A simple Monadic Parsec based parser in Haskell

Mario Lang
e-Mail: mlang@delysid.org

Revision: April 25, 2010

Abstract

This program implements a simple validating parser for a timesheet ASCII file. It is implemented in Haskell, using the monadic parser combinator library Parsec.

Contents

1 Introduction

The following module implements a timesheet validating parser written in Haskell.

It uses Parsec — a Monadic parser combinator library.

This is the first program I wrote in Haskell, so it is much of a learning project for me. If you have any suggestions, please send them via e-Mail.

1.1 Timesheet format

The format of my personal timesheet is very simple:

```
Mi 02.01.: 08:50-15:50 -01:00
Do 03.01.: 09:45-17:30 -00:15
Fr 04.01.: 09:50-17:40 -00:10
Sa 05.01.: 14:00-22:30 +08:30
So                    =+07:05
Mo 07.01.: 10:10-17:30 -00:40
Di 08.01.: 09:45-17:50 +00:05
Mi 09.01.: 11:00-17:00 -02:00
Do 10.01.: 09:30-17:15 -00:15
Fr 11.01.: 11:45-19:00 -00:45
Sa
So                    =+03:30
```

We assume that on a normal weekday (Monad through Friday) I have to work 8 hours.

The last column of a line from Monad through Saturday is simply the remaining time for this day. On a Sunday, we have the remaining time as a whole. This makes the file easy to read and also easy to verify for correctness.

I use this file-format personally to record how much I've worked.

The following code implements two goals at once:

1. Validate that the individual sums were entered correctly, and
2. sum up the whole timesheet and return the remaining time.

2 Module and import definitions

```
module Stunden where

import Parsec
import ParsecChar
import Maybe
import Monad(liftM)
```

3 Convenience

As we have some common characters in our file-format, we are going to define those here:

```
dot,colon,comma :: CharParser Minutes Char
dot = char '.'; colon = char ':'; comma = char ','
```

4 Representing and parsing time

Time in our specialized time-sheet format always looks like this: [+−−]*hh*:*mm*

4.1 Internal representation

So, how can we represent this internally. We want to calculate with time. Well, it's basicly nothing else than Minutes since midnight:

We define a newtype to be able to derive instances of this time:

```
newtype Minutes = Minutes Int deriving (Eq,Ord)
```

And now we define simple mathematical operations on Minutes:

```
instance Num Minutes where
  (Minutes a) + (Minutes b) = Minutes (a+b)
  (Minutes a) - (Minutes b) = Minutes (a-b)
  (Minutes a) * (Minutes b) = Minutes (a*b)
  fromInteger = Minutes . fromInteger
  abs (Minutes x) = Minutes (abs x)
  negate (Minutes x) = Minutes (-x)
  signum (Minutes x) = Minutes (signum x)
```

Also, it would be nice if we could use the function show on a value of type Minutes and get the representation we decided to use.

```
instance Show Minutes where
    show (Minutes m) | m >= 0 = '+' : hh m ++ ":" ++ mm m
                     | m < 0 = '-' : hh (-m) ++ ":" ++ mm (-m)
      where
        hh m = zeropad (m `div` 60)
        mm m = zeropad (m `mod` 60)
        zeropad x | x < 10 = '0' : show x
                  | otherwise = show x
```

To illustrate what the above code actually does, here is an example interactive session using our newly defined data constructor and show function.

Call: `Minutes 960`

Result:

Call: `Minutes 0 - Minutes 23`

Result:

Call: `Minutes (-99)`

Result:

4.2 Parsing time

Well, to be able to parse our hh:mm format, we are going to define a two-digit parser first:

(Note: One shouldn't really use read in Parsec-based programs. Any good alternative here?)

```
number :: CharParser Minutes Int
number = count 2 digit >>= return.read
```

Now we can define a simple hh:mm parser.

```
time :: CharParser Minutes Minutes
time = number >>= λh →
                  colon >>
                  number >>= λm →
                  return (Minutes (h*60+m))
```

A interval (09:00-17:00) indicates a time interval.

```
duration :: CharParser Minutes Minutes
duration = time >>= λt1 →
                              char '-' >>
                              time >>= λt2 →
                              return (t2−t1)
```

And we also want to be able to have several intervals on one day, separated by a comma.

```
durations :: CharParser Minutes Minutes
durations = liftM ∑ (duration `sepBy` comma)
```

5 A day of work

A day is represented using the german two-letter initials.

