Monadic Design Patterns for the Web

PrePrint™ Edition
Excerpt

artima
ARTIMA PRESS
WALNUT CREEK, CALIFORNIA

Buy the Book · Discuss

Chapter 1

Motivation and Background

Complexity management is a major issue facing the professional programmer today. A programmer building an Internet-based application interacts with, on average, no less than a dozen technologies. These applications need nearly continuous operation: 24-7 availability in order to service hundreds to thousands of concurrent requests. Developers need the tools to manage that complexity and this book serves that need.

The design patterns expressed in this book have been developed for nearly 50 years to address exactly these concerns. Scala, though not nearly as old, turns out to be an ideal framework for both realizing these patterns and discussing their ins and outs, and pros and cons. However, since these patterns don't originate in Scala, you can guess they have some significant applicability to the other 11 technologies the modern professional programmer juggles.

Note: This book will make more sense to you if you've already spent at least 5,000 hours working with either Scala or some other functional programming language. Now, if you've put in close to four years at a hobbyist level (say about 25 hours per week), you've met this requirement. Alternatively, if you've put in less than two years as a professional working solely in functional languages, you've also met the requirement.

1.1 Where are we?

How would we identify where we are in the development of computing as it relates to what humans need computers to do now and what humans might need computers to do in the future? There are three characteristics of this particular moment that I call:

- The concurrency squeeze – from the hardware up and the web down programming is more and more about concurrent and distributed execution.
- Ubiquitous access to robust VM-based computing – The JVM and the CLR have proven that VM-based computing has a significant role to play in industrial scale software.
- Advances in functional computing – monadic technology has provided a beautiful resolution to the problems that got `LISP` thrown out of industry in the 1980's.

Each of these characteristics plays a specific role in the adoption of particular approaches to structuring computations that get useful things done for humans. These approaches, in turn, influence the kinds of solutions we are likely to see in websites, mobile applications and servers. In a circular motion, these influence the kinds of features we see in programming languages, frameworks and platforms. These feedback loops between society and its tools are one of the central reasons why computing phenomena undergo such accelerated rates of change and why knowing where we are is such a key part of knowing where we are heading.

The concurrency squeeze

It used to be fashionable in academic papers or think-tank reports to predict, and then bemoan, the imminent demise of Moore's law, to wax on about the need to "go sideways" in hardware design from the number of cores per die to the number of processors per box. Those days of polite conversation about the oncoming storm are definitely in our rearview mirror. Today's developer knows that if her program is at all commercially interesting then it needs to be web-accessible on a 24-7 basis; and if it's going to be commercially significant, it will need to support at least hundreds, if not thousands, of concurrent accesses to its features and functions. This application is most likely hosted by some commercial outfit – a Joyent, an EngineYard, an Amazon EC3, and so on – that is deploying her code over multiple servers, each of which is in turn a multiprocessor with multiple cores. This means that from the hardware up and from the web down today's intrepid developer must deal with parallelism, concurrency, and distribution.

Unfortunately, in mainstream programming languages, the methods that deal with these different aspects of simultaneous execution cannot support development at this scale. The core issue is complexity. The modern application developer faces a large range of concurrency and concurrency control models, from transactions in the database to message-passing between server components. Whether to partition her data is no longer an option; instead, this developer must think hard about *how* to partition her data and whether or not this "eventual consistency" approach will liberate her or bring on a new host of programming nightmares. By comparison, thread packages seem like quaint relics from a time when concurrent programming was a hobby project she did after hours. The modern programmer needs to simplify her life in order to maintain a competitive level of productivity.

Functional programming provides a sort of transition technology. On the one hand, it's not much of a radical departure from mainstream programming like Java. On the other, it offers a simple, uniform model that introduces several key features that considerably improve productivity and maintainability. Java brought the C/C++ programmer several steps closer to a functional paradigm, introducing garbage collection and type abstractions, such as generics and other niceties. Languages like OCaml, F#, and Scala go a step further, bringing the modern developer into contact with higher-order functions; the relationship between types and pattern matching; and powerful abstractions, like monads. Yet, functional programming does not embrace concurrency and distribution in its foundations. It is not based on a computation model, like the actor model or the process calculi, in which the notion of execution is fundamentally concurrent. That said, it meshes nicely with various concurrency programming models. In particular, the combination of higher-order functions (with the ability to pass functions as arguments and return functions as values) together with a monad's structuring techniques make models, such as software transactional memory or data flow parallelism, quite easy to integrate. In addition, pattern-matching makes the message-passing style easier to incorporate.

Robust VMs are everywhere

Another reality of the modern programmer's life is the ubiquity of robust, high-performance virtual machines. Both the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) provide managed code execution environments that are not just competitive with their unmanaged counterparts (such as C and C++), but are actually the dominant choice for many applications. This has two effects that are playing themselves out in terms of industry trends.

First, it provides some level of insulation between changes in hardware design (from single core per die to multicore, for example) that impacts execution model and language-level interface. To illustrate the point, note that these changes in hardware have impacted hardware memory models. This has a greater impact on the C/C++ family of languages than on Java because the latter is built on an abstract machine that not only hides the underlying hardware memory model, but more importantly can hide changes to the model. You may, in fact, contemplate an ironic future in which this abstraction alone causes managed code to outperform C/C++ code because of C/C++'s faulty assumptions about the best use of memory that percolate all through application code.

