Pro Scala: Monadic Design Patterns for the Web

L.G. Meredith

Contents

Contents **i**

Preface **1**

1 Motivation and Background **3**

1.1 Where are we 4

1.1.1 The concurrency squeeze: from the hardware up, from the web down 4

1.1.2 Ubiquity of robust, high-performance virtual machines 5

1.1.3 Advances in functional programming, monads and the awkward squad 6

1.2 Where are we going 10

1.2.1 A functional web 10

1.2.2 DSL-based design 12

1.3 How are we going to get there 13

1.3.1 Leading by example 13

2 Toolbox **25**

2.1 Introduction to notation and terminology 25

2.2 Introduction to core design patterns 25

2.3 Variations in presentation 25

2.3.1 A little history 25

2.3.2 A little more history 28

3 An IO-monad for http streams **33**
3.1 Code first, questions later 33
3.1.1 An HTTP-request processor 40
3.1.2 What we did 40
3.2 Synchrony, asynchrony and buffering 40
3.3 State, statelessness and continuations 40

4 Parsing requests, monadically **41**
4.1 Obligatory parsing monad 41
4.2 Your parser combinators are showing 41
4.3 EBNF and why higher levels of abstraction are better 41

5 The domain model as abstract syntax **43**
5.1 Our abstract syntax 43
5.2 Our application domain model 43
5.3 A transform pipeline 43

6 Zippers and contexts and URI's, oh my! **45**
6.1 Zippers are not just for Bruno anymore 45
6.2 Constructing contexts and zippers from data types 45
6.3 Mapping URIs to zipper-based paths and back 45

7 A review of collections as monads **47**
7.1 Monad as container 47
7.2 Monads and take-out 47

8 Domain model, storage and state **49**
8.1 Mapping our domain model to storage 49
8.2 Storage and language-integrated query 49
8.3 Continuations revisited 49

9 Putting it all together **51**

9.1 Our web application end-to-end 51
9.2 Deploying our application 51
9.2.1 Why we are not deploying on GAE 51
9.3 From one web application to web framework 51

10 The semantic web **53**
10.1 How our web framework enables different kinds of application queries 53
10.2 Searching for programs 53

List of Figures

List of Tables

Preface

The book you hold in your hands, Dear Reader, is not at all what you expected...

Chapter 1

Motivation and Background

Where are we; how did we get here; and where are we going?

With apologies to Italo Calvino You've just picked up the new book by Greg Meredith, Pro Scala. Perhaps you've heard about it on one of the mailing lists or seen it advertised on the `Scala` site or at Amazon. You're wondering if it's for you. Maybe you've been programming in functional languages or even `Scala` for as long as you can remember. Or maybe you've been a professional programmer for quite some time. Or maybe you're a manager of programmers, now and you're trying to stay abreast of the latest technology. Or, maybe you're a futurologist who looks at technology trends to get a sense of where things are heading. Whoever you are, if you're like most people, this book is going to make a lot more sense to you if you've already got about five to ten thousand hours of either `Scala` or some other functional language programming under your belt. There may be nuggets in here that provide some useful insights for people with a different kind of experience; and, of course, there are those who just take to the ideas without needing to put in the same number of hours; but, for most, that's probably the simplest gauge of whether this book is going to make sense to you at first reading.

On the other hand, just because you've got that sort of experience under your belt still doesn't mean this book is for you. Maybe you're just looking for a few tips and tricks to make `Scala` do what you want for the program you're writing right now. Or maybe you've got a nasty perf issue you want to address and are looking here for a resolution. If that's the case, then maybe this book isn't for you because this book is really about a point of view, a way of looking at programming and computation. In some sense this book is all about programming and complexity management because that's really the issue that the professional programmer is up against, today. On average the modern programmer building an Internet-based

application is dealing with no less than a dozen technologies. They are attempting to build applications with nearly continuous operation, 24x7 availability servicing 100's to 1000's of concurrent requests. They are overwhelmed by complexity. What the professional programmer really needs are tools for complexity management. The design patterns expressed in this book have been developed for nearly *fifty* years to address exactly those concerns. Since `Scala` isn't nearly fifty years old you can guess that they have origins in older technologies, but `Scala`, it turns out, is an ideal framework in which both to realize them and to talk about their ins and outs and pros and cons. However, since they don't originate in `Scala`, you can also guess that they have some significant applicability to the other eleven technologies the modern professional programmer is juggling.

1.1 Where are we

1.1.1 The concurrency squeeze: from the hardware up, from the web down

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 on-coming storm are definitely in our rear view mirror. Today's developer knows that if her program is commercially interesting at all then it needs to be web-accessible on a 24x7 basis; and if it's going to be commercially significant it will need to support at least 100's if not thousands of concurrent accesses to its features and functions. Her application is most likely hosted by some commercial outfit, a Joyent or an EngineYard or an Amazon EC3 or ... who are deploying her code over multiple servers each of which is in turn multi-processor with multiple cores. This means that from the hardware up and from the web down today's intrepid developer is dealing with parallelism, concurrency and distribution.