Lets first define a data type for internal use:

```
data Day = Mo | Di | Mi | Do | Fr | Sa | So
                          deriving (Eq,Ord,Enum,Show)
```

Now we define a parser which tries to match one of our two-letter days, and if it matches one, it returns it.

```
dayOfWeek :: CharParser Minutes Day
dayOfWeek = foldr1 (<—>) (map isDay [Mo ..])
    where
      isDay x = try (string (show x)) >> return (x)
```

A day of work looks like this:

```
Mi 02.01.: 08:50-15:50 -01:00
```

The final column is the sum of remaining time for this day.

```
workday :: CharParser Minutes Minutes
workday =
        dayOfWeek >>= λdow → space >>
        number >> dot >> number >> dot >> colon >> space >>
        durations >>= λdur →
```

```
    updateState (+((−towork dow)+dur)) >>
    many1 space >>
    getState >>= λbalance →
    (if dow = So then
        string ("=" ++ show balance)
    else
        string (show ((−towork dow)+dur)))
    >> many (oneOf " \t\n") >>
    getState >>= return
where
    towork day | day `elem` [Mo .. Fr] = 8*60
               | otherwise = 0
```

6 Putting things together

The whole timesheet is a string composed of several lines.

```
workentries :: CharParser Minutes Minutes
workentries =
    many workday >> eof >>
    getState >>= return
```

Finally we need a function to handle the final parser result, so that we can print error messages, or the remaining time if everything went OK.

```
parsework :: String → IO ()
parsework =
        putStrLn . either error result . runParser workentries (Minutes 0) ""
  where
    result min = ("Remaining time " ++ show min ++ "!")
    error = (++) "parse error at " . show
```

7 Main

Very similar to C, a Haskell program also needs a Main module (and a main function) if you want to link it into an executable file.

Our Main module is very simple, it only tries to figure out if a command-line argument was passed, and if yes, use that as filename for the input file. Here it is:

```
module Main where

import System (getArgs)
import IO (hGetContents, openFile, stdin, IOMode(ReadMode))
import Stunden (parsework)

main = getArgs >>= λargs →
            (if null args then
                hGetContents stdin
            else
                openFile (args!!0) ReadMode >>= λfh →
                hGetContents fh)
```

8 Testing

Now that we have implemented all necessary functions and data-types, lets put in a test-case to illustrate how the whole thing actually works.

The following is a test-case on our program. First we show the given input data, below you can see the ouput the program produces.

Input:
```
Mi 02.01.: 08:50-15:50 -01:00
Do 03.01.: 09:45-17:30 -00:15
Fr 04.01.: 09:50-17:40 -00:10
Sa 05.01.: 14:00-22:30 +08:30
So 06.01.:            =+07:05
```

Output:

Now this is fine, but how does the parser react if we give it incorrect data?

Input:
```
Mi 02.01.: 08:50-15:50 -01:00
Do 03.01.: 09:45-17:30 -00:15
Fr 04.01.: 09:50-17:40 -00:10
Sa 05.01.: 14:00-22:30 +08:30
So 06.01.:             =+07:00
```

Output:

9 Conclusion

We clearly can see that the implicit error handling of Parsec is a real advantage here. Without ever explicitly specifying any kind of coloumn or line number handling, we get all this for free!

9.1 Open Questions

It would be nice if we could implement non-terminating Warnings.

For instance, the parsework function could only warn if the sum of a day was wrong, but fail (terminate) if the sum of a week is wrong.

I didn't find a way yet to pass warnings on in the Parser Monad, because the return type is Either, and Left returns the final result of the parser, whereas Riight returns an ParserError type which doesn't contain the state of the parser as far as I understand.

Any hints on how to easily implment warnings would be nice.

10 Building

The following listing shows how to compile and build all the files related to this project.

It is written using "make", a standard UNIX tool.

Listing: Makefile – The building process in a nutshell

```
# Some default values
HC = ghc
HFLAGS = -package text
HCFLAGS = -O2

# We are going to build those files
RESULTS = Stunden.ps Stunden.pdf Stunden

# Start of build-rules:
all: $(RESULTS)

Stunden: Stunden.o Main.o
	$(HC) $(HFLAGS) -o $@ $+
	strip $@

Stunden.dvi: Stunden.lhs Main.lhs Makefile Stunden

clean:
	rm -f *.hi *.o *.inp *.res $(RESULTS)

release:
	$(MAKE) all
	rm -f *.hi *.o *~
	cd .. && tar -czvf Stunden.tar.gz Stunden

# Implicit rules for Haskell compilation:
%.o: %.lhs
	$(HC) $(HCFLAGS) $(HFLAGS)  -o $@ -c $<

# Implicit rules for pdf/ps generation:
%.toc %.aux %.log %.inp %.res: %.lhs
	shell_escape=y latex $<

%.dvi: %.lhs %.toc
	shell_escape=y latex $<

%.ps: %.dvi
	dvips $<

%.pdf: %.dvi
	dvipdf $<

# make-specific settings
```

```
.PRONY: clean release
.INTERMEDIATE: Stunden.aux Stunden.dvi Stunden.inp Stunden.log Stunden.res
# DO NOT DELETE: Beginning of Haskell dependencies
Stunden.o : Stunden.lhs
Main.o : Main.lhs
Main.o : Stunden.hi
# DO NOT DELETE: End of Haskell dependencies
```