COMP3258: Functional Programming

Assignment 3

Deadline: 23:59, May 05, 2025 (HKT)

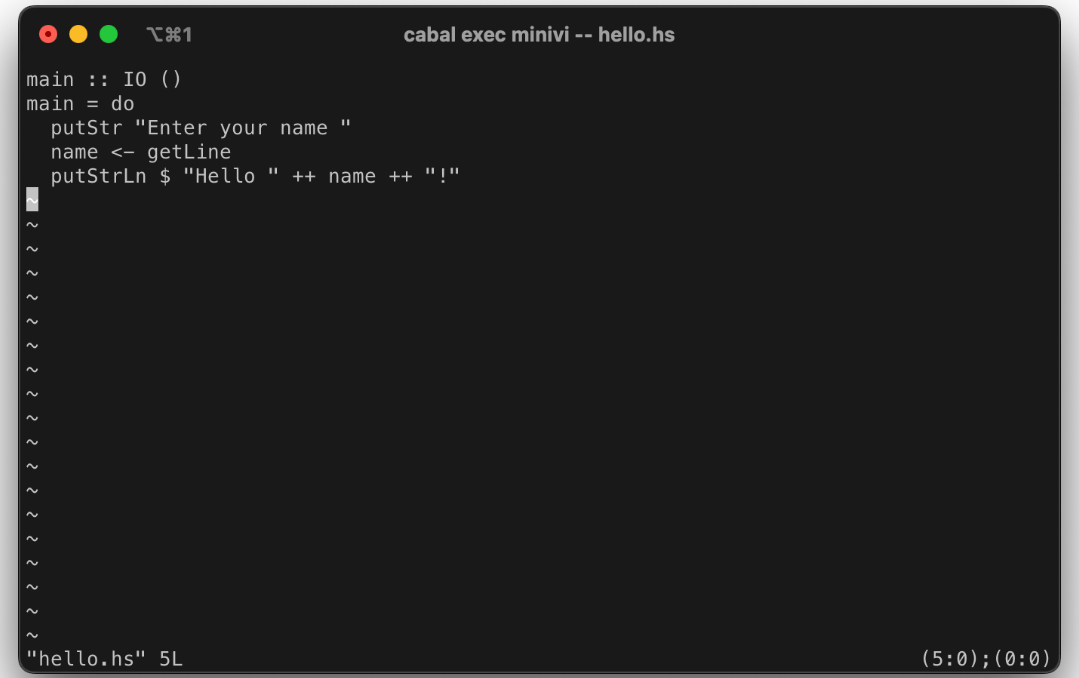
---

1. For absurd cases (don't be confused with base case), feel free to raise an exception (e.g. using `undefined`) or give a default value for it.
2. Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet (include generation tools like ChatGPT/GitHub Copilot) or were advised by your classmate, please mark and attribute the source in a comment, and explain it in a detailed manner. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism. *Additional rules may apply.*
3. The style and quality of the code will also be assessed. Check out the Haskell style guide for the recommended code style.

Caution*: This assignment involves reading from and writing to files using Haskell. Be aware that incorrect handling of file IO can lead to unexpected behaviors, data loss, or file system corruptions. Test carefully and always back up your files before running your program.*

Introduction

In this assignment, we are going to implement a simple `vim`-inspired terminal-based text editor called `minivi`.



```
main :: IO ()
main = do
  putStr "Enter your name "
  name <- getLine
  putStrLn $ "Hello " ++ name ++ "!"
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"hello.hs" 5L                                                          (5:0);(0:0)
```

Just like `vim`, our `minivi` editor is a *modal text editor.* It has the following four modes:

1. The *Normal* mode. In this mode:
   - Press `i` to go to *Insert* mode, or press `:` to go to *Command* mode.
   - Press the arrow keys or the `h, j, k, l` key to move the cursor to the left, down, up or right.
   - Pressing other keys does nothing.
2. The *Insert* mode. This is the mode where you do the text editing:
   - Press the `esc` key to go back to *Normal* mode.
   - Press the arrow keys to move the cursor.
   - Insert characters by pressing the keys.
   - Delete (and backspace) characters by pressing`delete` or `backspace`.
   - Create a new line by pressing `return`.
3. The *Command* mode. In this mode:
   - Press the `esc` key to go back to *Normal* mode.
   - Insert characters by pressing the keys.
   - Pressing the arrow keys should do nothing.
   - Press `return` to execute the command. The following commands are available:
     - `:w` write the buffered content to the file.
     - `:q` quit
     - `:q!` force quit
4. The *Message* mode* is like the *Normal* mode, but it is used to displace messages in the status bar. (i.e. telling the user that you cannot quit if there are unsaved changes)

Note that `vim` enables text wrapping by default (displaying overflowed characters in the lines below), but our editor does not support this feature.