Second, it completely changes the landscape for language development. By providing a much higher level and more uniform target for language execution semantics, it lowers the barrier to entry for contending language designs. It is not surprising, therefore, that we have seen an explosion in language proposals in the last several years, including Clojure, Fortress, Scala, F#, and many others. Note that all of those languages are either functional languages or object-functional languages, and that most proposals are either functional, object-functional, or heavily influenced by functional language design concepts.

Advances in FP: monads, and the awkward squad

Perhaps a chief reason for the popularity of functional programming language design is that the core of the functional model is inherently simple. The rules governing the execution of functional programs (the basis of an abstract evaluator) can be stated in half a page. In some sense, functional language design is a "path of least resistance" approach. A deeper reason for adopting functional language design is that the core model is *compositional*; that is, enriching the execution semantics amounts to the enrichment of the semantics' components. I can say much more about this, but I need to defer to a point where more context has been developed. Deep simplicity and compositionality are properties and principles that take some time to appreciate. Even the most impatient of pragmatic programmers can easily understand some of the practical reasons that recent language design proposals have been so heavily influenced by functional language design principles: functional language design has made significant and demonstrable progress addressing performance issues that plagued it at the beginning. Moreover, these developments significantly apply to the situation related to concurrent execution in which the modern programmer finds herself now.

Since the mid-1980s, when Lisp and its progeny were rejected by the industry for performance failures, excellent work has been put toward rectifying many of the problems those languages faced. In particular, while Lisp implementations tried to take a practical approach to certain aspects of computation, chiefly having to do with side-effecting operations and I/O, the underlying semantic model did not seem well-suited to address those kinds of computations. And yet, not only are side-effecting computations and especially I/O ubiquitous, using them led (at least initially) to considerably better performance. Avoiding those operations (sometimes called *functional purity*) seemed to be an academic exercise not well-suited to writing real-world applications.

However, while many industry shops were discarding functional languages except for niche applications, work was underway that would reverse this trend. One key development was an early bifurcation of functional language designs at a fairly fundamental level. The Lisp family of languages are untyped and dynamic. In the modern world, the lack of typing might seem egregiously unmaintainable, but in comparison to C these languages allow for a kind of dynamic meta-programming that more than made up for it. Programmers enjoyed a certain kind of productivity because they could "go meta" – that is, writing programs to write programs, even dynamically modifying them on the fly – in a uniform manner. This feature has become mainstream, as found in Ruby or even Java's Reflection API, precisely because it is so useful. Unfortunately, the productivity gains of meta-programming available in Lisp and its derivatives were not enough to offset the performance shortfalls at the time.

However, a statically typed branch of functional programming began to have traction in certain academic circles with the development of the ML family of languages, which today includes OCaml, the language that can be considered the direct ancestor of both Scala and F#. One of the first developments in that line of investigation was the recognition that data description came in not just one but *two* flavors: types and patterns, which were recognized as dual. Types tell the program how data is built up from its components while patterns tell a program how to take data apart in terms of its components. The crucial point is that these two notions are just two sides of the same coin and can be made to work together and support each other in the structuring and execution of programs. In this sense, the development, while enriching the language features, reduces the concept complexity. Both language designer and programmer think in terms of one thing, data description, while recognizing that such descriptions have uses for structuring and de-structuring data. These are the origins of elements in Scala's design, like `case` classes and the `match` construct.

The ML family of languages also gave us the first robust instantiations of parametric polymorphism. The widespread adoption of generics in C/C++, Java, and C# say much more about the importance of this feature than any impoverished account I can conjure here. Again, though, the moral of the story is that this represents a significant reduction in complexity. For example, you can separate common container patterns from the types they contain, allowing for programming that is considerably DRYer. [1]

Still these languages suffered when it came to a compelling and uniform treatment of side-effecting computations. That all changed with Haskell. In the mid-80s, a young researcher named Eugenio Moggi observed that an idea previously discovered in a then-obscure branch of mathematics (called *category theory*) offered a way to structure functional programs to allow them to deal with side-effecting computations in a uniform and compelling manner. Essentially, the notion of a *monad* (as it was called in the category theory literature) provided a language-level abstraction for structuring side-effecting computations in a functional setting. In today's parlance, Moggi found a domain-specific language (DSL) for organizing side-effecting computations in an ambient (or hosting) functional language. Once Moggi made this discovery, another researcher named Phil Wadler realized that this DSL had a couple of different presentations (different concrete syntaxes for the same underlying abstract syntax) that were almost immediately understandable by the average programmer. One presentation, called *comprehensions* (after its counterpart in set theory), could be understood directly in terms of a familiar construct `SELECT ... FROM ... WHERE ...`; while the other, dubbed `do` notation by the Haskell community, provided operations that behaved remarkably like sequencing and assignment. Haskell offers syntactic sugar to

[1] DRY is the pop culture term for "Don't repeat yourself."

support the latter while the former has been adopted in both XQuery's `FLWOR` expressions and Microsoft's LINQ.

