nglint: nginx configuration file linter

Federico Bond
fbond@itba.edu.ar

February 15, 2016

Introduction

The following report describes the design of a linter for **nginx** configuration files, written in Haskell as final course project for the Functional Programming course at ITBA.

Linters are widely used in the software industry. These tools run static analysis on code to pinpoint errors and potential trouble spots. They can be executed manually or configured to run automatically inside a Continuous Integration server, to ensure that no developer within the team introduces new issues.

Several linters have been written in Haskell before. The most famous outside the Haskell ecosystem is Shellcheck, which finds problematic constructs in shell scripts. Another well known linter is HLint, which was used in this project and provided useful suggestions for simplifying some of the code written. It is worth mentioning that these tools can also serve an educational purpose, providing contextual advice to the user in order to improve his or her code.

The nginx server

nginx is a widely used HTTP and reverse proxy server. It can serve static files over the web and/or proxy web applications, among other things. It is considered one of the fastest HTTP servers around, and companies as big as

Netflix (streaming video) and Automattic (which owns WordPress.com, one of the biggest blogging platforms in the web) use **nginx** to run their business.

The **nginx** server is configured via text files following a custom syntax definition resembling C code. This sintax makes it really easy to customize most aspects of the server behavour, but knowing how the different settings work together requires a lot of experience. There are hundreds of options and some of them have grave security consequences. Novice users could benefit from a tool that offered helpful contextual suggestions in order to harden and improve their server configuration.

Configuration 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 new contexts where these directives apply. An example configuration file is shown below:

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

events {
  worker_connections 2048;
  use epoll;
}

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

In this example, there are two blocks, `events` and `http`, and several directives such as `user` and `worker_connections`. A simplified version of this grammar is provided below in extended Backus-Naur form:

```
<config> ::= <declaration>*
<declaration> ::= <directive> | <block>
<directive> ::= <identifier> <arg>* ';'
<block> ::= <identifier> <arg>* '{' <declaration>* '}'
```

Preliminary Research

My first task was researching the topic of parsing within the functional programming paradigm. Many tools exist to build parsers for imperative and object-oriented languages. Lex and Yacc are two very popular ones that have been around for a very long time. Parsing is usually divided into two steps. First, a lexer turns a byte stream from a source file into a stream of tokens, and then a parser turns this stream of tokens into an Abstract Syntax Tree (AST) which is a representation of the structure of the source file inside the program.

Traditional tools for building parsers use their own syntax to describe the language grammar and the ways that constructs can be combined together is thus predetermined by the tool. One alternative is to use functional parser combinators, a topic that has gathered the attention of researchers for many decades. These combinators are first-class values within the host language, which opens the posibility to much richer composition.

In 2001, Daan Leijen and Erik Meijer introduced Parsec, a parser combinator library for Haskell that was robust enough to be used for real-world projects, unlike many of the previous implementations, which suffered from space leaks or inability to report precise error messages[1]. Parsec quickly gained adoption and is now used in more than 700 projects published in Hackage, the most used Haskell package archive.

The parser combinators from `parsec` work inside the `Parser` monad. A typical parser looks like this:

[1] http://research.microsoft.com/en-us/um/people/daan/download/papers/parsec-paper.pdf

```
identifier :: Parser String
identifier = do
  c <- identStart
  cs <- many identLetter
  return (c:cs)
```

This parser will combine the `identStart` and `identLetter` parsers to produce a new parser. Using this technique, we can start with a few primitives like `char 'a'`, which parses a literal `a`, and compose them together with combinators like `many` in order to produce more complex parsers. All combination is done within the `Parser` monad.

Program Architecture

The linting process can be divided into three steps:

```
Parse -> Lint -> Output
```

The parsing step takes the file contents as input and produces an Abstract Syntax Tree (AST) as output.

Once the parser has finished its job, the linter will apply one by one a set of rules to the AST, each returning a list of hints. Rules first match certain declarations against the syntax tree and then return a hint (which is a linter suggestion) for each one of the matched declarations. One way to do this would be to traverse the whole AST for each rule, recurring on declarations that contain other declarations inside, like blocks and if statements. Fortunately, we don't need to do that if we encapsulate the recursion into a set of matchers that can be composed together.

