Guidelines for the Creation of Readable Source Code

Varik Valefor

November 25, 2021

Contents

Chapter 1

Executive Summary

A very brief overview of VARIK's requirements for readable source code is as follows:

- For all lines, the length of a line is less than or equal to seventy-two (72) characters.
- For all functions, the length of a function is less than or equal to fifteen (15) lines.
- For all functions f, the purpose of f is explained within a comment which is specific to f.
- For all tokens t, the name of t is descriptive, or the meaning of t is fully explained.

Chapter 2

Disclaimer

This document is a work in progress. Some things may be missing or overall a bit cheesy. The inclusion of stuff can be requested at `https://github.com/varikvalefor/readablesourcecode/issues`.

Chapter 3

Comments

3.1 Executive Summary

For all functions k, there exists a comment c such that c describes the functionality of k without too verbosely describing the inner workings of k.

For all variables v, there exists a comment c such that c describes the content of c and *possibly* the purpose of v.

3.2 Functions

3.2.1 Top-Level Summaries

For all functions f, there exists a comment c such that c describes the functionality of c without describing the non-limiting technical details of the implementation of f. c immediately precedes f.

For all functions f, for all arguments a of f, the purpose of a is explained.

3.2.2 Implementation Details

For all functions f, for all crappy/odd/confusing parts of f l, a comment which explains the purpose and functionality of f precedes f.

Additionally, for all functions f, for all sub-functions and variables φ of f, the purpose or meaning of φ should be explained if determining the purpose or meaning of φ is not *really* a trivial exercise for the reader.

3.2.3 Examples

Example qK49R6kK

An example of a well-documented function is as follows:

```
-- | @loadQuiz k@ returns a 'Quiz'-based representation of the quiz
-- whose name is @k@.
--
-- Quiz files are read from @/usr/share/games/quiz.db/@.
loadQuiz :: String
         -- ^ The name of the quiz which should be returned
         -> IO Quiz;
loadQuiz = parseQuiz <.> T.readFile . ("/usr/share/games/quiz.db/" ++);
```

`loadQuiz`'s top-level comment is sufficiently detailed to be of great use to the users of `loadQuiz`. However, the top-level comment of `loadQuiz` does not contain any unnecessary detail.

No comments are present within the body of `loadQuiz`; the body of `loadQuiz` is simple such that comments in the body of `loadQuiz` are of real use only to beginners.

Example gZbkNtqW

An example of a function which is documented *reasonably* well is as follows:

```
-- | @((a <.> b) c) == (a <$> b c)@.
(<.>) :: Functor f
      => (a -> b)
      -> (c -> f a)
      -> c
      -> f b;
(<.>) a b c = a <$> b c;
infixl 1 <.>;
```

The term "reasonably" is used because although the top-level comment of `(<.>)` is essentially the body of `(<.>)`, explaining `(<.>)` in other terms involves writing a pretty decent amount of text, and the resulting explanation is likely relatively confusing.

The arguments of `(<.>)` are not described with comments because such comments would just be equivalent to the type signature of `(<.>)`.

Example LMjrUhHR

A well-documented function is as follows:

```
-- | Where @k@ represents a @m.room.message@ of message type
-- @m.location@, @valueMTextToStdMess@ is a 'StdMess' which should be
-- equivalent to @k@.
valueMLocationToStdMess :: Value
                        -- ^ The representation of the message which
                        -- should become a 'StdMess'
                        -> StdMess;
valueMLocationToStdMess k = Def.stdMess {
  msgType = Location,
  body = k .! "{content:body}",
  geo_uri = k .! "{content:geo_uri}",
  boilerplate = valueToECF k
};
```

The functionality of the source code of the function is fairly obvious if function `(.!)` is understood. However, `valueMLocationToStdMsg` is complex such that the length of the top-level comment which describes `valueMLocationToStdMsg` can be less than the length of the source code of `valueMLocationToStdMsg`.