Project Structure

In this section, we will go through the scaffolding code.

The design of `minivi` follows the *Elm Architecture* (or the *Model-View-Update* (MVU) architecture).
- A *Model* represents the *state* of the program.
- A *View* is used to describe to UI. Here, it handles the terminal rendering.
- An *Update* (function) is the only place that manipulates the model. This is where the main logic of our program sits.

This is the project structure:

```
.
├── app
│   └── Main.hs     # program entry
├── c               # c source
├── minivi.cabal
├── README.md
├── src
│   ├── App.hs      # model
│   ├── Update.hs   # update (main logic)
│   ├── Util.hs     # some helper functions
│   └── View.hs     # view (rendering)
├── test
└── <other files>
```

Model

The *model* (or state) of our program is described using the `App` data type.

```
data App = App {
    mode     :: Mode,         -- current editor mode
    buffer   :: [String],     -- text buffer
    cursor   :: Pos,          -- cursor position
    offset   :: Pos,          -- view port offset
    dirty    :: Flag,         -- flag for refreshing terminal buffer
    modified :: Bool,         -- buffer modified
    termSize :: (Int, Int),   -- terminal size
    file     :: FilePath      -- file path
  }
```

For each field:

- `mode` describes the current *mode* of our text editor, which can be one of the following:
  1. `Normal`: the normal mode.
  2. `Insert`: the insert mode.
  3. `Command`: the command mode. This mode also contains a string payload to store the command typed so far.
- `buffer` stores the whole file as a list of strings, where each string represents a line of the file.
- `cursor` stores the *cursor position for the terminal.* (to be explained later)
- `offset` stores the *view port offset.* (to be explained later)
- `dirty` describes the *dirty flag*, which can be one of the following:
  1. `FNone`: no refresh (do not update terminal).
  2. `FStatus`: only refresh the status bar.
  3. `FFull`: a full refresh for the terminal output.
- `modified` indicated whether the buffer has been modified.
- `termSize` stores the size of the terminal
- `file` stores the file name of the opened file.