Unfortunately, the methods available in in mainstream programming languages of dealing with these different aspects of simultaneous execution are not up to the task of supporting development at this scale. The core issue is complexity. The modern application developer is faced with a huge 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, she's thinking hard about *how* to partition her data and whether or not this "eventual consistency" thing is going to liberate her or bring on a new host of programming nightmares. By comparison threads packages seem like quaint relics from a time

when concurrent programming was a little 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 that much of a radical departure from mainstream programming like Java. On the other it offers simple, uniform model that introduces a number of key features that considerably improve productivity and maintainability. Java brought the C/C++ programmer several steps closer to a functional paradigm, introducing garbage collection, 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 model of computation, like the actor model or the process calculi, in which the notion of execution that is fundamentally concurrent. That said, it meshes nicely with a variety of 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 the structuring techniques of monads make models such as software transactional memory or data flow parallelism quite easy to integrate, while pattern-matching additionally makes message-passing style easier to incorporate.

1.1.2 Ubiquity of robust, high-performance virtual machines

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 actually the dominant choice for many applications. This has two effects that are playing themselves out in terms of industry trends. Firstly, it provides some level of insulation between changes in hardware design (from single core per die to multi-core, 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 much 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. One 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 best use of memory that percolate all through application code. Secondly, it completely changes the landscape for language development. By provid-

ing 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. It should not escape notice that all of the languages in that list are either functional languages, object-functional languages, and the majority of the proposals coming out are either functional, object-functional or heavily influenced by functional language design concepts.

1.1.3 Advances in functional programming, monads and the awkward squad

Perhaps chief among the reasons for the popularity of developing a language design based on functional concepts 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 adoption of functional language design is that the core model is *compositional*. Enrichment of the execution semantics amounts to enrichment of the components of the semantics. Much more can be said about this, but needs to be deferred to a point where more context has been developed. Deep simplicity and compositionality are properties and principles that take quite some time to appreciate while some of the practical reasons that recent language design proposals have been so heavily influenced by functional language design principles is easily understood by even the most impatient of pragmatic of programmers: functional language design has made significant and demonstrable progress addressing performance issues that plagued it at the beginning. Moreover, these developments have significant applicability to the situation related to concurrent execution that the modern programmer finds herself now.

Since the mid '80's when `Lisp` and it's progeny were thrown out of the industry for performance failures a lot of excellent work has gone on that has rectified 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 throwing out functional languages,

except for niche applications, work was going on that would reverse this trend. One of the key developments in this 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 by comparison to `C` it was more than made up for by the kind of dynamic meta-programming that these languages made possible. Programmers enjoyed a certain kind of productivity because they could "go meta" – writing programs to write programs (even dynamically modify them on the fly) – in a uniform manner. This sort of feature has become mainstream, as found in `Ruby` or even `Java`'s reflection API, precisely because it is so extremely useful. Unfortunately, the productivity gains of meta-programming available in `Lisp` and it's derivatives were not enough to offset the performance shortfalls at the time.

There was, however, a statically typed branch of functional programming that 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 very first developments in that line of investigation was the recognition that data description came in not just one but *two* flavors: types and *patterns*. The two flavors, it was recognized, are 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 an enrichment of the language features – is a reduction in the complexity of concepts. Both language designer and programmer think in terms of one thing, description of data, 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 class**es 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 the author can conjure here. Again, though, the moral of the story is that this represents a significant reduction in complexity. Common container patterns, for example, can be separated from the types they contain, allowing for programming that is considerably DRYer.

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-80's a young researcher by the name of 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 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, he found a domain specific language, a DSL, for organizing side-effecting computations in an ambient (or hosting) functional language. Once Moggi made this discovery another researcher, 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 it's counter part in set theory), could be understood directly in terms of a very 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 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 monads are supported in the language. There are actually three elements that come together to make this work. First, expressing the pattern at all 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. Taken together, these features allow the compiler to work out which interpretations of sequencing, assignment and return are in play – without type annotations. The simplicity of the design 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 it becomes a lot easier to spot (often by type analysis alone) situations where programs can be rewritten to equivalent programs that execute much better on existing hardware. This is one of the central benefits of the monad abstraction, and these sorts of 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 `LINQ`-based expressions be retargeted to different storage models (from relational database to `XML` database) they can be rewritten to execute in a data parallel fashion. Results of this type suggest that we are really just at the beginning of understanding the kinds of performance optimizations available through the use of 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, i.e. 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. So, 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 which 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

- extending the underlying mechanism at work 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

Taken together with the inherent simplicity of functional language design and its compositional nature we have the makings of a revolution in complexity management. This is the real dominating trend in the industry. Once `Java` was within 1.4X the speed of `C/C++` the game was over because `Java` offered such a significant reduction in application development complexity which turned into gains in both productivity and manageability. Likewise, the complexity of `Java` [1] development and especially `Java` development on Internet-based applications has become nearly prohibitive. Functional languages, especially languages like `Scala` which run on the `JVM` and 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++`.

[1] and here we are not picking on `Java`, specifically, the same could be said of `C#`

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. Both Javascript (with it's origins in Self) and Rails must be counted amongst the functionally influenced.

1.2.1 A functional web

Because there are already plenty of excellent functional web frameworks in the open source community our aim is not to build another. Rather our aim is to 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 the pipeline of a web-application from the pipeline 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.