Example 747g64rm

A well-documented function is as follows:

```
-- | @grab@ is used to fetch and output the messages of a room.
--
-- @grab@'s argument follows the pattern [NUMBER OF MESSAGES, "EARLY" OR
-- "RECENT", JUNK DATA, ID OF DESIRED MATRIX ROOM].
```

```
grab :: [String]
     -- ^ The first 3 elements of this list are the decimal number of
     -- messages which should be nabbed, "early" or "recent", some junk
     -- data, and the internal Matrix ID of the room from which messages
     -- should be nabbed.
     -> Auth
     -- ^ The authorisation information
     -> IO ();
grab k a
  | k == [] = error "Repent, motherfucker."
  | n < 0 = error "I need a natural number, not garbage."
  | n == 0 = error "Why in the hell would you want to take 0 messages?"
  | otherwise = case k !! 1 of
    "recent" -> recentMessagesFrom n room a >>= mapM_ print
    "early"  -> earlyMessagesFrom n room a >>= mapM_ print
    _        -> error "I'll grab you if you don't grab some sense."
  where
  -- \| This variable refers to the number of messages which should be
  -- fetched.
  n :: Integer
  n = fromMaybe (-42) $ readMaybe $ head k
  --
  room :: Room
  room = Def.room {roomId = k !! 3};
```

In addition to including a few jokes, `grab` has good documentation; `grab`'s documentation outlines the purpose of `grab`, the purposes of the arguments of `grab`, and the purpose of a variable whose name is not terribly descriptive.

Example NyT0xsii

A poorly-documented function is as follows:

```
mean :: Num a => Floating b => [a] -> b;
mean l = fromIntegral (sum l) / fromIntegral (length l);
```

Although the name "`mean`" is not at all misleading, a bit of confirmation would be nice.

A relatively well-documented version of `mean` is as follows:

```
-- | @mean k@ is the mean of @k@.
mean :: Num a
     => Floating b
     => [a]
     -- ^ The list whose mean is output
     -> b;
mean l = fromIntegral (sum l) / fromIntegral (length l);
```

3.3 Variables

For all variables v, there exists a comment c such that c describes the content of v and *possibly* the purpose of v.

c explains the purpose of v if v has *a single purpose*.

3.3.1 Examples

Example 2PH4x9Qv

An example of a well-documented variable is as follows:

```
-- | @homeslice@ returns the content of the "HOME" environment
-- variable.
homeslice :: IO String;
homeslice = getEnv "HOME";
```

The documentation of `homeslice` is succinct, and all good documentation is succinct.

The only potential problem is the dumb joke name of `homeslice`. However, because the value of `homeslice` is explained, the name "`homeslice`" `should` not cause any problems. But dummies are known to do some crazy things.

The phrase "content of" is not redundant; if "content of" is not present, then the reader may assume that `homeslice` returns the entire `HOME` environment variable, e.g., `HOME=/root`.

Example pN5Neyrs

A completely inexcusable variable declaration is as follows:

```
a :: IO Int;
a = (`mod` 5) . (^2) <$> klang;
```

The content and purpose of `klang` are all but entirely invisible. The relationship of `a` and `klang` is obvious — the monadic value of `a` is congruent to the square of the monadic value of `klang` modulo 5 — but the purpose of `a` is completely undocumented, and the name "`a`" provides no insight as to why the reader should care about the existence of `a`.

Chapter 4

Length

4.1 Executive Summary

For all lines l, the character-based length of l is less than or equal to seventy-two (72).

For all functions f, the line-based length of f's definition is less than or equal to fifteen (15).

4.2 Lines

For all lines l, the character-based length of l is less than or equal to seventy-two (72).

The maximum line length of seventy-two (72) characters is chosen because all terminals of a decent size can easily display a line whose length is less than or equal to seventy-two (72) characters.

4.2.1 Explaining the Choice

Definitions

$\mathbb{L}$ denotes the set of all lines.

$\mathbb{T}$ denotes the set of all terminals.

For all $\langle l, t \rangle \in \mathbb{L} \times \mathbb{T}$, $d(t, l)$ iff t easily displays l properly.

A Proof!

Theorem 1. *For all $l \in \mathbb{L}$, $L(l) \leq 72$.*

Proof.

$$\forall l \in \mathbb{L},\ \nexists t \in \mathbb{T} : \neg d(t,l) \implies L(l) \leq 72.$$

$$\forall l \in \mathbb{L},\ \nexists t \in \mathbb{T} : \neg d(t,l).$$

$$\therefore \forall l \in \mathbb{L},\ L(l) \leq 72.$$

□

4.3 Functions

4.3.1 Executive Summary

For all functions f, the line-based length of f's definition is less than or equal to fifteen (15). Lines of documentation are not counted.