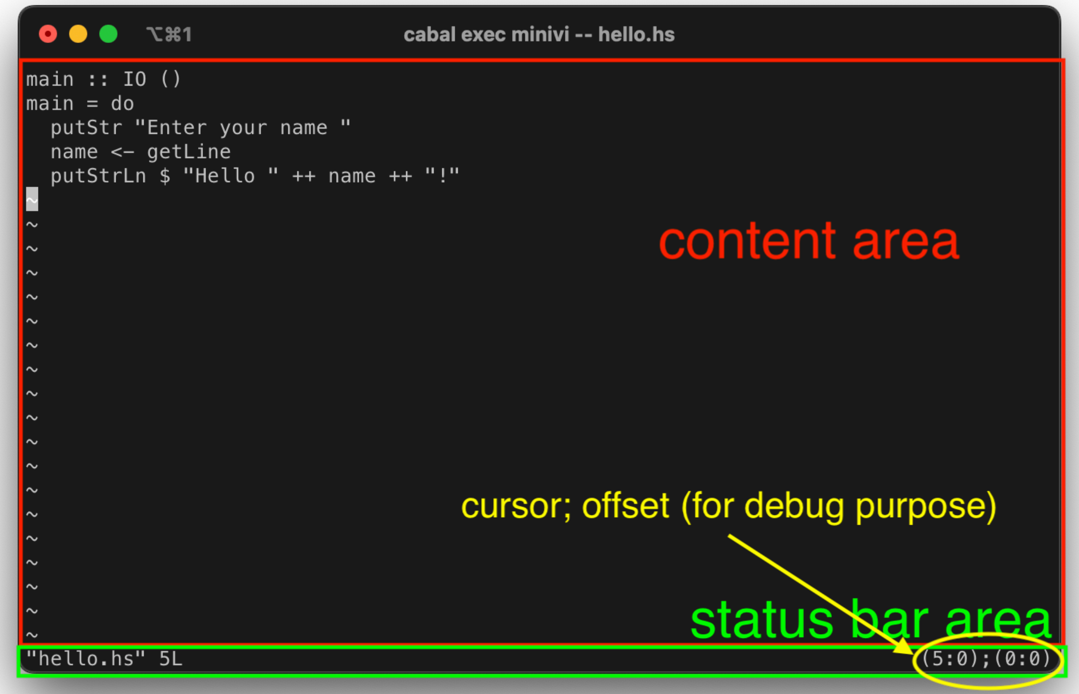
Notes

- The data type `data Pos = Pos { row :: Int, col :: Int }` represents the position of a cursor.
- The *terminal size* is a tuple (r, c), where r represents the number of *rows* (number of lines) and c represents the number of *columns* of your terminal.
- The *cursor position* (0-indexed) (x, y) is stored at the `cursor` field of `App`.
  - It represents the cursor position for the terminal (the position that *blinks* in your terminal).
  - We use x to refer to *row* and y for *column*. The cursor position $(0, 0)$ represents the *top-left* position in the terminal.
- The *view port offset* (0-index) (dx, dy) is stored at the `offset` field of `App`.
  - This is the coordinate where we start rendering the content from the buffer.
  - For example, for an offset of `(5, 10)`, we start rendering everything after row 5, and column 10.
- The *content cursor position* (0-indexed) (i, j), can be obtained via `contentPos :: App -> Pos`.
  - It it used to *index* into the buffer.
  - The coordinate (i, j) can be off-by-one. So we can have `i == length buffer`. This allows us to easily add new lines to the file or new characters to the end of a line.
  - So when (i, j) is not off-by-one, the expression `(buffer !! i) !! j` should give us the corresponding character.

View

The *View* part of our program is responsible for rendering the buffer and the status bar in the terminal. It consists of two parts:

1. The content, ranging from $(0, 0)$ to $(r - 2, c - 1)$, renders the text buffer.
2. The status bar at $r - 1$ (the last line of the terminal), renders the status bar that displays the current mode, command buffer etc.
   - To help you debugging, the bottom right corner shows the current cursor position and offset.



```
main :: IO ()
main = do
  putStr "Enter your name "
  name <- getLine
  putStrLn $ "Hello " ++ name ++ "!"
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"hello.hs" 5L                                              (5:0);(0:0)
```

⌥⌘1
cabal exec minivi -- hello.hs
content area
cursor; offset (for debug purpose)
status bar area

Here, since our offset (i, j) is $(0, 0)$, the content is rendered from $(0, 0)$ of the buffer (the character `'m'` in `main :: IO ()` of the first line).

Update

The *Update* part is the main logic of our program, which will be explained in the following section.

Tasks (90pts)

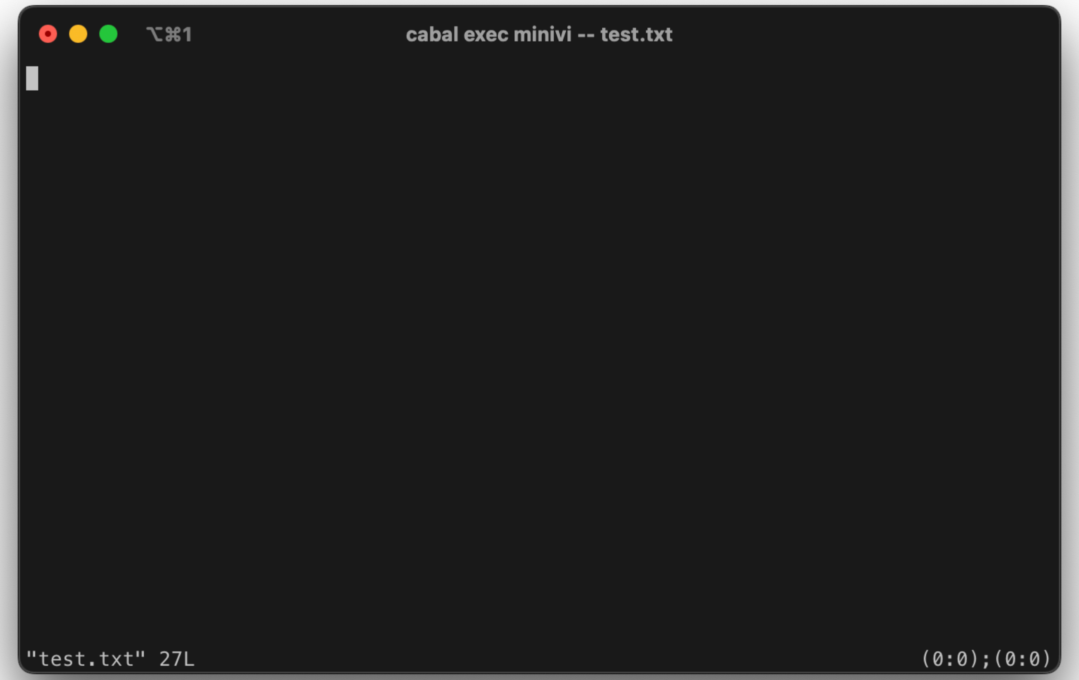
After you have downloaded the scaffolding, you can build the project using: `$ cabal build`

You can launch it with `test.txt` using: `$ cabal exec minivi -- test.txt`

Currently, what you can do is limited to:

- Pressing `esc` to *Normal Mode*.
- Pressing `i` to *Insert Mode*.
- Pressing `:` to *Command Mode*.
- Use the command `:q` to exit the program in *Normal Mode*. (i.e. typing `q` after entering *Command Mode*)
- Use `ctrl+c` to interrupt program execution.