To that end we will be looking at

- processing HTTP-streams using presence of *delimited* continuations to allow for a sophisticated state management
- parser combinators for parsing HTTP-requests and higher-level application protocols using HTTP as a transport
- application domain model as an abstract syntax
- zippers as a means of automatically generating navigation
- collections and containers in memory
- storage, including a new way to approach query and search

In each case there is an underlying organization to the computation that solves the problem. In each case we find an instance of the monadic design pattern. 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 remains to be seen. What can be said even at this early stage of the game is 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 domain of application. The failure of object-oriented design methods to make good on these claims might be an argument for caution. Sober assessment of the situation, however, 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 of 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 fifty years old. If we include the investigation of set comprehensions as a part of the QA process we add another one hundred years. If we include the forty 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 importantly than any of those is the fact that while object-oriented techniques *as realized in mainstream language designs* [2] 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, it defines an 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 in the sense of 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 for introducing performance concerns. The parametric form of composition encapsulated in the notion of monad gives us a language for talking about both kinds of scaling and connecting the two ideas. It provides a language for talking about the interplay between the composition of structure and the composition of the flow of control. It encapsulates stateful computation. It encapsulates 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.

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

1.2.2 DSL-based design

It has become buzz-word du jour to talk about DSL-based design. So much so that it's becoming hard to understand what the term means. In the functional setting the meaning is really quite clear and since the writing of the Structure and Interpretation of Computer Programs (one of the seminal texts of functional programming and one of the first to pioneer the idea of DSL-based design) the meaning has gotten considerably clearer. In a typed functional setting the design of a collection of types tailor-made to model and address the operations of some domain is the basis is effectively the design of an abstract syntax of a language 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 little 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 the specification of a language. The commonly received wisdom is that a language is comprised of a *syntax* and a *semantics*. The syntax carries the structure of the expressions of the language while the semantics says 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*. Whether we're building a concrete syntax, like the `ASCII` strings one types to communicate `Scala` expressions to the compiler or building an abstract syntax, like the expression trees of `LINQ`, syntax is governed by grammar.

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 one is defining a grammar one is 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 very 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 it's 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 it will be made 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 are we going to get there

1.3.1 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 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 and then create an application project which allows them

- to write code in a structured editor that understands the language;
- manage files in the application project;
- compile the application;
- run the application

Our toy language

Abstract syntax For our example we'll need a toy language. Fittingly for a book about `Scala` we'll use the λ-calculus as our toy language. The core *abstract* syntax of the lambda calculus is given by the following *EBNF* grammar.

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

A simple-minded representation At a syntactic level this has a direct representation as the following `Scala` code.

```
trait Expressions {
  type Nominal
  abstract class Expression

  case class Mention( reference : Nominal )
     extends Expression

  case class Abstraction(
     formals : List[Nominal],
     body : Expression
  ) extends Expression

  case class Application(
     operation : Expression,
     actuals : List[Expression]
  ) extends Expression
}
```

In this representation each *syntactic category*, EXPRESSION, MENTION, ABSTRACTION and APPLICATION, is represented by a **trait** or **case class**. EXPRESSION's are **trait**'s because they are pure placeholders. The other categories elaborate the syntactic form, and the elaboration is matched by the **case class** structure. Thus, for example, an ABSTRACTION is modeled by an instance of the **case class** called Abstraction having members formal for the formal parameter of the abstraction, and body for the λ-term under the abstraction that might make use of the parameter. Similarly, an APPLICATION is modeled by an instance of the **case class** of the same name having members operation for the expression that will be applied to the actual parameter called (not surprisingly) actual.

Type parametrization and quotation One key aspect of this representation is that we acknowledge that the abstract syntax is strangely silent on what the *terminals* are. It doesn't actually say what x's are. Often implementations of the λ-calculus will make some choice, such as Strings or Integers or some other representation. With `Scala`'s type parametrization we can defer this choice. In

fact, to foreshadow some of what's to come, we illustrate that we never actually have to go outside of the basic grammar definition to come up with a supply of identifiers.

In the code above we have deferred the choice of identifier. In the code below we provide several different kinds of identifiers (the term of art in this context is "name"), but defer the notion of an expression by the same trick used to defer the choice of identifiers.

```
trait Nominals {
  type Term
  abstract class Name
  case class Transcription( expression : Term )
     extends Name
  case class StringLiteral( str : String )
     extends Name
  case class DeBruijn( outerIndex : Int, innerIndex : Int )
     extends Name
  case class URLLiteral( url : java.net.URL )
     extends Name
}
```

Now we wire the two types together.

```
trait ReflectiveGenerators
extends Expressions with Nominals {
  type Nominal = Name
  type Term = Expression
}
```

This allows us to use *quoted* terms as variables in *lambda*-terms! The idea is very rich as it begs the question of whether such variables can be *unquoted* and what that means for evaluation. Thus, `Scala`'s type system is already leading to some pretty interesting places! In fact, this is an instance of a much deeper design principle lurking here, called two-level type decomposition, that is enabled by type-level parametricity. We'll talk more about this in upcoming chapters, but just want to put it on the backlog.

Some syntactic sugar To this core let us add some syntactic sugar.

PREVIOUS	LET	SEQ
$M, N ::= ...$	$\mid let\ x = M\ in\ N$	$\mid M; N$

This is sugar because we can reduce $let\ x\ =\ M\ in\ N$ to $(\lambda x.M)N$ and $M; N$ to $let\ x\ =\ M\ in\ N$ with x not occurring in N.

