TextScape: Making Your Own Editor is Easy, Right?

Leland Miller, Michael Kell

2014-03-21 Fri

Contents

1 Intro

Have you ever found yourself sitting at your computer, hacking away in C, Python, Haskell, or any languages for that matter, when you thought to

yourself, "Man I hate this editor! Who made the decision to bind that unchangeable hokey just out of my pinky's reach? Who made it so you couldn't customize this feature? And how come it doesn't have that feature, which my other favorite editor has?" These types of questions, really complaints for that matter, were the motivation that propelled the following project about creating our own simple, yet complete, editor. Ultimately, we realized that the types of questions we had at the outset of this endeavor were paltry in comparison to the complexity that goes into building an an editor such as vim, emacs, or yi.

2 Lets Begin With Some Simple Questions

To begin with, there are several major decisions you have to make, when designing an editor, before a single line of code is written. And the decisions you make almost play out like the questions that get answered by the end of every episode, of every cop drama, where the the who, what, when, where, and why of the crime committed in the episode get answered:

- You first have to ask yourself who this editor is for? Is this editor just for you, for a group of friends and/or business associates, for the world? Is it an open source editor that will constantly be in a process of evolution/upgrades as users give input/feedback, or have you honed down your design decisions and will one or two updates of your editor be sufficient for its usage in the long run?

- Second, what is the main purpose of your editor? Is it for hacking in a specific language, or is it meant to be as universal/portable as possible? Is it meant to modify/improve the features of an existing editor? Are you going to combines several features of existing editors to create a Franken-editor that shall obey your every command?!?

- Third, if your editor has a scheduled release date, then the notion of when your editor is due is a very important design parameter. If you open up beta testing of your editor in six months, then you might have to cut certain features out of the initial release. And if you have no strict release date, then you can take your time and add as many features, facets and functions to your editor as you like. However, the answer to the question of when your editor is due will have a profound effect upon the final release of your editor.

- Next, there's the where question, which in this case is "where shall you choose to place your allegiance in the many realms of computer languages?" Put a little more simply, with so many editors already out there, where will you draw influence from in term of your editor? With so many existing languages/editors to choose from, picking the specific instances where to draw influence from is possibly the most important design choice, since it will provide a foundation/roadmap that will guide you in the process of completing your own editor.

- Finally, it's as simple as why? This question is intricately tied to all of the previous questions, but understanding the driving force of why you decided to make your editor is a huge design question. This is a very simple question but requires an immense amount of thought, and is very different for the implementation of any individual editor.

It was really important for us to ruminate upon these questions before we wrote a single line of code, and was the most incredibly, mind-boggling aspect of this project. Unfortunately, the answer to these questions for our editor came down mostly to issues of simplicity, which was a direct result of not only time constraints, but the overall complexity of editor implementation as a whole. Lets explore our answers to a few of these questions:

2.1 What

When we really started to map out the structure of our editor we had a few major design conceptions of what we wanted our editor to look like. We definitely wanted to work with Haskell in one way or another, not only because functional programming was the major focus of our class, but also because we wanted to gain a better grasp of a functional programming language. Ultimately, this is why we chose to write the interpreter for our editor in Haskell. Since the process of writing this interpreted took up the majority of our time, we ultimately chose code to our server and editor in Ruby and JavaScript, respectively, since there are a ton of pre-existing libraries/wrapper scripts in existence that would make this end of the project less time consuming, and ultimately allow us to complete a working version of our editor.

2.2 Where

In terms of drawing influence from other editors/languages, we first needed to think about the Domain Specific Language we would build for our editor, followed by the rest of the editor. Since we spent so much time of the

implementation of our interpreter, we kept the actual look of our editor quite simple, to ensure the functions/features of our editor were working properly.

2.3 Why

The real motivation behind writing TextScape came as a result of one main motivation outside of the fact that we had to create a project for this class. Since us programmers use editors on a daily basis, it is undeniable that they are an integral part of our lives. Since this is the case, we should have a pretty good understanding of how editors work. In this sense, the real motivation of our project was to gain an even better understanding of the complexity of editors by building our own. This process would force us to think about the entire scope of not only crafting our own Domain Specific Language, but also implementing a relatively basic editor to engage with that language. Lets move on the some of the specifics of TextScape...