To make the editor fully functional, please follow the instructions in the sections below.



```
"test.txt" 27L                                                (0:0);(0:0)
```

Rendering (10pts)

To start, implement the function `replaceTabs` that replaces every tab (`\t`) with a *single* space for the input string.

```
replaceTabs :: String -> String
```

Next, implement the function `bufferToContent` which takes an `App` and return the *content* (of type `[String]`) that will be printed to the terminal.

Given the current view port offset (dx, dy) and terminal size (r, c), You need generate a list of strings with $r - 1$ lines, where each line's length is c.

1. Keep only the items after the offset (dx, dy). Then,
   - Adjust each line to be length of c by either keeping the first c characters or appending spaces.
   - Adjust the content to have $r - 1$ lines by either keeping the first $r - 1$ lines or append lines starting with `'~'` of length c.
2. Use `replaceTabs` to replace tabs with spaces.

```
bufferToContent :: App -> [String]
```

Notes

1. The content returned by `bufferToContent` should have `r-1` rows and `c` columns for each row.
2. Test cases are located in the `test` directory.
3. You can run `$ cabal test` to test your implementation.

Command Handling (10pts)

Update the function `handleCommand :: App -> String -> Key -> IO App` to handle the following commands:

- `:q`: check the `modified` flag,
  - If the buffer is modified, update `mode` to *Message* mode with the message `"Buffer modified, use ! to force quite"`, and set the `dirty` flag to `FStatus`.
  - Otherwise, use `exitSuccess` to exit.
- `:q!`: use `exitSuccess` to exit directly.
- `:w`: write the `buffer` to file, and
  1. update `mode` to *Message* mode with the message `"File written"`,
  2. set the `dirty` flag to `FStatus`,
  3. change the `modified` flag to `False`.

Note

~~1. When writing the file, check if you have accidentally appended an empty line to the file.~~

- Updated: Use `unlines` (from *Prelude*) to convert `buffer` to the string that will be written to the file.

Cursor Movement (30pts)

The cursor movement relies on the `updateCursor` function, which takes the app and the **target** *content cursor position* as input. And returns the app updated with:
- the new `cursor` (*cursor position*),
- the new `offset` (*view port offset*),
- the new `dirty` flag:
  - If `offset` has changed or the original `dirty` flag is `FFull`, set it to `FFull`,
  - Otherwise, set the `dirty` flag to `FStatus`.

```
updateCursor :: App -> Pos -> App
```

Hints: When should you update `cursor`? What about `offset`?

Note

- If the current cursor position and offset are (x, y) and (dx, dy), and the target position is (i, j) (in bound/off-by-one). After the update, we get the new cursor position (x', y') and new offset (dx', dy'). We should always have the invariant $x' + dx' = i$ and $y' + dy' = j$.
- If the target position is out of bound (off by 2 etc), you should set it to the **nearest** valid position.
- Test cases are located in the `test` directory.
- You can run `$ cabal test` to test your implementations.

Text Editing (40pts)

The text editing function is built upon the following three functions. Your implementation should resemble how a normal text editor works.

insert

The `bufIns` function handles the *insert* feature, which insert one character **before** the cursor.

It takes the character to be inserted, *content cursor position* (i, j), the buffer, and returns the updated buffer.

```
bufIns :: Char -> Pos -> [String] -> [String]
```

delete

The `bufDel` function handles the *delete* feature, which removes one character **before** the cursor.

It takes the *content cursor position* (i, j), the buffer, and returns the updated *buffer*, and the updated *content cursor position* (i', j').

```
bufDel :: Pos -> [String] -> ([String], Pos)
```

Hint: you may need to move to the end of last line in some cases.

return

The `bufRet` function handles the *return* (or new line) feature, which moves everything **at and after** the cursor to a newline.

It takes the *content cursor position* (i, j), the buffer, and returns the updated *buffer*, and the updated *content cursor position* (i', j').

```
bufRet :: Pos -> [String] -> ([String], Pos)
```

Notes

- Test cases are located in the `test` directory.
- You can run `$ cabal test` to test your implementations.

