nglint: nginx configuration file linter

Federico Bond
fbond@itba.edu.ar

February 15, 2016

Contents

Introduction **1**

Syntax definition **1**

Preliminary research **2**

Design Considerations **2**

Difficulties encountered **3**

Future extensions **3**

Further research **3**

Conclusions **3**

Bibliography **4**

Introduction

The following report describes the design of a linter for **nginx** configuration files, written in Haskell. **nginx** is a widely used HTTP and reverse proxy server. The motivation behind this project was to further the author's understanding of functional parsing techniques, and apply them in a real-world context.

Syntax definition

The **nginx** configuration uses a C-like syntax consisting of directives, which let you specify the behaviour of the server, and blocks, which define contexts where

these directives apply.

A directive consists of an identifier followed by a list of arguments and a semicolon.

A block consists of an identifier followed by an optional list of arguments and a list of blocks or directives surrounded by braces. Blocks can be nested inside one another.

An example configuration file is shown below:

```
user www-data;
worker_processes auto;
pid /var/run/nginx.pid;

events {
  worker_connections 2048;
  multi_accept on;
  use epoll;
}

http {
  # default http server
  server {
    listen 80;
    root /srv/website;
  }
}
```

Preliminary research

The research for this project included reading the original `parsec` paper (Daan Leijen 2001), several articles and tutorials on Monads and the documentation of more than a dozen other libraries.

Other topics studied, whose understanding did not contribute much to this project, were Functional Reactive Programming, Arrows and the Template Haskell extension to GHC.

Design Considerations

The project has two main components, a parser and a linter. The parser is built with the `parsec` (Daan Leijen 2001) library of monadic parser combinators. The linter uses reverse function composition to apply different filters to the AST generated by the parser, finally returning a list of problematic declarations which is then mapped to individual lint messages.

An additional library, `ansi-terminal`, was used for colored console output.

Difficulties encountered

The most difficult part of this project was to find a proper abstraction for constructing linter rules, which avoided unnecessary boilerplate. I considered the use of Monads, Zippers and the `Traversable` type class, but finally settled for simple composition of filter functions, which proved good enough for the task at hand and kept mental overhead at a reasonable level.

The error output from `ghc` also proved hard to understand at first, especially for someone used to traditional stack traces and syntax errors in imperative / object-oriented code. With practice it became quite natural, though.

Future extensions

Throughout the duration of the project, I kept a To Do list with ideas for future extensions. A sample of them has been reproduced below:

- Add rules which depend on the presence or absence of siblings or otherwise related declarations.
- Add command line options for ignoring certain rules.
- Ignore linter errors when preceded by a specially formatted comment.
- Use the QuickCheck library for testing the codebase.

Further research

I would like to explore more idiomatic alternatives for constructing linter rules. The `hlint` and `shellcheck` projects might be useful for this task. I would also like to deepen my understanding of Monad Transformers, which appear useful for reducing the nesting of code and improving error handling.

Conclusions

I had never written a complete program in Haskell before. The experience proved highly valuable and encouraged me to use Haskell and other pure functional programming languages like Elm and PureScript in future scenarios. I was surprised at how easily it was to reason about the code at hand.

Having said that, the state of documentation in the Haskell ecosystem leaves much to be desired. Having to read academic papers to understand how to use each library is a bit tedious, but you grow used to it over time. Many libraries

could benefit from a concise README with sample code and common caveats, specially considering that the heavy use of types in Haskell makes it easy to find what you need from there.

Bibliography

Daan Leijen, Erik Meijer. 2001. "Parsec: Direct Style Monadic Parser Combinators for the Real World." http://research.microsoft.com/en-us/um/people/daan/download/papers/parsec-paper.pdf.