Of course, to say that Haskell offers syntactic sugar hides the true nature of how the language supports monads. Three elements actually come together to make this work. First, expressing the pattern requires support for parametric polymorphism, generics-style type abstraction. Second, another mechanism, Haskell's `typeclass` mechanism (the Haskell equivalent to Scala's `trait`) is required to make the pattern itself polymorphic. Then there is the `do` notation itself and the syntax-driven translation from that to Haskell's core syntax. Together, these features allow the compiler to work out which interpretations of sequencing, assignment, and return are in play – without type annotations. The design simplicity sometimes makes it difficult to appreciate the subtlety, or the impact it has had on modern language design, but this was the blueprint for the way Scala's `for` comprehensions work.

With this structuring technique (and others like it) in hand, you can easily spot (often by type analysis alone) situations where programs can be rewritten to equivalent programs that execute much better on existing hardware. This is a central benefit of the monad abstraction, and these powerful abstractions are among the primary reasons why functional programming has made such progress in the area of performance. As an example, not only can you retarget LINQ-based expressions to different storage models (from relational database to XML database), but you can rewrite them to execute in a data-parallel fashion. Results of this type suggest that we are just beginning to understand the kinds of performance optimizations available from using monadic programming structuring techniques.

It turns out that side-effecting computations are right at the nub of strategies for using concurrency as a means to scale up performance and availability. In some sense, a side effect really represents an interaction between two systems (one of which is viewed as "on the side" of the other; for example, at the boundary of some central locus of computation). Such an interaction, say between a program in memory and the I/O subsystem, entails some sort of synchronization. Synchronization constraints are the central concerns in using concurrency to scale up both performance and availability. Analogies to traffic illustrate the point. It's easy to see the difference in traffic flow if two major thoroughfares can run side-by-side versus when they intersect and have to use some synchronization mechanism, like a traffic light or a stop sign. In a concurrent world, functional purity, which insists on no side effects (i.e., no synchronization), is no longer an academic exercise with unrealistic performance characteristics. Instead, computation that can proceed without synchronization, including side-effect-free code, becomes the gold standard. Of course, it is not realistic to expect computation never to synchronize, but now this is seen in a different light, and is perhaps the most stark way to illustrate the promise of monadic structuring techniques in the concurrent world programmers find themselves. They allow us to write in a language that is at least notionally familiar to most programmers and yet analyze what's written and retarget it for the concurrent setting.

In summary, functional language design improved in terms of the following:

- Extending the underlying mechanism in how types work on data, exposing the duality between type conformance and pattern-matching
- Extending the reach of types to parametric polymorphism
- Providing a framework for cleaning up the semantics of side-effecting or stateful computations and generalizing them

Combined with the inherent simplicity of functional language design and its compositional nature, we have the making of a revolution in complexity management. This is the real dominating trend in the industry. Once Java was within 1.4 times the speed of C/C++, the game was over because Java significantly reduced application development complexity. This increased both productivity and manageability. Likewise, the complexity of Java[2]

1.2 Where are we going?

With a preamble like that, it doesn't take much to guess where all this is heading. More and more, we are looking at trends that lead toward more functional and functionally based web applications. We need not look to the growing popularity of cutting-edge frameworks like Lift to see this trend.

[2]Here, we are not picking on Java, specifically. The same could be said of C# development, but Java development on Internet-based applications especially has become nearly prohibitive. Functional languages – especially languages like Scala that run on the JVM, have excellent interoperability with the extensive Java legacy, and have performance on par with Java – are poised to do to Java what Java did to C/C++.

We must count both JavaScript (with its origins in Self) and Rails among the functionally influenced.

A functional web

Because plenty of excellent functional web frameworks already exist in the open source community, we won't aim to build another. Rather, we will supply a set of design patterns that will work with most – in fact, are already implicitly at work in many – but that when used correctly will reduce complexity.

Specifically, we will look at the organization of a web application pipeline, from the stream of HTTP requests through the application logic to the store and back. We will see how in each case judicious use of the monadic design pattern provides for significant leverage in structuring code, making it both simpler, more maintainable, and more robust in the face of change. In addition, we will provide a more detailed view of the abstractions underlying these design patterns, introducing the reader who is less experienced with some of the theoretical background to this toolset and challenging the more experienced with new ways of thinking about and deploying it.

To that end, we will look at the following:

- Processing HTTP streams using *delimited* continuations to allow for a sophisticated state management Chapter 3
- Parser combinators for parsing HTTP requests and higher-level application protocols, using HTTP as a transport Chapter 4
- Application domain model as an abstract syntax Chapter 5
- Zippers as a means of automatically generating navigation Chapter 6
- Collections and containers in memory Chapter 7
- Storage, including a new way to approach query and search Chapter 8

In each processing step, there is an underlying organization to the computation that solves the problem. In each case, we find an instance of the monadic design pattern. It remains to be seen whether this apparent universal applicability is an instance of finding a hammer that turns everything it encounters into nails or that structuring computation in terms of monads has a genuine depth. However, at this early stage, we can say that object-oriented design patterns were certainly proposed for each of these situations and many others. It was commonly held that such techniques were not merely universally applicable, but of genuine utility in every application domain. The failure of object-oriented design methods to make good on these claims might be an argument for caution but sober assessment of the situation gives cause for hope.