A matcher is thus defined as a function of type `[Decl] -> [Decl]`. Functions that take values from type `a` and return values of the same type can be composed together trivially. Matcher composition is also an associative operation. Matching directive `d` with argument `a`, inside block `b` is equivalent to matching directive `d` inside block `b`, with argument `a`. The reason why we can't use simple function composition via tde dot operator is that it would require us to write the matchers backwards: the initial list of declarations would be passed as an argument to the rightmost function, whose result would become an input to the function on its left. Let's take a look at a simple example:

```
matchRootInsideLocation :: Matcher
matchRootInsideLocation =
    matchBlock "location" >>>
    matchDirective "root"

-- >>> is defined in Control.Arrow as:
-- f (>>>) g = g . f
```

We want to find the blocks named `location`, which contain a directive named `root` inside. The first function will take a list of declarations and return only the `location` blocks. The second must then filter and find the directives named `root`. This style of composition reads more natural than:

```
matchRootInsideLocation :: Matcher
matchRootInsideLocation =
  matchDirective "root" . matchBlock "location"
```

Finally, we pass the list of hints to a formatter, which takes the list and outputs it to the console. The formatter can be configured via command-line options. A `pretty` formatter is configured as default for manual usage. This formatter outputs uses ANSI terminal colors using the `ansi-terminal`[2] library. Another formatter, named `gcc` is provided for which works better for programmatic consumption or integration into other tools.

[2]https://hackage.haskell.org/package/ansi-terminal-0.6.2.3

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 function composition in the form of the `(>>>)` operator, 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. The `Parser` monad described earlier, for example, is defined as the monad transformer `ParserT` applied to the `Identity` monad.

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.

Appendix A: Code

```haskell
import Control.Monad
import Data.Version (showVersion)
import NgLint.Linter
import NgLint.Messages
import NgLint.Parser
import NgLint.Output.Common
import Paths_nglint (version)
import System.Console.GetOpt
import System.Environment
import System.Exit
import Text.Parsec
import Text.Parsec.Error
import qualified NgLint.Output.Pretty as Pretty
import qualified NgLint.Output.Gcc as Gcc

data OutputFormat = Pretty | Gcc deriving (Show, Eq)
data Flag = Format OutputFormat deriving (Show, Eq)
data LinterConfig = LinterConfig
    { outputFormat :: OutputFormat }

defaultConfig = LinterConfig
    { outputFormat = Pretty }

printUsage :: IO ()
printUsage =
    putStrLn $ usageInfo header options
    where header = unlines [ "nglint - nginx configuration file linter"
                           , "version: " ++ showVersion version ]

lintFile :: Formatter -> FilePath -> IO [LintMessage]
lintFile format fileName = do
```

```
    content <- readFile fileName
    let config = parse configFile fileName content
    case config of
        Left error -> do
            print error
            return []
        Right (Config decls) -> do
            let messages = lint decls
            format content messages
            return messages


formatp :: Maybe String -> Flag
formatp (Just str) =
    case str of
        "gcc" -> Format Gcc
        "pretty" -> Format Pretty
        _ -> error "unrecognized output format"
formatp Nothing = Format Pretty


options :: [OptDescr Flag]
options = [Option "f" ["format"] (OptArg formatp "FORMAT") "message output format"]


configFromOpts :: LinterConfig -> [Flag] -> LinterConfig
configFromOpts config (Format outputFormat : opts) =
    configFromOpts (config { outputFormat = outputFormat }) opts
configFromOpts config [] = config


getFormatter :: OutputFormat -> (String -> [LintMessage] -> IO ())
getFormatter Pretty = Pretty.printMessages
getFormatter Gcc = Gcc.printMessages
```

```haskell

main :: IO ()
main = do
    args <- getArgs
    let (opts, nonOpts, errors) = getOpt Permute options args
        linterConfig = configFromOpts defaultConfig opts

    when (null nonOpts) $
        printUsage >> exitSuccess

    let printMessages = getFormatter $ outputFormat linterConfig

    totalMessages <- mapM (lintFile printMessages) nonOpts
    let num = length $ concat totalMessages

    when (outputFormat linterConfig == Pretty) $
        putStrLn $ show num ++ " hints."

    if num > 0 then exitFailure else exitSuccess
```

