

# Yi

## An Editor in Haskell for Haskell

Jean-Philippe Bernardy

Computer Science and Engineering, Chalmers University of Technology  
bernardy@chalmers.se

### Abstract

Yi is a text editor written in Haskell and extensible in Haskell. We take advantage of Haskell's expressive power to define embedded DSLs that form the foundation of the editor. In turn, these DSLs provide a flexible mechanism to create extended versions of the editor. Yi also provides some support for editing Haskell code.

*Categories and Subject Descriptors* D.2.3 [Coding Tools and Techniques]: Program editors

*General Terms* Design, Languages

*Keywords* Editor, Haskell, Functional Programming

### 1. Motivation

All software developers want to customize and extend their editor. We spend so much time working with editors that we want them to behave exactly as we wish. Using Haskell as an extension language is promising, because it is both general purpose and high-level. This combination of properties means that extensions and configurations can remain concise, and still have unrestricted access to external resources, for example by to existing Haskell libraries.

Also, users generally want to experiment with changes without prior study of the system they tweak. The well-known safety features of Haskell are very useful in this case: users can tinker with the editor and rely on the type system to guide them in writing correct code.

### 2. Overview

Yi is a text editor implemented in Haskell and, more importantly, extensible in Haskell. It is structured around four embedded DSLs:

**BufferM** A DSL for all buffer-local operations, like insertion and deletion of text, and annotation of buffer contents. It can be understood as a monad that encapsulates the state of one buffer.

**EditorM** A DSL for editor-level operations, e.g., opening and closing windows and buffers. Operations involving more than one buffer are handled at this level too.

**YiM** A DSL for IO-level operations. There, one can operate on files, processes, etc. This is the only level where IO can be done.

```
import Yi
import Yi.Keymap.Emacs as Emacs
import Yi.String (modifyLines)

increaseIndent :: BufferM ()
increaseIndent = do
  r <- getSelectRegionB
  r' <- unitWiseRegion Line r
  -- extend the region to full lines
  modifyRegionB (modifyLines (' ':)) r'
  -- prepend each line with a space

main :: IO ()
main = yi $ defaultConfig {
  defaultKm =
    -- take the default Emacs keymap...
    Emacs.keymap <|>
    -- ... and bind the function to 'Ctrl->'
    (ctrl (char '>') ?>>! increaseIndent)
}
```

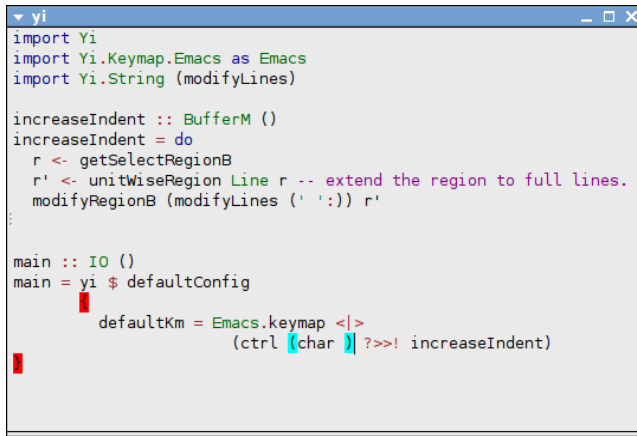
Figure 1. Configuration file example.

**KeymapM** Key-binding descriptions. The structure of this DSL closely follows that of classic parser-combinator libraries. The semantics are a bit different though: the intention is to map a stream of input events to a stream of actions, instead of producing a single result. The actions of the output stream can come from any of the above DSLs.

Yi also contains user-interface (UI) code for rendering the editor state, and getting the stream of input events from the user. Finally, there is some glue code to tie the knot between all these components. This glue is the only part that accesses the UI code.

The structure described above is very flexible: there is very low coupling between layers. One can easily swap out a component for another in the same category. For example, the user can choose between various UI components (vty, gtk, cocoa) and key-bindings (emacs, vim).