Unlike the notion monad, objects began as folk tradition. It was many years into proposals for object-oriented design methods before there were commonly accepted formal or mathematical accounts. By contrast, monads began as a mathematical entity. Sequestered away in category theory, the idea was one of a whole zoology of generalizations about common mathematical entities. It took some time to understand that both set comprehensions and algebraic data types were instances monads and that the former was a universal language for the notion. It took even more time to see the application to structuring computations. Progress was slow and steady and built from a solid foundation. This gave the notion an unprecedented level of quality assurance testing. The category theoretic definition is nearly 50 years old. If we include the investigation of set comprehensions as a part of the QA process, we add another 100 years. If we include the 40 years of vigorous use of relational databases and the `SELECT-FROM-WHERE` construct in the industry, we see that this was hardly just an academic exercise.

Perhaps more important is the fact that while object-oriented techniques, as realized in mainstream language designs,[3] ultimately failed to be compositional in any useful way – inheritance, in fact, being positively at odds with concurrent composition – the notion of monad is actually an attempt to capture the meaning of composition. As we will see in the upcoming sections, this abstraction defines a powerful notion of parametric composition. This is crucial because in the real world composition is the primary means to scaling – both in the sense of performance and complexity. As pragmatic engineers, we manage complexity of scale by building larger systems out of smaller ones. As pragmatic engineers, we understand that each time components are required to interface or synchronize we have the potential to introduce performance concerns. The parametric form of composition encapsulated in the notion of monad gives us a language for talking about both kinds of scal-

[3]To be clear, message-passing and delegation are certainly compositional. Very few mainstream languages support these concepts directly

ing and connecting the two ideas. It provides a language for talking about the interplay between the compositions of structure and control flow. It encapsulates stateful computation and data structure. In this sense, the notion of monad is poised to be the rational reconstruction of the notion of object. Telling this story was my motivation for writing this book.

DSL-based design

It has become buzzword *du jour* to talk about DSL-based design. So much so that it's becoming difficult to understand what the term means. In the functional setting, the meaning is quite clear and has become considerably clearer since the writing of *Structure and Interpretation of Computer Programs* by Hal Abelson, et al. (MIT Press, 1985) – one of the seminal texts of functional programming and one of the first to pioneer the idea of DSL-based design. In a typed functional setting, designing a collection of types tailor-made to model and address the operations of some domain is effectively the design of an abstract language syntax for computing over the domain.

To see why this must be so, let's begin from the basics. Informally, DSL-based design means we express our design in terms of a mini-language, tailor-made for our application domain. When push comes to shove, though, if we want to know what DSL-based design means in practical terms, eventually we have to ask what goes into a language specification. The commonly received wisdom is that a language is comprised of a *syntax* and a *semantics*. The syntax carries the structure of the language expressions while the semantics say how to evaluate those expressions to achieve a result – typically either to derive a meaning for the expression (such as *this expression denotes that value*) or perform an action or computation indicated by the expression (such as *print this string on the console*). Focusing, for the moment, on syntax as the more concrete of the two elements, we note that syntax is governed by *grammar*. Syntax is governed by grammar, whether we're building a concrete syntax, like the ASCII strings you type to communicate Scala expressions to the compiler or building an abstract syntax, like the expression trees of LINQ.

What we really want to call out in this discussion is that a collection of types forming a model of some domain is actually a grammar for an abstract syntax. This is most readily seen by comparing the core of the type definition language of modern functional languages with something like EBNF, the most prevalent language for defining context-free grammars. At their heart, the two structures are nearly the same. When defining a grammar, you are defining a collection of types that model some domain and vice versa. This is blindingly obvious in Haskell, and is the essence of techniques like the application of two-level type decomposition to model grammars. Moreover, while a little harder to see in Scala it is still there. It is in this sense that typed functional languages like Scala are well suited for DSL-based design. To the extent that the use of Scala relies on the functional core of the language (not the object-oriented bits), virtually every domain model is already a kind of DSL in that its types define a kind of abstract syntax.

Taking this idea a step further, in most cases such collections of types are actually representable as a monad. Monads effectively encapsulate the notion of an algebra, which in this context is a category theorist's way of saying a certain kind of collection of types. If you are at all familiar with parser combinators and perhaps have heard that these too are facilitated with monadic composition then the suggestion that there is a deeper link between parsing, grammars, types, and monads might make some sense. On the other hand, if this seems a little too abstract, I will make it much more concrete in the following sections. For now, we are simply planting the seed of the idea that monads are not just for structuring side-effecting computations.

1.3 How will we get there?

Equipped with a sense of where we are and where we are planning to go what remains is to discuss how we'll get there. In the words of a good friend, "how we get there *is* where we are going." So, much of how we get there in this book is based in service, practicality and humility – while at the same time remembering that a good grasp of the conceptual basis underlying what it is we're up to *is* practical. We know that it takes a lot of different kinds of expertise to pull together good webapps. We know that there is more to know about the theory of computation than anyone person can genuinely understand. Yet, by bringing these two together we aim to serve real people who really want to get things done with computers, to unburden them by helping to reduce the perceived and actual complexity of writing web-based applications.

Leading by example

The principal technique throughout this book is leading by example. What this means in this case is that the ideas are presented primarily in terms of a coherent collection of examples, rendered as Scala code, that work together to do something. Namely, these examples function together to provide a prototypical web-based application with a feature set that resonates with what application developers are building today and contemplating building tomorrow.