Digression: complexity management principle In terms of our implementation, the existence of this reduction means that we can choose to have explicit representation of these syntactic categories or not. This choice is one of a those design situations that's of significant interest if our concern is complexity management. [Note: brief discussion of the relevance of super combinators.]

Concrete syntax Now let's wrap this up in concrete syntax.

EXPRESSION	MENTION	ABSTRACTION	APPLICATION
$M, N ::=$	x	$\mid$ `(`x_1`,...,`x_k`) =>` M	$\mid M$`(`N_1`,...,`N_k`)`

LET	SEQ	GROUP
$\mid$ `val` x `=` M`;`N	$\mid M$`;`N	$\mid$ `{` M `}`

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 that is essentially the λ-calculus. Once you familiarize yourself with the λ-calculus as a kind of 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.

If you've been following along so far, however, you will spot that something is actually wrong with this grammar. We still don't have an actual terminal! *Concrete* syntax is what "users" type, so as soon as we get to concrete syntax we can no longer defer our choices about identifiers. Let's leave open the door for both ordinary identifiers – such as we see in `Scala` – and our funny quoted terms. This means we need to add the following productions to our grammar.

IDENTIFIER	STRING-ID	QUOTATION
$x, y ::=$	$\mid String$	$\mid$ `@<`M`>`

(The reason we use the `@` for quotation – as will become clear later – is that when we have both quote and dequote, the former functions a lot like asking for a *pointer* to a term while the latter is a lot like dereferencing the pointer.)

Translating concrete syntax to abstract syntax The translation from the concrete syntax to the abstract syntax is compactly expressed as follows. Perhaps the best way to understand this presentation is in terms of a `Scala` implementation.

Structural equivalence and Relations or What makes abstract syntax abstract Apart from the fact that concrete syntax forces commitment to explicit representation of terminals, you might be wondering if there are any other differences between concrete and abstract syntax. It turns out there are. One of the key properties of abstract syntax is that it encodes a notion of equality of terms that is not generally represented in concrete syntax.

It's easier to illustrate the idea in terms of our example. We know that programs that differ only by a change of bound variable are essentially the same. Concretely, the program (x) =>x + 5 is essentially the same as the program (y) =>y + 5. By "essentially the same" we mean that in every evaluation context where we might put the former if we substitute the latter we will get the same answer.

However, this sort of equivalence doesn't have to be all intertwined with evaluation to be expressed. A little forethought shows we can achieve some separation of concerns by separating out certain kinds of *structural* equivalences. Abstract syntax is where we express structural equivalence. In terms of our example we can actually calculate when two λ-terms differ only by a change of *bound* variable, where by bound variable we just mean a variable mention in a term also using the variable as formal parameter of an abstraction.

Since we'll need that notion to express this structural equivalence, let's write some code to clarify the idea, but because it will be more convenient, let's calculate the variables not occurring bound, i.e. the *free* variables of a λ-term.

```
trait Expressions {
  ...
  def freeVariables( term : Expression ) : Set[Nominal] = {
    term match {
      case Mention( reference ) => Set( reference )
      case Abstraction( formals, body ) =>
         freeVariables( body ) &~ formals.toSet
      case Application( operation, actuals ) =>
         ( freeVariables( operation ) /: actuals )(
         { ( acc, elem ) => acc ++ freeVariables( elem ) }
         )
    }
  }
```

With this code in hand we have what we need to express the structural equivalence of terms.

```
...
 def substitute(
  term : Expression,
  actuals : List[Expression],
  formals : List[Nominal]
) : Expression = {
  term match {
    case Mention( ref ) => {
      formals.indexOf( ref ) match {
        case -1 => term
        case i => actuals( i )
      }
    }
    case Abstraction( fmls, body ) => {
      val fmlsN = fmls.map(
        {
          ( fml ) => {
            formals.indexOf( fml ) match {
              case -1 => fml
              case i => fresh( List( body ) )
            }
          }
        }
      )
      val bodyN =
        substitute(
          body,
          fmlsN.map( _ => Mention( _ ) ),
          fmlsN
        )
      Abstraction(
        fmlsN,
        substitute( bodyN, actuals, formals )
      )
    }
    case Application( op, actls ) => {
      Application(
        substitute( op, actuals, formals ),
        actls.map( _ => substitute( _, actuals, formals ) )
      )
    }
```

```
    }
  }

  def fresh( terms : List[Expression] ) : Nominal

  def `=a=`(
      term1 : Expression,
      term2 : Expression
  ) : Boolean = {
  ( term1, term2 ) match {
    case (
      Mention( ref1 ),
      Mention( ref2 )
    ) => {
      ref1 == ref2
    }
    case (
      Abstraction( fmls1, body1 ), Abstraction( fmls2, body2 )
    ) => {
      if ( fmls1.length == fmls2.length ) {
        val freshFmls =
          fmls1.map(
            { ( fml ) => Mention( fresh( List( body1, body2 ) ) ) }
          )
        `=a=`(
          substitute( body1, freshFmls, fmls1 ),
          substitute( body2, freshFmls, fmls2 )
        )
      }
      else false
    }
    case (
      Application( op1, actls1 ),
      Application( op2, actls2 )
    ) => {
      ( `=a=`( op1, op2 ) /: actls1.zip actls2 )(
        { ( acc, actlPair ) =>
          acc && `=a=`( actlPair._1, actlPair._2 )
        }
      )
    }
```

```
    }
  }
```

Digression: the internal structure of the type of variables If you've been paying attention, you will note that there's something very funny going on in the calculation of freeVariables. To actually perform the remove or the union we have to have equality defined on variables. Now, this works fine for Strings, but what about Quotations?