Bonus (10pts)

While this section is named "Bonus", it is still mandatory. To claim the bonus points, you need to select **ONE** feature from the feature descriptions to implement.

For most students, please implement **one simple feature** listed below.

Also note that:

- The score you get from this section is **not** tied to the difficulty of the feature you implement. So you can get the full 10pts by implementing a simple feature.
- In addition to the implementation, you also have to *briefly document* the features that you have implemented.

Simple features (for most students)

1. Extend the editor, so the user can:
   - Start the program without the initial file (i.e. launch using `$ cabal exec minivi`)
   - Specify the file name for the *write* (or save) command (e.g. `:w hello.txt` to write to `hello.txt`)
2. Extend the navigation functions in *Normal* mode:
   - Press `w` to jump to the start of next word, and `b` to jump back to start of last word.
   - Press `0` to jump to the first character of the line, and `$` to jump to the last character of the line.

Hints for simple features:

1. *Please try these features in your vim. But you don't need to replicate its behavior exactly.*
2. *If you select feature 1, you need to:*
   - *Without modifying the definition of `file :: FilePath` in `App`, how can you know if the program is started with a file or not?*
   - *Update the command parser (you don't need a sophisticated parser here) to parse commands like `:w hello.txt`.*

   - *Update the `prepare :: IO App` function in `Main.hs` to correctly initialize an `App`.*
3. *If you select feature 2, you need to:*
   - *Figure out how to move the cursor accordingly.*
   - *You may need to jump to the next/last line for the word.*
   - *To simplify, we consider words as consecutive characters separated by spaces.*

Moderate features (for advanced students)

1. Implement the following feature in *Normal* mode:
   - **Undo** last change (e.g. character insert, delete) by pressing `u` (lower case `u`).
   - **Redo** last change (e.g. character insert, delete) by pressing `U` (upper case `U`).[1]
   - You should keep the editing history of round 4 steps, so we can undo/redo 4 times.
2. Implement the following feature:
   - Start a simple text **search** by pressing the key `/` (slash) in *Normal* mode.
   - After typing the search string (e.g. type `/hello` to search the word `hello`), press `return` to enter the search mode and jump to the first matching location.
   - Press `n` to jump to **next** search result and `N` to jump to **last** one.

Note for implementing moderate features:
1. *You may need to modify existing definitions (e.g. adding cases to `Mode`, or adding fields to `App`)*
2. *You must document all your changes and provide justifications for why such modification is necessary.*

Notes

Grading

Grading of this assignment is based on the features you have implemented. Points may be deducted if your program crashes unexpectedly.

System Requirements

To successfully build and run the project, you need:

1. The Haskell toolchain: `ghc` (recommended version `9.4.8`), and `cabal` (recommended version `3.12`).
2. A *unix-like* OS (e.g. *macOS*, *linux* etc). If you are using *Windows*, you can use *WSL2*.
3. An *ANSI* compatible terminal emulator. Any modern terminal emulator should be enough (i.e. iTerm2, macOS's built-in terminal, KDE Konsole, Windows Terminal, VSCode's built-in terminal).

If you have encountered any technical issues when setting up the project, please search on the internet for solution or ask TA for help.

Use of other modules

While most of the work can be done without using modules other than a few list functions, you are still allowed to import modules provided by:
- `base` (e.g. `System.IO`, `Data.List`, `Data.Char` etc)
- `containers` (e.g. `Data.Map`, `Data.Set` etc)
- `array` (e.g. `Data.Array`)

Advanced students may check out the *advanced* section in `minivi.cabal` for additional rules.

[1]Press `ctrl+r` to redo in vim.

Modifying existing definitions

Unless explicitly stated separately:

- You should **NOT** modify any existing definitions in `Util.hs`.
- You should **NOT** modify any *data type definitions* or *function signatures*.
- You should **NOT** modify `minivi.cabal` and `c/terminal.c`.
- You should **NOT** add extra Haskell files (i.e. all your codes should be put to the existing files).

Advanced students may check out the *advanced* section in `minivi.cabal` for additional rules.

Submission

Make sure your code can be compiled before submission.

Please submit a single zip file `A3_XXX.zip`, with `XXX` replaced by your UID, which includes the following directories/files:

```
.
├── app
│   └── Main.hs
├── c
│   └── terminal.c
├── minivi.cabal
├── README.md
├── src
│   ├── Main.hs
│   ├── Update.hs
│   ├── Util.hs
│   └── View.hs
└── test
    └── Test.hs
```

You can also use the following shell command to create the zip file:

```
$ zip -r A3_XXX.zip README.md minivi.cabal app src test c
```

After compressing your files, you may want to unzip it and check if it can be successfully compiled.