4.3.2 Explaining the Choice

Definitions

$\mathbb{F}$ denotes the set of all functions.

For all $f \in \mathbb{F}$, $d(f)$ denotes the source code-based definition of f.

For all $f \in \mathbb{F}$, $l(f)$ denotes the line-based length of $d(f)$. $l(f)$ *does not* include documentation which is contained within $d(f)$.

For all $f \in \mathbb{F}$, $c(f)$ iff f is excessively complex.

For all $f \in \mathbb{F}$, $e(f)$ iff the definition of f can be easily read and understood by a man who is not a novice of the language in which f is written.

A Proof!

Theorem 2. *For all $f \in \mathbb{F}$, $l(f) \leq 15$.*

Proof.

$$\forall C \in \Omega^3,\ (C_1 \implies (\neg C_2) \vee C_3) \wedge (C_2 \wedge (\neg C_3)) \implies \neg C_1.$$
$$\forall f \in \mathbb{F},\ l(f) > 72 \implies (\neg e(f)) \vee (c(f))\,.$$
$$\forall k \in \mathbb{N}^2,\ \neg\,(k_1 > k_2) \iff k_1 \leq k_2.$$
$$\forall f \in \mathbb{F},\ l(f) \in \mathbb{N}.$$
$$\forall f \in \mathbb{F},\ e(f) \wedge \neg c(f).$$
$$15 \in \mathbb{N}.$$
$$\therefore \forall f \in \mathbb{F},\ l(f) \leq 15.$$

□

4.3.3 Exceptions

This subsection lists a subset of the "types" of functions which can reasonably be relatively long.

Functions which Output Values of Algebraic Data Types

Functions which output values of algebraic data types can be relatively lengthy; the definitions of such values are often inherently long.

Functions which Contain Good Sub-Functions and Variables

For all functions f, for all sub-functions and variables s of f, s is good iff s is of real use *only* within f.

For all functions f, the lengths of the declarations of the good sub-functions and variables of f are not included in the length of f.

4.3.4 Examples

Example pNFOEuuN

A function whose length can be justified is as follows:

```
instance Options Opt where
  defineOptions = pure Opt
    -- predetBase
    <*> defineOption optionType_string (\o -> o
        {
          optionShortFlags = "f",
          optionDefault = "",
          optionDescription = "This value is the name of the \
          \pre-defined character set which functions as the encoding \
          \of the input."
        })
    -- predetDest
    <*> defineOption optionType_string (\o -> o
        {
          optionShortFlags = "t",
          optionDefault = "decimal",
          optionDescription = "This value is the name of the \
          \pre-defined character set which functions as the encoding \
          \of the output."
        })
    -- customBase
    <*> defineOption optionType_string (\o -> o
        {
          optionShortFlags = "F",
          optionDefault = "",
          optionDescription = "This value is the character set of the \
          \input.  The first character is the 0-valued digit, and the \
          \final value is the n-valued digit, where n is difference of \
          \the length of the input character set and 1."
        })
    -- customDest
    <*> defineOption optionType_string (\o -> o
        {
          optionShortFlags = "T",
          optionDefault = "",
          optionDescription = "This value is the character set of the \
          \output.  The first character is the 0-valued digit, and the \
          \final value is the n-valued digit, where n is the \
          \difference of the length of the output character set and 1."
        });
```

The definition of `Opt`'s `defineOptions` is reasonably long; however, the definition of `defineOptions` can be “followed” with reasonable ease, as a few comments which separate the options are included.

Chapter 5

Breaking Stuff Up

5.1 Executive Summary

Functions which are terribly long should be split into multiple functions.

5.2 "When Should Functions be Split?"

For all functions f, f should be split into multiple functions iff f is excessively lengthy, f does some stuff such that other functions may benefit from such stuff's being moved into a global function, or f contains some function arguments which are frequently re-used.

5.3 Methods of Splitting Functions

5.3.1 Executive Summary

Functions which are excessively lengthy should be split into multiple local or global functions.

5.3.2 Global Splitting

For all excessively lengthy functions f, f can be split into multiple functions such that the resulting functions are accessible by functions which are defined outside of the scope of f.

For all excessively lengthy functions f, f should be split into multiple global functions iff there may exist another function g such that g benefits from the creation of such global functions.