The question reveals something quite startling about the types of variables. Clearly, the type has to include a definition of equality. Now, if we want to have an inexhaustible supply of variables, then the definition of equality of variables must make use of the "internal" structure of the variables. For example, checking equality of Strings means checking the equality of the respective sequences of characters. There are a finite set of characters out of which all Strings are built and so eventually the calculation of equality grounds out. The same would be true if we used Integers as "variables". If our type of variables didn't have some evident internal structure (like a String has characters or an Integer has arithmetic structure) and yet it was to provide an endless supply of variables, then the equality check could only be an *infinite* table – which wouldn't fit inside a computer. So, the type of variables must also support some internal structure, i.e. it must be a container type!

Fortunately, our Quotations are containers, by definition. However, they face another challenge: are the Quotations of two structurally equivalent terms equal as variables? If they are then there is a mutual recursion! Equivalence of terms depends on equality of Quotations which depends on equivalence of terms. It turns out that we have cleverly arranged things so that this recursion will bottom out. The key property of the structure of the abstract syntax that makes this work is that there is an alternation: quotation, term constructor, quotation, ... Each recursive call will lower the number of quotes, until we reach 0.

Evaluation – aka operational semantics Here's the code

```
trait Values {
  type Environment
  type Expression
  abstract class Value
  case class Closure(
     fn : List[Value] ⇒ Value
  ) extends Value
  case class Quantity( quantity : Int )
```

```
    extends Value
}

trait Reduction extends Expressions with Values {
  type Dereferencer = {def apply( m : Mention ) : Value }
  type Expansionist =
  {def extend(
    fmls : List[Mention],
    actls : List[Value]
    ) : Dereferencer}
  type Environment <: (Dereferencer with Expansionist)
  type Applicator = Expression => List[Value] => Value

  val initialApplicator : Applicator =
  { ( xpr : Expression ) => {
      ( actls : List[Value] ) => {
        xpr match {
          case IntegerExpression( i ) => Quantity( i )
          case _ => throw new Exception( "why are we here?" )
        }
      }
    }
  }

  def reduce(
     applicator : Applicator,
     environment : Environment
  ) : Expression => Value = {
    case IntegerExpression( i ) => Quantity( i )
    case Mention( v ) => environment( Mention( v ) )
    case Abstraction( fmls, body ) =>
    Closure(
       { ( actuals : List[Value] ) => {
           val keys : List[Mention] =
           fmls.map( { ( fml : Nominal ) => Mention( fml ) });
           reduce(
              applicator,
              environment.extend(
                 keys,
                 actuals ).asInstanceOf[Environment]
           )( body )
```

```
            }
          }
        )
        case Application(
          operator : Expression,
          actuals : List[Expression]
        ) => {
          reduce( applicator, environment )( operator ) match {
            case Closure( fn ) => {
              fn.apply(
                (actuals
                 map
                 {( actual : Expression) =>
                   (reduce( applicator, environment ))( actual )})
              )
            }
            case _ =>
              throw new Exception( "attempt to apply non function" )
          }
        }
      case _ => throw new Exception( "not implemented, yet" )
    }
}
```

Chapter 2

Toolbox

Notation and terminology

TBD

2.1 Introduction to notation and terminology

While we defer to the rich and growing body of literature on Scala to provide a more complete set of references for basic Scala notation, to be somewhat self-contained in this section we review the notation and terminology we will need for this book.

2.2 Introduction to core design patterns

2.3 Variations in presentation

2.3.1 A little history

Haskell was the first programming language to popularize the notion of monad as a structuring technique for functional programming. There were several key ideas that went into the `Haskell` packaging of the idea. One was to treat the core elements that make up a monad more or less *directly* without appeal to category theory – the branch of mathematics where the notion originated. This is considerably easier to do in a functional programming language because the ambient language can be thought of as a category; thus, for the average programmer there is no need to

refer to categories, in general, but only to the "universe" of programs that can be written in the language at hand. Then, because Haskell already has a notion of parametric polymorphism, a monad's most central piece of data is a parametric type constructor, say T.

Given such a type constructor, you only need a pair of maps (one of which is higher order). Thus, in `Haskell` a monad is presented in terms of the following data

- a parametric type constructor, T a
- a **return** map enjoying the signature **return** :: a −> T a
- a bind map enjoying the signature bind : T a −> (a −> T b) −> T b

In `Haskell` these elements can be collected inside a typeclass. Resulting in a declaration of the form

```
typeclass Monad T a where
 return :: a -> T a
 bind :: T a -> (a -> T b ) -> T b
```

Listing 2.1: monad typeclass

Now, it's not enough to simply have this collection of pieces. The pieces have to fit together in a certain way; that is, they are subject to the following laws:

- **return** (bind a f) ≡ f a
- bind m **return** ≡ m
- bind (bind m f) g ≡ bind m (\ x −> bind (f x) g)