Let's illustrate this in more detail by telling a story. We imagine a cloud-based editor for a simple programming language, not unlike Mozilla's `bespin`. A user can register with the service, (see Figure 1.1), and then create an application project that allows them to do the following:

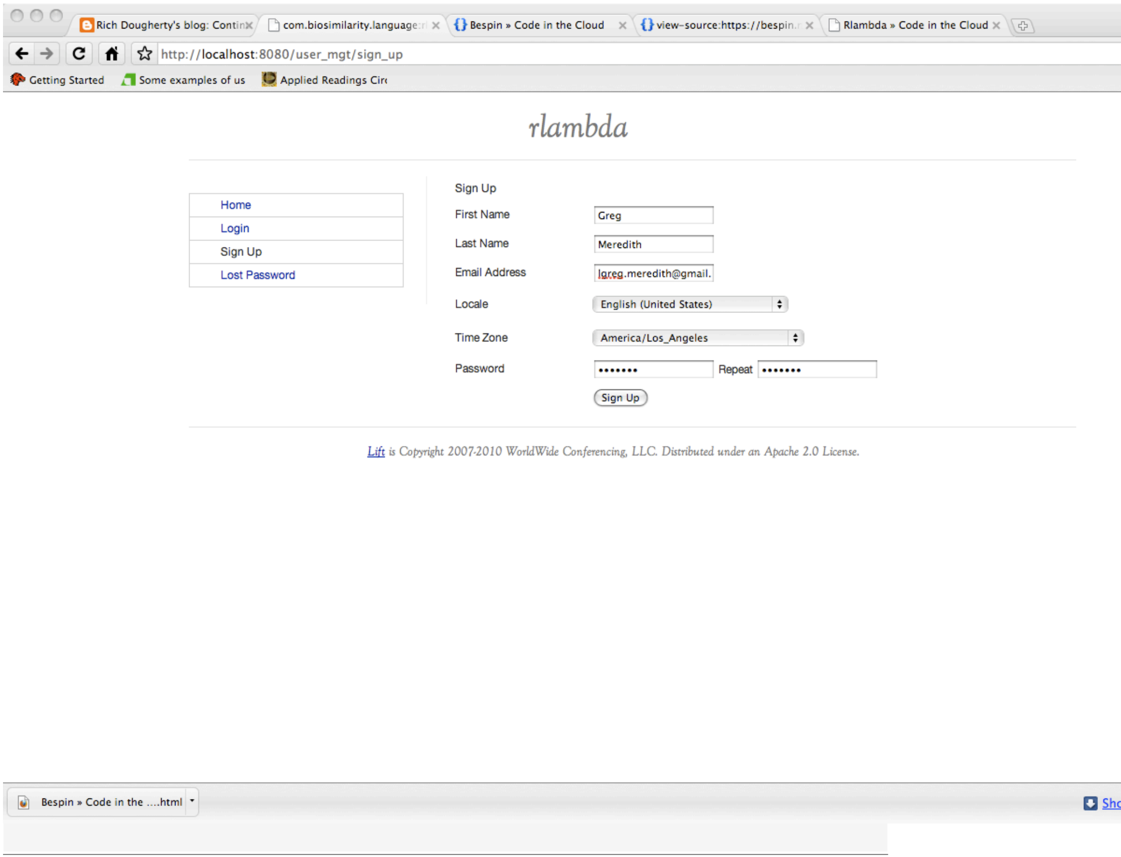
- Write code in a structured editor that understands the language
- Manage files in the application project
- Compile the application
- Run the application

These core capabilities wrap around our toy programming language in much the same way a modern Integrated Development Environment (IDE) might wrap around development in a more robust, full-featured language. Hence, we want the application capabilities to be partially driven from the specification of our toy language. For example, if we support some syntax highlighting or syntax validation on the client, we want that to be driven from that language spec to the extent that changes to the language spec ought to result in changes to the behavior of the highlighting and validation. Thus, at the center of our application is the specification of our toy language.

Our toy language

Abstract syntax Fittingly for a book about Scala, we'll use the λ-calculus as our toy language.[4] The core *abstract* syntax of the λ-calculus is given by

[4] A word to the wise: Even if you are an old hand at programming language semantics, and even if you know the λ-calculus like the back of your hand, you are likely to be surprised by some of the things you see in the next few sections. Just to make sure that everyone gets a chance to look at the formalism as if it were brand new, I've thrown in a few recent theoretical developments, so watch out!