The various DSLs have composability properties, and this makes them convenient to extend and configure the editor. This is illustrated in figure 1, where a simple extension is shown. In that example, the user has defined a new **BufferM** action, `increaseIndent`, using the library of functions available in Yi. Then, he has created a new key-binding for it. Using the disjunction operator, this binding has been merged with the emacs emulation key-map. A more typical example would involve many more functions, and could call various Haskell packages to make their capabilities available within the editor, but the structure would remain essentially the same.



```
import Yi
import Yi.Keymap.Emacs as Emacs
import Yi.String (modifyLines)

increaseIndent :: BufferM ()
increaseIndent = do
  r <- getSelectRegionB
  r' <- unitWiseRegion Line r -- extend the region to full lines.
  modifyRegionB (modifyLines (' ' :)) r'

main :: IO ()
main = yi $ defaultConfig
      defaultKm = Emacs.keymap <|>
                (ctrl [char ] ?>>! increaseIndent)
```

**Figure 2.** Screenshot. The configuration file is being edited, and Yi gives feedback on matching parenthesis by changing the background color. The braces do not match because of the layout rule: the closing one should be indented. Yi understands that and shows them in a different color.

We see that Yi is not so much an editor than a rich library for building editors. Indeed, this is exactly how users create extended versions of Yi: they create a program *from the ground up* by combining the higher-order functions and (lazy) data structures offered in the Yi library. This approach to configuration was pioneered by Stewart and Sjanssen [5] for the XMonad window manager.

### 3. Editing Haskell code

Being implemented and extensible in Haskell, it would be natural that Yi had extensive support for *editing* programs, and in particular Haskell code. At the time of writing, we have implemented this partially. Syntax of programming languages can be described using lexers and a parsing combinator library. When a syntax is associated with a buffer, its content is parsed, incrementally, and the result is made available to the rest of the code.

We take advantage of this infrastructure to provide support for Haskell: among other things, feedback on parenthesis matching is given (as shown in figure 2), and there is simple support for auto-indentation.

### 4. Limitations and Further work

The parsing mechanism is not perfect yet: we only have a coarse-grained syntax for Haskell, and the error-correction scheme is lacking generality. A further step will be to bind to Haskell compilers, and in particular GHC, to provide full-fledged IDE capabilities, in the fashion of Visual Haskell [2].

Yi is also lacking dynamic capabilities: while the configuration mechanism is flexible, activating a new configuration requires *restarting* the editor. We plan to solve this problem by saving the editor state before restart and reloading it afterwards. This approach is feasible because the state of the editor is a purely functional data structure.

We point the interested reader to the Yi homepage [3] for further information, and to Hackage [1] for downloading and installing Yi.

### Acknowledgments

The Yi project was started in 2004 by Don Stewart [4]. Yi has had more than forty contributors since then —too many to cite individually— but they shall all be thanked for sharing the load in pushing Yi forward. I would like to mention a few of them, though: the early adopters Allan Clark and Corey O’Connor and the current maintainer of the Vim key-bindings, Nicolas Pouillard. I am also grateful to my colleagues Gustav Munkby and Krasimir Angelov for the local support they provided, in addition to their contributions.

Finally, the Haskell community as a whole helped enormously in making Yi a reality: the Glasgow Haskell Compiler and the numerous Haskell libraries available on Hackage [1] form an excellent platform for the development of Yi.

### References

- [1] hackageDB: a repository for Haskell packages. URL <http://hackage.haskell.org/packages/hackage.html>.
- [2] K. Angelov and S. Marlow. Visual Haskell: a full-featured Haskell development environment. In *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 5–16, New York, NY, USA, 2005. ACM.
- [3] J.-P. Bernardy et al. The Yi page on the Haskell wiki. URL <http://haskell.org/haskellwiki/Yi>.
- [4] D. Stewart and M. M. T. Chakravarty. Dynamic applications from the ground up. In *Haskell ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2005. ACM Press.
- [5] D. Stewart and S. Sjanssen. Xmonad. In *Haskell ’07: Proceedings of the ACM SIGPLAN workshop on Haskell*, page 119, New York, NY, USA, 2007. ACM.