One of the driving motivations for this particular formulation of the concept is that it makes it very easy to host a little DSL inside the language. The syntax and semantics of the DSL is simultaneously given by the following procedure for de-sugaring, i.e. translating expressions in the DSL back to core `Haskell`.

```
do { x } = x

do { x ; <stmts> }
  = bind x (\_ -> do { <stmts> })

do { v <- x ; <stmts> }
  = bind x (\v -> do { <stmts> })
```

```
do { let <decls> ; <stmts> }
  = let <decls> in do { <stmts> }
```

The assignment-like operation extends to full pattern matching with

```
do { p <- x ; <stmts> }
= let f p = do { <stmts> }
            f _     = fail "..."
    in bind x f
```

On the face of it, the notation provides both a syntax and a semantics reminiscent of the standard side-effecting operations of mainstream imperative languages. In presence of polymorphism, however, these instruments are much more powerful. These operations can be *systematically* "overloaded" (meaning the overloaded definitions satisfy the laws above). This allows to systematically use the notation for a wide variety of computations that all have some underlying commonality. Typical examples include I/O, state management, control flow (all three of which all bundle up in parsing), and also container navigation and manipulation. It gets better for many of the tools of mathematics that are regularly the subject of computer programs such probability distributions, integration, etc., also have presentations as monads. Thus, innocent examples like this one

```
do { putStrLn "Enter a line of text:";
     x <- getLine;
     putStrLn ("you wrote: " ++ x) }
```

as might be found in some on-line tutorial on monads belie the potency of this combination of ideas.

Unlike `Haskell`, `Scala` does not reify the notion of monad under a **trait**, the language's equivalent of `Haskell`'s typeclass. Instead the systematic means of de-sugaring **for**-notation and polymorphic interpretations of flatMap, etc are the effective definitions of the notion in `Scala`.

The basic `Scala` construct looks like

```
for( p <- e [; p <- e] [p = e] [if t] ) yield { e }
```

and the de-sugaring looks like

```
for( x <- expr_1 ; y <- expr_2 ; <stmts> ) yield expr_3

=

expr_1 flatMap( x => for( y <- expr_2; <stmts> ) yield expr_3 )
```

```
for( x <- expr_1 ; y = expr_2 ; <stmts> ) yield expr_3

=

for( ( x, y ) <- for ( x <- expr_1 ) yield ( x, expr_2 ); <stmts> )
yield expr_3

for( x <- expr_1 if pred ) yield expr_2

=

expr_1 filter ( x => pred ) map ( x => expr_2 )
```

Again, general pattern matching is supported in assignment-like statements.

```
for( p <- expr_1 ; <stmts> ) yield expr_2

=

expr_1 filter {
  case p => true
  case _ => false
} flatMap {
  p => for( <stmts> ) yield expr2
}
```

This means, therefore, that we have the following correspondence

2.3.2 A little more history

If one were to reify the notion in `Scala` there are several design choices – all of which endure some desiderata. Following the original presentation developed in category theory, however, has some crucial advantages:

- intuition
- correspondence to previously existing structures
- decomposition of the requirements

which we explore in some detail here.

Intuition: Monad as container

As we will see the notion of monad maps nicely onto an appropriately parametric notion of container. From this point of view we can imagine a container "API" that has three basic operations.

Shape of the container The first of these is a *parametric* specification of the *shape* of the container. Examples of container shapes include: List[A], Set[A], Tree[A], etc. At the outset we remain uncommitted to the particular shape. The API just demands that there is some shape, say S[A].

Putting things into the container The next operation is very basic, it says how to put things into the container. To align with a very long history, we will refer to this operation by the name unit. Since the operation is supposed to allow us to put elements of type A into containers of shape S[A], we expect the signature of this operation to be unit : A =>S[A].

Flattening nested containers Finally, we want a generic way to flatten nested containers. Just like there's something fundamentally the same about the obvious way to flatten nested lists and nested sets, we ask that the container API provide a canonical way to flatten nested containers. If you think about it for a moment, if a container is of shape, S[A], then a nested container will be of shape, S[S[A]]. If history demands that we call our flattening operation mult, then our generic flatten operation will have signature, mult : S[S[A]] =>S[A].

Preserving connection to existing structure: Monad as generalization of monoid

Programmers are very aware of data structures that support a kind of concatenation operation. The data type of String is a perfect example. Every programmer expects that the concatenation of a given String, say s, with the empty String, "" will return a result string equal to the original. In code, s.equals(s + "") ==**true** . Likewise, string concatenation is insensitive to the order of operation. Again, in code, ((s + t) + u).equals(s + (t + u)) ==**true** .

Most programmers have noticed that these very same laws survive polymorphic interpretations of +, equals and the "empty" element. For example, if we substituted the data type Integer as the base type and used integer addition, integer equality, and 0 as the empty element, these same code snippets (amounting assertions) would still work.

Many programmers are aware that there is a very generic underlying data type, historically referred to as a *monoid* defined by these operations and laws. In code, we can imagine defining a **trait** in `Scala` something like

```
trait Monoid {
  def unit : Monoid
  def mult( that : Monoid )
}
```

This might allow *views* of Int as a monoid as in

```
class MMultInt extends Int with Monoid {
  override def unit = 1
  override def mult( that : Monoid ) = this * that
  }
```

except for the small problem that Int is **final** (illustrating an important difference between the adhoc polymorphism of `Haskell`'s typeclass and `Scala`'s **trait**).