Rich Dougherty's blog: Contin
com.biosimilarity.language.r
Bespin » Code in the Cloud
view-source:https://bespin.r
Rlambda » Code in the Cloud
Getting Started
Some examples of us
Applied Readings Circ
Home
Login
Sign Up
Lost Password
Sign Up
Bespin » Code in the ....html
Sho

http://localhost:8080/user_mgt/sign_up

rlambda

Sign Up

First Name	Greg
Last Name	Meredith
Email Address	lgreg.meredith@gmail.
Locale	English (United States)
Time Zone	America/Los_Angeles
Password	••••••• Repeat •••••••

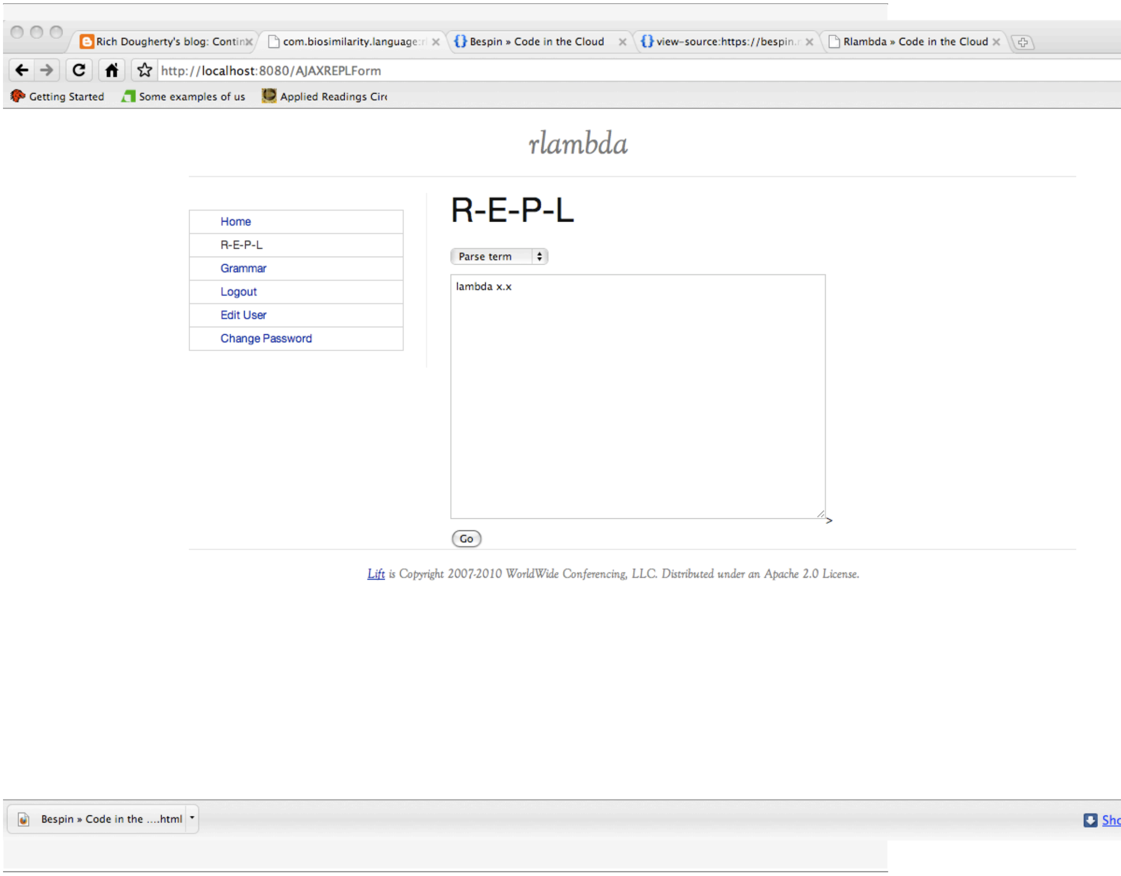
Lift is Copyright 2007-2010 WorldWide Conferencing, LLC. Distributed under an Apache 2.0 License.

Figure 1.1 · Example sign-up page

the following EBNF grammar.

EXPRESSION	MENTION	ABSTRACTION	APPLICATION
$M,N ::=$	x	$\mid \lambda x.M$	$\mid MN$

Informally, this is really a language of pure variable management. For example, if the expression M mentions x, then $\lambda x.M$ turns x into a variable in M and provides a means to substitute values into M, via application. Thus, $(\lambda x.M)N$ will result in a new term, sometimes written $M[N/x]$, in which every occurrence of x has been replaced by an occurrence of N. Thus, $(\lambda x.x)M$ yields M, illustrating the implementation in the λ-calculus of the identity


Rich Dougherty's blog: Contin
com.biosimilarity.language
Bespin » Code in the Cloud
view-source:https://bespin.
Rlambda » Code in the Cloud
Getting Started
Some examples of us
Applied Readings Circ
rlambda
Home
R-E-P-L
Grammar
Logout
Edit User
Change Password
R-E-P-L
Go
Bespin » Code in the ....html