3 Language

It is fitting to start at with the language when discussing our project, as the rest of the project is built on top of it. Our original intent was to take influence from EMACS and focus on building a language interpreter, and then build a text editor in it. In designing the language, we came up with some interesting ideas that changed the course of our project. The language of TextScape is very simple, but also very powerful. Though we were not able to include everything we would have liked in the language, it is fairly functional in its current state. This section will discuss TextScape in its current state.

3.1 Memory Model and Data Structures

In the language design there are two fundamental data constructs, variables and namespaces. However, in the implementation of the language we added lists, and kernel functions to the core of the language, this gave us a data system that all together included:

1. Variables
2. Namespaces
3. Kernel Functions

4. Lists

The entire interpreter environment is referenced through a single namespace object known as the root. Every item then exists as a node in a symbol tree rooted at the root namespace.

Although, ideally to the spirit of the language that developed, we thought about generalizing lists to variables, but did not have time. The kernel functions are internal functions that are coded into the interpreter, and all user defined functions are represented as variables.

Variables are simply textual data stored on the symbol tree. This means that user defined functions are stored as there source on the symbol tree and evaluated when necessary. This was probably the part of the project that was most interesting, it allowed the editor to directly edit the current execution environment by modifying the source code of the function that were already in the symbol tree. It also allows for many meta-programming opportunities, as functions can be modified in the same way as any other variable.

3.2 Syntax

We decided to borrow the parenthesized list based syntax of the lisp family of languages, due to its elegant simplicity. It also makes parsing a little bit easier. This means that the basic element of TextScape syntax is a list that looks like:

```
(e1 e2 ... en)
```

Every list represents a computation of `e1` on the elements `e2` through `en`. The argument list can be overloaded, and the functions only fail if trying access arguments that were not defined. The evaluator takes these lists one by one and computes the result. Every computation can return a result, and the list is replaced by the returned value in its context. If the list is in the top-level context, then the evaluator returns its result to standard output (if the evaluator is evaluating multiple lists it discards all but the last result.

Each one of the list elements `e1` through `en` can be one of several types:

1. A symbol table reference
2. A literal
3. Another parenthesized list

4. A pointer parenthesized list

All of these types can also be a part of a record type which is represented as `name:*`, where name is the name of the record and star is a type. For example `filename:/in.cpp/` would set an argument called filename to the value "in.cpp".

Also note that a pandoc (markdown) style source file can be loaded using the loadPandoc function provided by stdlib.ts. For more information on creating these source files see stdlib.ts or test.ts.

- Symbol Table Reference
  Symbol table references are expressed as text strings with no delimiters. For example `cat` is a symbol table reference referring to an entry "cat" under the root namespace. A symbol reference can refer to any symbol entry using its fully qualified path separated by periods. So `Kernel.Buffer.new!` refers to the entry "new!" in the namespace "Buffer", which is then in the namespace `Kernel` under the root namespace.

  In the evaluation of a statement, these references are replaced by their values. The convention of an exclamation point is used to separate functions from other types.

- Literals
  Literals are elements containing text that is directly used in computation, such as setting the values of variables. All variables are what would be considered strings in many languages, so literals are as well. There are currently two literal syntaxes, forward slashes can contain literals, as in `/this is a literal/`, and a pound, or hash sign, followed by an identifier contain a string until the next pound sign with that given identifier, such as `#T this is a literal #T`. These can both be used across newlines, and the pound sign notation trims the outer whitespace around the literal, so `# hello #` would become "hello".

  Very basic escaping is included in the language, including `//` becomes "/", and `\n` becomes a newline.

  Since the language relies so much on text processing, the literal system could definitely use some enhancement. The pound sign notation in particular is buggy in its current state.

- Parenthesized Lists & Pointer Lists
  Parenthesized lists are computed, and are replaced with the result of

their computation as earlier stated, and the computation continues in the context of the calculations side-effects (note arguments are evaluated from left to right).

Pointer lists are of the form `*(e1 e2 ... en)`. For this construct, the list is first computed, but the result is considered a symbol table reference instead of a literal in the context of its containing list.