5.3.3 Local Splitting

In many programming languages, for all excessively lengthy functions f, f can be split into multiple functions such that the resulting functions are accessible only within the scope of f.[1]

For all excessively lengthy functions f, f should be broken into multiple functions which are usable only within the scope of f iff the resulting sub-functions are likely of real use only within the scope of f.

[1]"In many programming languages" is written because some programming languages, e.g., C, support only janky-looking sub-functions...or do not support *any* sub-functions. When such programming langages are used, the use of local functions is infeasible or outright impossible; as such, global functions should generally be used.

5.4 Examples

Example VZk6MTuH

A function definition which is unreasonably long is as follows:

```
-- | @longAssPieceOfCrap k@ iff @k@ is prime.
longAssPieceOfCrap :: Integer
                   -- ^ The number whose primality is checked
                   -> Bool;
longAssPieceOfCrap n = [] == filter ((== 0) . (mod n)) [2..n `div` 2];
```

`longAssPieceOfCrap` is *somewhat* true to `longAssPieceOfCrap`'s name. However, `longAssPieceOfCrap` is hardly the worst thing in the world. Just pardon the cheesy little "`` `div` 2``" thing.

The mildly experienced Haskeller can fairly quickly re-write `longAssPieceOfCrap` as follows:

```
-- | @longAssPieceOfCrap k@ iff @k@ is prime.
longAssPieceOfCrap :: Integer
                   -- ^ The number whose primality is checked
                   -> Bool;
longAssPieceOfCrap n = [] == divisors n;

-- | @divisors k@ is a list which contains all non-1 and non-@k@
-- divisors of @k@.
divisors :: Integer
         -- ^ The number whose divisors should be output
         -> [Integer];
divisors k = filter ((== 0) . (mod n)) [2..k `div` 2];
```

`divisors` can be reasonably added as a global function because other functions may benefit from the existence of `divisors`.

Example Q1Kd5TTS

A terribly long — and contrived — function definition is as follows:

```
←
IOO
factor ← {1 =   : 1   ↓    =/1(0=(1+)|)/1+ :     ,/factor ¨    {, ÷}  / ↓  {1(0=(1+)|)/1+}}
```

Dear God! Some APL symbols are not typeset correctly!

Hopefully, this problem is fixed in the future.

In the meantime, the reader must tolerate looking at some broken APL code. Alternatively, the reader can view the raw APL source code which is contained within the source code of this document, *or* the reader can even attempt to fix the problem.

Returning to the actual point of this subsubsection, `factor` can be shortened significantly if a portion of `factor` is moved to a function `isPrime`. The result of such a move is as follows:

```
←
IOO
isPrime ← ↓   {=/1(0=(1+)|)/1+}
factor ← {1 =   : 1   isPrime   :     ,/factor ¨    {, ÷}  / ↓  {1(0=(1+)|)/1+}}
```

Although functions `factor` and `isPrime` can be reverse-engineered within a few hours, the readability of `factor` and `isPrime` is hardly maximised.

A part of `factor` can become a new function `divisors`. The result of moving part of `isPrime` into `divisors` is as follows:

```
 ←
I00
divisors ←↓    {1(0=(1+)|)/1+}
isPrime ←  {=/divisors  }
factor ← {1 =   : 1   isPrime   :     ,/factor ¨    {,÷}  /divisors  }
```

However, the snippet's simplicity is still not maximised!

A function `divides` can be extracted from `divisors`, yielding the following result:

```
 ←
I00
divides ←  {0=|}
divisors ←↓  {1((1+) divides   )/1+}
isPrime ←  {=/divisors  }
factor ← {1 =   : 1   isPrime   :     ,/factor ¨    {,÷}  /divisors  }
```

Simplification is possible still.

The janky-looking defun-like thing which appears multiple times in `divisors` can become a tradfun, yielding the following result:

```
 ←
I00
i ←  {1+}
divides ←  {0=|}
divisors ←↓ {1(( i) divides  )/ i}
isPrime ←  {=/divisors  }
factor ← {1 =   : 1   isPrime   :     ,/factor ¨    {,÷}  /divisors  }
```

Adding documentation to this snippet is left as an exercise for the reader.

Further simplifying this snippet is also left as an exercise for the reader; VARIK finds that attempting to further simplify the snippet may be a waste of time, and VARIK does not wish to waste time.