http://localhost:8080/AJAXREPLForm

Parse term

```
lambda x.x
```

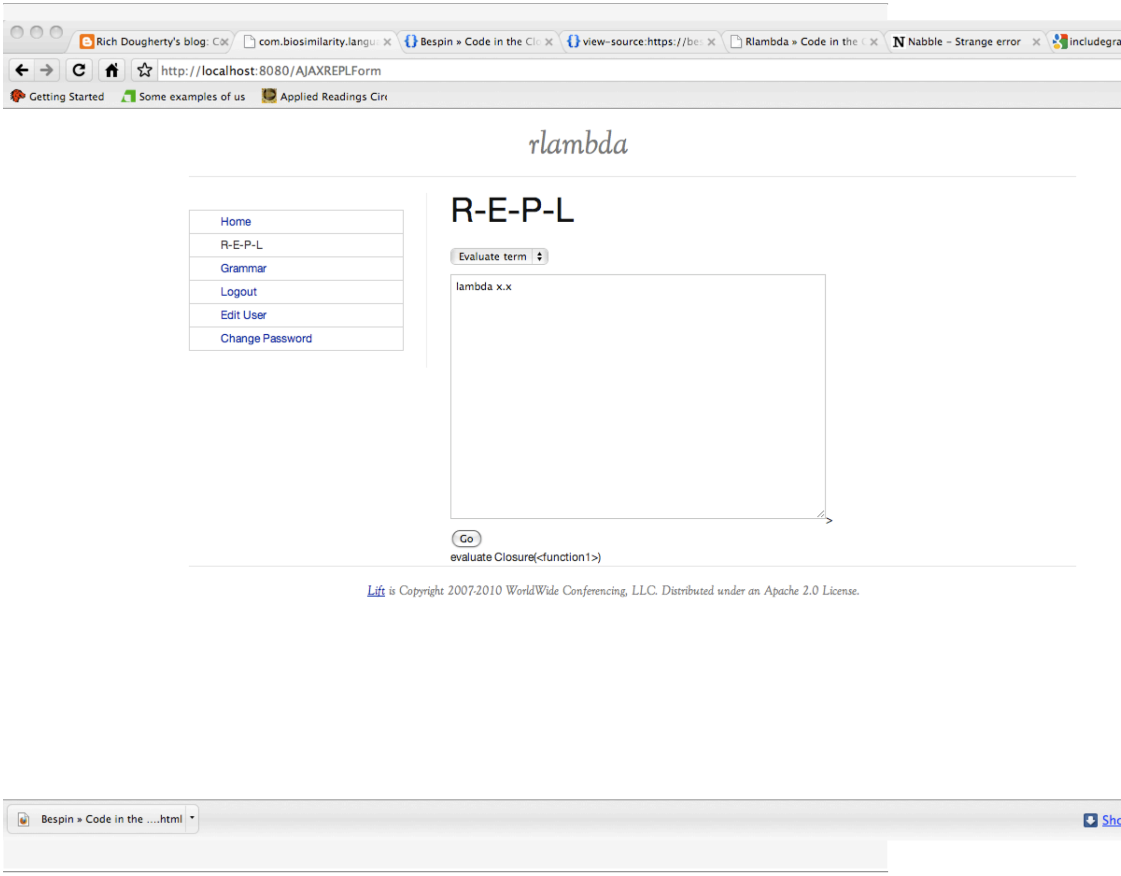
Lift is Copyright 2007-2010 WorldWide Conferencing, LLC. Distributed under an Apache 2.0 License.

Figure 1.2 · Example REPL page

function. It turns out to be quite remarkable what you can do with pure variable management.

Concrete syntax We'll wrap this up in concrete syntax.

$$\begin{array}{llll}
\text{EXPRESSION} & \text{MENTION} & \text{ABSTRACTION} & \text{APPLICATION} \\
M,N ::= & x & \mid (x_1, \ldots, x_k) \Rightarrow M & \mid M(N_1, \ldots, N_k) \\
\text{LET} & \text{SEQ} & \text{GROUP} & \\
\mid \texttt{val}\ x = M ; N & \mid M ; N & \mid \{\ M\ \} &
\end{array}$$


Rich Dougherty's blog: C
com.biosimilarity.langu
Bespin » Code in the Cl
view-source:https://be
Rlambda » Code in the
Nabble - Strange error
includegra

http://localhost:8080/AJAXREPLForm

Getting Started
Some examples of us
Applied Readings Circ

rlambda

R-E-P-L

Home
R-E-P-L
Grammar
Logout
Edit User
Change Password

Evaluate term

```
lambda x.x
```

Go

```
evaluate Closure(<function1>)
```

Lift is Copyright 2007-2010 WorldWide Conferencing, LLC. Distributed under an Apache 2.0 License.

Bespin » Code in the ....html

Figure 1.3 · Example evaluation result page

It doesn't take much squinting to see that this looks a lot like a subset of Scala, and that's because – of course! – functional languages like Scala all share a common core, which is essentially the λ-calculus. Once you familiarize yourself with the λ-calculus as a design pattern you'll see it poking out everywhere: in Clojure and OCaml and F# and Scala. In fact, as we'll see later, just about any DSL you design that needs a notion of variables could do worse than simply to crib from this existing and well-understood design pattern.

We'll also extend this core language with a little arithmetic and look at how language extensions like that raise important questions about the compositionality of DSL-based design vis-a-vis the composition of monads; more on that in Chapter 4.

Code editor

Project editor

Advanced features

Chapter map

Taking a step back from the technical discussion, let's review what we'll cover and how we'll cover it. Essentially, the book is organized to follow the processing of HTTP requests from the browser through the server and application code out to the store and back.

- Chapter 2, "Toolbox," introduces terminology, notation, and concepts necessary for the rest of the book.
- Chapter 3, "An I/O Monad for HTTP Streams," looks at the organization of an HTTP server.
- Chapter 4, "Parsing Requests, Monadically," investigates parsing the transport and application-level requests.
- Chapter 5, "The Domain Model as Abstract Syntax," focuses on the application domain model.
- Chapter 6, "Zippers and Contexts and URIs, Oh My!," addresses at the navigation model.
- Chapter 7, "A Review of Collections as Monads," reviews collections.
- Chapter 8, "Domain Model, Storage, and State," looks at the storage model.
- Chapter 9, "Putting It All Together," investigates application deployment.
- Chapter 10, "The Semantic Web," addresses new foundations for semantic query.