Any solution will depend on type parametrization. For example

```
trait Monoid[Element] {
  def unit : Element
  def mult( a : Element, b : Element )
}
```

and corresponding view of Int as a monoid.

```
class MMultInt extends Monoid[Int] {
  override def unit : Int = 1
  override def mult( a : Int , b : Int ) = a * b
}
```

This parametric way of viewing some underlying data structure is natural both to the modern programmer and the modern mathematician. Both are quite familiar with and make extensive use of overloading of this kind. Both are very happy to find higher levels of abstraction that allow them to remain DRY when the programming demands might cause some perspiration. One of the obvious places where repetition is happening is in the construction of view. Consider another view of Int

```
class MAddInt extends Monoid[Int] {
  override def unit : Int = 0
  override def mult( a : Int , b : Int ) = a + b
}
```

It turns out that there is a lot of machinery that is common to defining a view like this for any given data type. Category theorists realized this and recognized that you could reify the view which not only provides a place to refactor the common machinery, but also to give it another level of polymorphism. Thus, a category theorist's view of the monad API might look something like this.

```
trait Monad[Element,M[_]] {
  def unit( e : Element ) : M[Element]
  def mult( mme : M[M[Element]] ) : M[Element]
}
```

The family resemblance to the Monoid API is not accidental. The trick is to bring syntax back into the picture. Here's an example.

```
case class MonoidExpr[Element]( val e : Element )
class MMInt extends Monad[Int,MonoidExpr] {
  override def unit( e : Int ) = MonoidExpr( e )
  override def mult( mme : MonoidExpr[MonoidExpr[Int]] ) =
  mme match {
    case MonoidExpr( MonoidExpr( e ) ) => MonoidExpr( e )
  }
}
```

Decomposition of monad requirements

The constraints on any given monad candidate are well factored into three different kinds of requirements – operating at different levels of the "API": functoriality, naturality and coherence. Often these can be mechanically verified, and when they can't there are natural ways to generate spot-checks that fit well with tools such as `ScalaCheck`.

A categorical way to look at monads

In category theory the monad is presented in terms of the following data

- a "unit" map enjoying the signature $unit : A \to T[A]$
- a "mult" map enjoying the signature $mult : T[T[A]] \to T[A]$

subject to the following laws:

- $T(id_A) = id_{T[A]}$
- – $unit(f) \circ unit_A = unit_B \circ unit(B)$
  – $mult(f) \circ mult_{T[A]} = mult_{T[B]} \circ mult(B)$
- – $mult \circ Tmult = mult \circ multT$
  – $mult \circ Tunit = mult \circ unitT$

These two definitions are interchangeable. That's what makes them "presentations" of the same underlying idea.

One of the reasons for the difference in presentation is that `Haskell` doesn't treat the Monad type class as a Functor. The refactoring of the *mult* map into the *bind* map is that it builds functoriality into definition. The other reason is the do notation.

Chapter 3

An IO-monad for http streams

Code first; questions later

The following code is adapted from Tiark Rompf's work using delimited continuations for handling HTTP streams.