../src/Main.hs

```haskell
module NgLint.Parser where

import Control.Applicative (liftA)
import Data.Maybe
import NgLint.Position
import Text.Parsec
import Text.Parsec.Language
import Text.Parsec.String
import qualified Text.Parsec.Token as P

data RegexOp = CaseSensitiveRegexMatch
              | CaseInsensitiveRegexMatch
              | CaseSensitiveRegexNotMatch
```

```
              | CaseInsensitiveRegexNotMatch
              deriving (Show, Eq)

data CmpOp = Equal | NotEqual deriving (Show, Eq)

data FileOp = FileExists
              | FileNotExists
              | DirectoryExists
              | DirectoryNotExists
              | AnyExists
              | AnyNotExists
              | Executable
              | NotExecutable
              deriving (Show, Eq)

data Literal = Variable String | StringLit String deriving (Show, Eq)
data Condition =
    ConditionVariable String
    | Compare CmpOp Literal Literal
    | RegexMatch RegexOp Literal Literal
    | FileOperation FileOp Literal
    deriving (Show, Eq)

data Decl =
    Comment SourcePos String
    | Block SourcePos String [String] [Decl]
    | Directive SourcePos String [String]
    | IfDecl SourcePos Condition [Decl]
    deriving (Show)

data Config = Config [Decl] deriving (Show)

instance Position Decl where
    getPos (Comment pos _) = pos
    getPos (Block pos _ _ _) = pos
```

```haskell
    getPos (Directive pos _ _) = pos
    getPos (IfDecl pos _ _) = pos

configFile :: Parser Config
configFile = do
    spaces
    lst <- decl `sepBy` spaces
    spaces
    eof
    return $ Config lst

nginxDef = emptyDef
    { P.identStart     = letter <|> char '_'
    , P.identLetter    = alphaNum <|> char '_'
    , P.opLetter       = oneOf ":!#$%&*+./<=>?@\\^|-~"
    }

lexer = P.makeTokenParser nginxDef

parens      = P.parens lexer
braces      = P.braces lexer
identifier  = P.identifier lexer

decl :: Parser Decl
decl = choice $ map try [comment, ifDecl, directive, block]

arg = many1 (alphaNum <|> oneOf "\"*_-+/.'$[]~\\:^()|=?!")

-- http://stackoverflow.com/questions/34342911/parsec-parse-nested-code-blocks
sepBy1Try :: (Stream s m t) => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
sepBy1Try p sep = do
  x <- p
  xs <- many (try $ sep *> p)
```

```
  return (x:xs)

sepByTry p sep = sepBy1Try p sep <|> return []

block :: Parser Decl
block = do
    pos <- getPosition
    name <- identifier
    spaces
    args <- arg `sepByTry` spaces
    spaces
    decls <- braces (decl `sepEndBy` spaces)
    spaces
    return $ Block pos name args decls
    <?> "block"

comment :: Parser Decl
comment = do
    pos <- getPosition
    char '#'
    msg <- manyTill anyChar endOfLine
    return $ Comment pos msg
    <?> "comment"


variable :: Parser String
variable = do
    char '$'
    name <- identifier
    return name

stringLit :: Parser Literal
stringLit = do
    char '"'
    str <- manyTill anyChar (try $ char '"')
```

```haskell
    return $ StringLit str

condVariable :: Parser Condition
condVariable = do
    name <- variable
    return $ ConditionVariable name

condCompare :: Parser Condition
condCompare = do
    var <- variable
    spaces
    op <- cmpOp
    spaces
    str <- stringLit
    return $ Compare op (Variable var) str

regexMatch :: Parser Condition
regexMatch = do
    var <- variable
    spaces
    op <- regexOp
    spaces
    regex <- stringLit
    return $ RegexMatch op (Variable var) regex

fileOp :: Parser FileOp
fileOp = do
    negative <- liftA isJust $ optionMaybe (char '!')
    char '-'
    op <- choice $ map char "fdex"

    let isNegative a b = if negative then a else b
    return $ case op of
       'f' -> isNegative FileExists FileNotExists
       'd' -> isNegative DirectoryExists DirectoryNotExists
```

```haskell
       'e' -> isNegative AnyExists AnyNotExists
       'x' -> isNegative Executable NotExecutable

regexOp :: Parser RegexOp
regexOp = do
    negative <- liftA isJust $ optionMaybe (char '!')
    char '~'
    insensitive <- liftA isJust $ optionMaybe (char '*')
    return $ case (negative, insensitive) of
        (False, False) -> CaseSensitiveRegexMatch
        (False, True)  -> CaseInsensitiveRegexMatch
        (True,  False) -> CaseSensitiveRegexNotMatch
        (True,  True)  -> CaseInsensitiveRegexNotMatch

cmpOp :: Parser CmpOp
cmpOp = do
    negative <- liftA isJust $ optionMaybe (char '!')
    char '='
    return (if negative then NotEqual else Equal)


fileOperation :: Parser Condition
fileOperation = do
    op <- fileOp
    spaces
    var <- variable
    return $ FileOperation op (Variable var)


condition :: Parser Condition
condition = choice $ map try [condCompare, regexMatch, condVariable, fileOperation]


ifDecl :: Parser Decl
```

```haskell
ifDecl = do
    pos <- getPosition
    string "if" <?> "if"
    spaces
    cond <- parens condition
    spaces
    decls <- braces (decl `sepEndBy` spaces)
    spaces
    return $ IfDecl pos cond decls

directive :: Parser Decl
directive = do
    pos <- getPosition
    name <- identifier
    spaces
    args <- arg `sepEndBy` spaces
    char ';'
    return $ Directive pos name args
    <?> "directive"
```