Understanding that there are different readers with different interests and different levels of experience, the chapters are organized into two primary sections: an "on the way in" section and an "on the return" section. The former is a more pragmatic, code-first view of the material. The latter elaborates this material with relevant discussion from the theory on which the solutions are based. In many instances, in these sections, I will introduce new ideas that expand or challenge the current theoretical account, and so both kinds of readers are invited to look upon the discussion and proposals with a critical eye. The reader less experienced in this sort of thinking is especially invited to engage as fresh eyes have a way of seeing things differently.

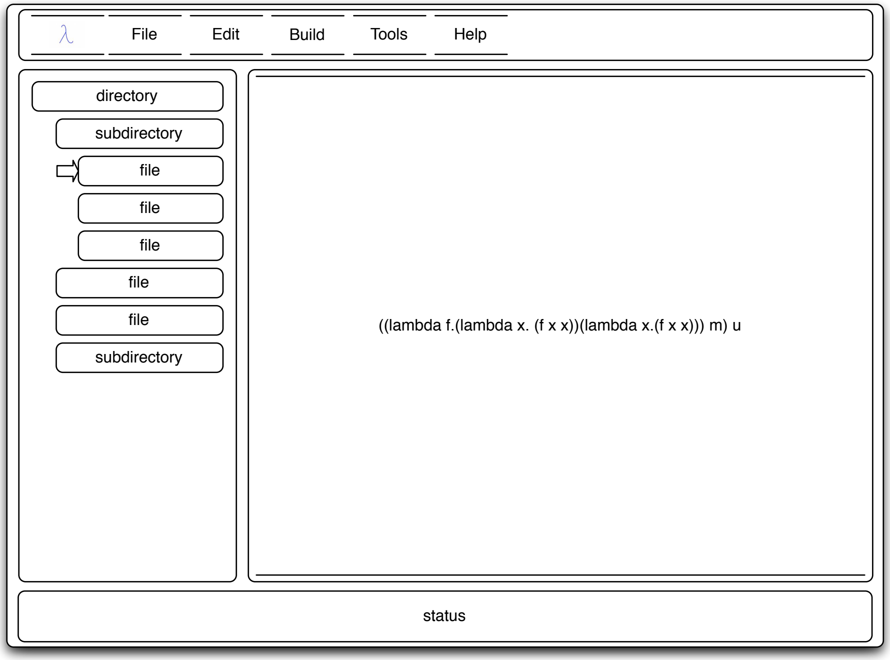


Figure 1.4 · Project and code editor

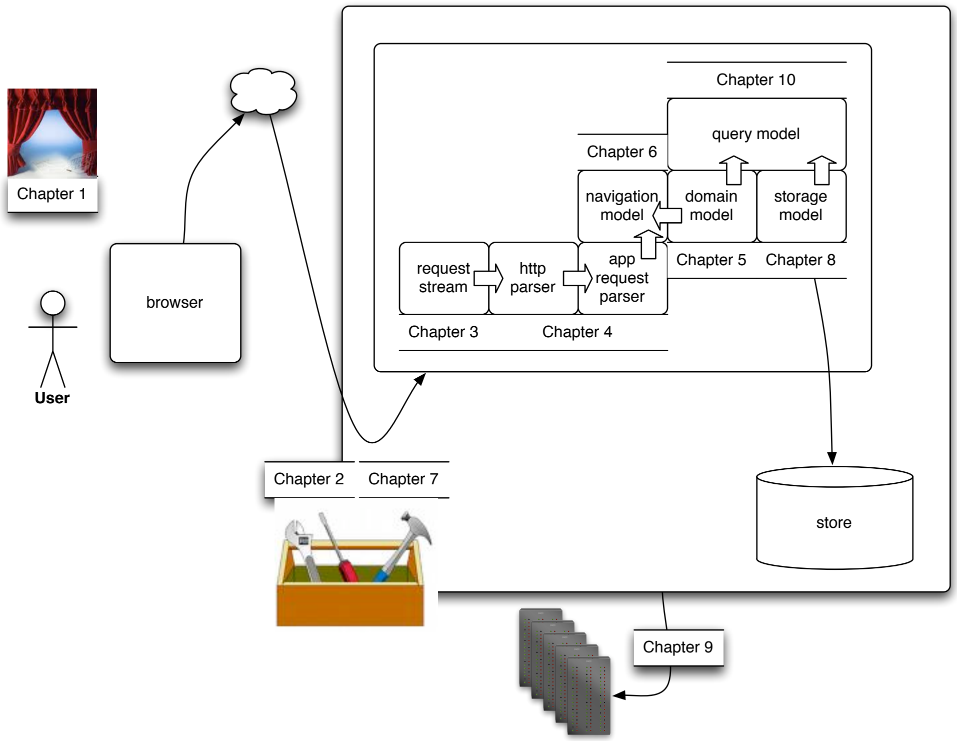


Figure 1.5 · Chapter map