 as well, do not settle for two sets of movement implementations.

1-9 + … All movements can be prefixed with a number to indicate how many times the movement should be executed. This also combines with actions that work with movements, e.g., d3w deletes the three following words.

Count Extra

Exactly the same as count but also allows for multiple digits, i.e., d20w.

Visual

v (visual) All movements that are being done are shown in the editor through selecting text. For example, vj would select the contents that would be deleted through dj.

V (visual line) All movements select text, except this only selects complete lines. Pressing V once selects the current line.

Find

f (find) + key a movement that jumps to the first word in the line that starts with 'key'

Come up with your own extension!

Vi(m) has plenty of popular plugins to further increase its strengths, be inspired and create your own.