../src/NgLint/Parser.hs

```haskell
module NgLint.Linter where

import Data.List
import Data.Maybe
import NgLint.Messages
import NgLint.Parser
import NgLint.Rules

rules = [ noRootInsideLocation
        , ifIsEvil
        , sslv3Enabled
        , serverTokensOn
        , tlsv1Enabled ]
```

```haskell

lint :: [Decl] -> [LintMessage]
lint decls = sort $ concatMap (\rule -> rule decls) rules
```

../src/NgLint/Linter.hs

```haskell
module NgLint.Messages where

import Control.Arrow ((>>>))
import NgLint.Parser
import NgLint.Position
import Text.Parsec.Pos (SourcePos)

data ErrorCode = NG001 | NG002 | NG003 | NG004 | NG005 deriving (Eq)

data LintMessage = LintMessage SourcePos ErrorCode deriving (Eq)

instance Show LintMessage where
    show (LintMessage pos code) = show pos ++ ": " ++ show code

instance Ord LintMessage where
    compare (LintMessage p1 _) (LintMessage p2 _) = compare p1 p2

instance Position LintMessage where
    getPos (LintMessage pos _) = pos

instance Show ErrorCode where
    show NG001 = "NG001: root directive inside location block"
    show NG002 = "NG002: if can be replaced with something else"
    show NG003 = "NG003: enabling SSLv3 leaves you vulnerable to POODLE attack"
    show NG004 = "NG004: enabling server_tokens leaks your web server version number"
    show NG005 = "NG005: enabling TLSv1 leaves you vulnerable to CRIME attack"
```

```haskell

label :: ErrorCode -> [Decl] -> [LintMessage]
label code = map buildMessage
    where buildMessage decl = LintMessage (getPos decl) code
```