3.1 Code first, questions later

```
import scala.continuations._
import scala.continuations.ControlContext._

import scala.concurrent._
import scala.concurrent.cpsops._


import java.net.InetSocketAddress
import java.net.InetAddress

import java.nio.channels.SelectionKey
import java.nio.channels.Selector
import java.nio.channels.SelectableChannel
import java.nio.channels.ServerSocketChannel
import java.nio.channels.SocketChannel
import java.nio.channels.spi.SelectorProvider

import java.nio.ByteBuffer
import java.nio.CharBuffer
```

```
import java.nio.charset.Charset
import java.nio.charset.CharsetDecoder
import java.nio.charset.CharsetEncoder

import java.util.regex.Pattern
import java.util.regex.Matcher

import java.util.Set

import scala.collection.JavaConversions._

// adapted from http://vodka.nachtlicht-media.de/tutHttpd.html

object DCWebserver
 extends FJTaskRunners {

  case class Generator[+A,-B,+C](val fun: (A => (B @cps[Any,Any])) => (C @c

    def copy = null // FIXME: workaround for named/default params bug

    final def foreach(f: (A => B @cps[Any,Any])): C @cps[Any,Any] = {
      fun(f)
    }
  }

  def selections( selector: Selector )
  : ControlContext[Set[SelectionKey], Unit, Unit] =
  shiftR {
    k: (Set[SelectionKey] => Any) =>

    println("inside select")

      while(true) { // problem???
        val count = selector.selectNow()
        if (count > 0)
          k(selector.selectedKeys())
      }
  }

  def createAsyncSelector() = {
    val selector = SelectorProvider.provider().openSelector()
```

```
    // TODO: this should run in its own thread, so select can block safel
    spawn {
      selections(selector).fun {
        keySet =>
          for (key <- keySet) {
            println("Select: " + key)
            val handler = key.attachment().asInstanceOf[(SelectionKey =>

            println("handling: " + handler)
            handler(key)
          }
        keySet.clear()
      }
    }

    selector
  }


  def callbacks(channel: SelectableChannel, selector: Selector, ops: Int)
    Generator {
      k: (SelectionKey => Unit @cps[Any,Any]) =>
        println("level 1 callbacks")

      shift {
        outerk: (Unit => Any) =>

          def callback(key: SelectionKey) = {
            key.interestOps(0)

            spawn {
              println("before continuation in callback")

              k(key)

              println("after continuation in callback")

              if (key.isValid()) {
                key.interestOps(ops)
                selector.wakeup()
```

```
            } else {
              outerk()
              //returnto.gen();
            }
          }
        }

        println("before registering callback")

        val selectionKey = channel.register(selector, ops, callback _)

        println("after registering callback")
        // stop
        ()
      }
    }

  def acceptConnections(selector: Selector, port: Int) =
    Generator {
      k: (SocketChannel => Unit @cps[Any,Any]) =>
        val serverSocketChannel = ServerSocketChannel.open()
        serverSocketChannel.configureBlocking(false)

        val isa = new InetSocketAddress(port)

        serverSocketChannel.socket().bind(isa)

        for (
          key <-
          callbacks( serverSocketChannel, selector, SelectionKey.OP_ACCEPT
        ) {

          val serverSocketChannel =
            key.channel().asInstanceOf[ServerSocketChannel]

          val socketChannel = serverSocketChannel.accept()
          socketChannel.configureBlocking(false)

          k(socketChannel)
        }
```

```
      println("accept returning")
  }


  def readBytes(selector: Selector, socketChannel: SocketChannel) =
    Generator {
      k: (ByteBuffer => Unit @cps[Any,Any]) =>
        shift {
          outerk: (Unit => Any) =>
            reset {
              val bufSize = 4096 // for example...
              val buffer = ByteBuffer.allocateDirect(bufSize)

              println("about to read")

              for (
                key
                <- callbacks(
                  socketChannel, selector, SelectionKey.OP_READ
                )
              ) {

                println("about to actually read")

                val count = socketChannel.read(buffer)

                if (count < 0) {
                  println("should close connection")
                  socketChannel.close()

                  println("result of outerk " + outerk())
                  //returnto.gen() should cancel here!
                } else {

                  buffer.flip()

                  println("about to call read cont")

                  k(buffer)

                  buffer.clear()
```

```
                shift { k: (Unit⇒Any) ⇒ k() }
              }
            }

            println("readBytes returning")
            outerk()
          }
        }
      }


  def readRequests(selector: Selector, socketChannel: SocketChannel) =
    Generator {
      k: (String ⇒ Unit @cps[Any,Any]) ⇒

        var s: String = "";

        for ( buf <- readBytes( selector, socketChannel ) ) {
          k("read: " + buf)
        }
    }


  def writeResponse(
    selector: Selector,
    socketChannel: SocketChannel,
    res: String
  ) = {
    val reply = res

    val charset = Charset.forName("ISO-8859-1")
    val encoder = charset.newEncoder()

    socketChannel.write(encoder.encode(CharBuffer.wrap(reply)))
  }


  def handleRequest(req: String) = req


  def test() = {
```

```
    val sel = createAsyncSelector()

    println("http daemon running...")

    for (socketChannel <- acceptConnections(sel, 8080)) {

      spawn {
        println("Connect: " + socketChannel)

        for (req <- readRequests(sel, socketChannel)) {

          val res = handleRequest(req)

          writeResponse(sel, socketChannel, res)

          shift { k: (Unit => Any) => k() } // FIXME: shouldn't be needed
        }

        println("Disconnect: " + socketChannel)
      }

      shift { k: (Unit => Any) => k() } // FIXME: shouldn't be needed
    }

  }



  // def main(args: Array[String]) = {

//      reset(test())

//      Thread.sleep(1000*60*60)  // 1h!
//      //     test.mainTaskRunner.waitUntilFinished()

//   }

}
```

3.1.1 An HTTP-request processor

3.1.2 What we did

3.2 Synchrony, asynchrony and buffering

TBD

3.3 State, statelessness and continuations

TBD

Chapter 4

Parsing requests, monadically

How to get from the obligatory to the well formed

TBD

4.1 Obligatory parsing monad

TDB

4.2 Your parser combinators are showing

TBD

4.3 EBNF and why higher levels of abstraction are better

TBD

Chapter 5

The domain model as abstract syntax

In which Pooh and Piglet understand the value of pipelines

TBD

5.1 Our abstract syntax

TBD

5.2 Our application domain model

TBD

5.3 A transform pipeline

TBD

Chapter 6

Zippers and contexts and URI's, oh my!

Zippers are not just for Bruno anymore

TBD

6.1 Zippers are not just for Bruno anymore

TBD

6.2 Constructing contexts and zippers from data types

TBD

6.3 Mapping URIs to zipper-based paths and back

TBD

Chapter 7

A review of collections as monads

Where are we; how did we get here; and where are we going?

TBD

7.1 Monad as container

TBD

7.2 Monads and take-out

TBD

Chapter 8

Domain model, storage and state

Mapping to the backend

TBD

8.1 Mapping our domain model to storage

TBD

8.2 Storage and language-integrated query

TBD

8.3 Continuations revisited

TBD

Chapter 9

Putting it all together

The application as a whole

TBD

9.1 Our web application end-to-end

TBD

9.2 Deploying our application

9.2.1 Why we are not deploying on GAE

9.3 From one web application to web framework

TBD

Chapter 10

The semantic web

Where are we; how did we get here; and where are we going?

TBD

10.1 How our web framework enables different kinds of application queries

TBD

10.2 Searching for programs

TBD