3.3 Defining Functions in TextScape

Since functions are just saved as variables, defining a function is like defining any other variable. Use the let function to bind a symbol name to a value that represents the function. For example:

```
(let /sayHello/ /(cat //Hello!//)/
```

Defines a function `sayHello` that returns the value "Hello!". However, thats not much fun. To use arguments, we use `@i`, where i is the index of an argument to refer to the first anonymous argument and `$name` to refer to the record argument called name. For example:

```
(let /sayHello/ /(cat //Hello, // $name // . You are // @0 // years old.)/)
(sayHello name:/Leland/ /25/)
```

Defines a function sayHello that takes a named argument name and the first anonymous argument and returns a message. Anonymous arguments and named arguments can be used interchangeably.

4 Implementation

4.1 Interpreter

As previously mentioned, the bulk of our project was spent writing the interpreter for our project. We broke this down into four major sub-parts: our parser, evaluator, kernel, and data storage tree/functions. On top level, there is a main function that uses these four major subparts to interpret an expression input into the editor, and ultimately perform the appropriate command. However, each of these sub-parts provided its own special challenges.

- Parser
  Because of the nature of our syntax, parsing was a fairly simple task. We used the parsec library, which is built into Haskell in newer version.

- Evaluator
  Since TextScape was built as an expression based language, the parsed input is nothing more than an expression, stored as a string, that is then evaluated to see what command needs to be performed. However, there was one clear distinction we had to make in regards to the evaluation of an expression: does the expression involve pure functions, or impure functions? This clear differentiation between pure and impure functions was absolutely necessary since the main focus of our project deals with I/O. To deal with this question, we built a simple kernel where all of our internal functions were stored.

- Kernel
  This portion of our interpreter proved to be the most entertaining to implement. Having spent a good amount of time working with imperative programming languages, this was the most accessible part of writing our own editor in the beginning. We wanted to implement a few simple pure commands, such as cd, and ls, to make our editor have some of the basic commands we used while learning programming on the school Unix server. The other major pure command was the actual eval command, which would evaluate a user input statement, such as 2+2. For impure functions, there was also the simple impure command cat that became one of our major output commands. However, we also implemented the LISP convention of let so that user could define their own functions in our editor and then subsequently call them as they saw fit. However, for this to occur we needed one final piece to tie hold our interpreter together.

- Data
  Of course, since an editor involves storing user input data, we needed to build our own simple data structure to store all the user input expressions. This process involves not only storing new data, but also accessing or updating that data if necessary.

4.2 Server

The server is nothing more than a simple Ruby script that uses the WEBrick library, which is built into newer versions of Ruby, to serve our web files and provide an HTTP interface to the TextScape interpreter. The interface is used by sending a POST request to the server under the "/run" path.

4.3 Editor

The editor is where all the fun is. The editor is accessed from the root directory while the server is running (for example "http://localhost:1234"). On the right is a tree that shows all of the entries in the symbol tree and updates live. On the right is an editor that allows editing of variables selected by clicking on symbols in the tree. There is a commit button on the top menu to the right that pushes the text in the editor back into its symbol entry. There is also a shell in the bottom half of the screen. Input is done on the bottom line, the enter or return key runs the commands and the result of the function is displayed in the panel above the prompt.

The editor uses the jsTree library and the Ace editor library.

5 Room for Improvement

Of course, in a project with a time constraint such as this there are always many room for improvements. This is definitely a project that either of us may continue working on into the future.

Most immediately, there is a severe lack commentation in the source code, mostly due to a rapid rewrite of the program to clean up a lot of code in the last minute. Please excuse us for this. The code should mostly self-documenting ;-). There are about a million other features we would have liked to include, but hey... we did what we could.

6 Conclusion

When all was said and done, the project proved to be an insightful challenge that gave us a far greater respect for editors. Of course, we had a general understanding of how editors worked beforehand, since we had both used many different types of editors of the course of our years of programming, but this project ultimately gave us the means to engage with what ended up being very intriguing design/creation process. The project began with some broad generalizations about concepts/abstractions, in terms of the five questions we dealt with at the beginning of this report, and eventually lead to the coding of all the individual features that we thought were most important for our editor. Of course, there were challenges along the road to finishing our first version of TextScape, but this was a learning process in and of itself: we had to learn to make sacrifices in terms of our some of our grander goals, in order to focus upon building a editor. All in all, building TextScape was

not only an lesson in the appreciation of editors as an invaluable tool for programmers.