../src/NgLint/Messages.hs

```haskell
module NgLint.Matchers where

import NgLint.Parser

type Matcher = [Decl] -> [Decl]

matchBlock :: String -> Matcher
matchBlock name = concatMap matches
    where matches block@(Block _ bname _ decls) =
              if bname == name
                  then block : matchBlock name decls
                  else matchBlock name decls
          matches _ = []

matchDirective :: String -> Matcher
matchDirective name = concatMap matches
    where matches directive@(Directive _ dname _) =
              [directive | dname == name]
          matches (Block _ _ _ decls) = matchDirective name decls
          matches (IfDecl _ _ decls) = matchDirective name decls
          matches _ = []

matchIfDecl :: Matcher
matchIfDecl = concatMap matches
    where matches ifDecl@(IfDecl _ _ _) = [ifDecl]
          matches (Block _ _ _ decls) = matchIfDecl decls
          matches _ = []

```

```haskell
matchArg :: String -> Matcher
matchArg str = filter hasArg
    where hasArg (Directive _ _ args) = str `elem` args
          hasArg _ = False
```

../src/NgLint/Matchers.hs

```haskell
module NgLint.Position where

import Text.Parsec.Pos

class Position a where
    getPos :: a -> SourcePos
```

../src/NgLint/Position.hs

```haskell
module NgLint.Rules where

import Control.Arrow ((>>>))
import NgLint.Matchers
import NgLint.Messages
import NgLint.Parser

type Rule = [Decl] -> [LintMessage]

mkRule :: ErrorCode -> Matcher -> Rule
mkRule code matcher = matcher >>> label code

noRootInsideLocation :: Rule
noRootInsideLocation = mkRule NG001 $
    matchBlock "location" >>>
    matchDirective "root"

ifIsEvil :: Rule
ifIsEvil = mkRule NG002 matchIfDecl
```

```haskell

sslv3Enabled :: Rule
sslv3Enabled = mkRule NG003 $
    matchDirective "ssl_protocols" >>>
    matchArg "SSLv3"

serverTokensOn :: Rule
serverTokensOn = mkRule NG004 $
    matchDirective "server_tokens" >>>
    matchArg "on"

tlsv1Enabled :: Rule
tlsv1Enabled = mkRule NG005 $
    matchDirective "ssl_protocols" >>>
    matchArg "TLSv1"
```

../src/NgLint/Rules.hs

```haskell
module NgLint.Output.Common where

import NgLint.Messages

type Formatter = String -> [LintMessage] -> IO ()
```

../src/NgLint/Output/Common.hs

```haskell
module NgLint.Output.Gcc where

import Data.List
import NgLint.Messages
import NgLint.Output.Common
import System.Console.ANSI
import Text.Parsec.Pos

printMessage :: LintMessage -> IO ()
```

```haskell
printMessage (LintMessage pos code) = do
    let (errorNumber, message) = splitAt 5 (show code)
    putStr $ intercalate ":" $ map ($ pos) [sourceName, show . sourceLine, show . sourceColumn]
    putStrLn $ ": warning" ++ message ++ " [" ++ errorNumber ++ "]"

printMessages :: Formatter
printMessages contents = mapM_ printMessage
```

../src/NgLint/Output/Gcc.hs

```haskell
module NgLint.Output.Pretty where

import Data.List
import NgLint.Messages
import NgLint.Output.Common
import System.Console.ANSI
import Text.Parsec.Pos

printMessage :: LintMessage -> IO ()
printMessage (LintMessage pos code) = do
    setSGR [SetColor Foreground Vivid Yellow]
    putStr $ replicate (sourceColumn pos - 1) ' '
    putStrLn ("^-- " ++ show code)
    setSGR [Reset]

printMessageGroup :: [String] -> [LintMessage] -> IO ()
printMessageGroup lns messages = do
    let (LintMessage pos _) = head messages

    setSGR [SetConsoleIntensity BoldIntensity]
    putStrLn $ "In " ++ sourceName pos ++ ", line " ++ show (sourceLine pos) ++ ":"
    setSGR [Reset]
    putStrLn (lns !! (sourceLine pos - 1))
```

```haskell

    mapM_ printMessage messages
    putChar '\n'

printMessages :: Formatter
printMessages contents messages = do
    let lns = lines contents
        messageGroups = groupBy eq messages
        eq (LintMessage p1 _) (LintMessage p2 _) = p1 == p2

    mapM_ (printMessageGroup lns) messageGroups
```

../src/NgLint/Output/Pretty.hs