Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment 1 – Part 1

Jeroen Bransen, João Pizani

Deadline: Tuesday, 25 November 2014 23:59

As a result of the first two assignments of the course, we will end up building a parser (and a few so-called "semantic functions") for files in the *iCalendar* format, a calendar exchange format. See for instance Wikipedia at

`http://en.wikipedia.org/wiki/ICalendar`

for an informal explanation of the format.

The iCalendar format is used to store and exchange meeting requests, tasks and appointments in a standardized format. The format is supported by a large number of products, including Google Calendar and Apple iCal.

In this first assignment, we will implement a parser for a standard date/time format, and we will define a set of datatypes to represent an iCalendar file. There are some bonus exercises for implementing more features.

Parser combinators

For this task, you are supposed to use the parser combinators as discussed in the lectures. These are contained in a Haskell package called `uu-tc` which is available from Hackage[1].

There are two versions of the parser combinator library in that package. You can get the one which is as described in the lecture notes by saying `import ParseLib` or alternatively `import ParseLib.Simple`. In the lectures, I use a variant of that library that keeps the parser implementation abstract. This variant is available by saying `import ParseLib.Abstract`.

You can choose which variant you want to use, but I recommend `ParseLib.Abstract`.

Alternatively, for bonus points you can use the `uu-parsinglib` package, see exercise 7 for more information. You are allowed to directly implement your all parsers in your solution using that library, but I recommend that you start with `uu-tc`.

[1] `http://hackage.haskell.org/package/uu-tc`

General remarks

Here are a few remarks:

- Make sure your program compiles (with an installed `uu-tc` package). Verify that `ghc --make -O Assignment1.hs` works prior to submission. If it does not, your solution will not be graded.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of your program, special cases, preconditions etc.
- Try to write readable and idiomatic Haskell. Style influences the grade! The use of existing higher-order functions such as `map`, `foldr`, `filter`, `zip` – just to name a few – is highly encouraged. The use of existing libraries is allowed (as long as the program still compiles with the above invocation).
- Copying solutions from the internet is not allowed.
- We prefer teams of size two, but a one person team size is allowed. A team must submit a single assignment and put both names on it.
- Textual answers to tasks can be included as comments in the source file submitted
- Submission is done through DomJudge, at `https://domjudge.cs.uu.nl/tc/team`. In the same package where you got this PDF file there is a file named "Date-Time.hs". This is the *starting framework* file in which you should write the answers to the programming questions. Some datatypes and type signatures might already be defined. The rest is up to you to define. The outputs of your solution will be compared with a "model solution" through the automatic *DomJudge* system. You can submit as many attempts as you want until the deadline.

Date and time

Let's start our journey towards the iCalendar format with a simple task: parsing a date/time format.

The concrete syntax of a date and time value in so-called *Standard Algebraic Notation* (SAN) is given by the following grammar:

datetime	::= *date datesep time*
date	::= *year month day*
time	::= *hour minute second timeutc*
year	::= *digit digit digit digit*
month, *day*, *hour*, *minute*, *second*	::= *digit digit*
timeutc	::= ε | `Z`
digit	::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
datesep	::= `T`

Terminals are written in typewriter font, nonterminals in italics. A *datetime* value has a fixed length and contains the date and time values as fixed length integers. The optional trailing `Z` is used to indicate that this date/time is expressed in UTC (Coordinated Universal Time). If the `Z` is omitted the time should be interpreted as local time.

No whitespace is allowed anywhere in a date/time value!

When writing a parser, one of the most important decisions is which data structure to use as *target* of the parser, that is, which datatype will be produced by the parser. In this case, we will represent date/time values using the following Haskell datatype:

```
data DateTime = DateTime {date :: Date
                         ,time :: Time
                         ,utc  :: Bool}
  deriving Eq

data Date = Date {year  :: Year
                 ,month :: Month
                 ,day   :: Day}
  deriving Eq

newtype Year  = Year  {unYear :: Int}  deriving Eq
newtype Month = Month {unMonth :: Int} deriving Eq
newtype Day   = Day   {unDay :: Int}   deriving Eq

data Time = Time {hour   :: Hour
                 ,minute :: Minute
                 ,second :: Second}
  deriving Eq

newtype Hour   = Hour   {unHour :: Int}   deriving Eq
newtype Minute = Minute {unMinute :: Int} deriving Eq
newtype Second = Second {unSecond :: Int} deriving Eq
```

1 (3 pt, medium)**.** Define a parser

```
parseDateTime :: Parser Char DateTime
```

that can parse a single date and time value. This implies that you have to define parsers for all the other types (Date, Time, Hour, etc.) too.

2 (1 pt)**.** Define a function

```
run :: Parser a b -> [a] -> Maybe b
```

that applies the parser to the given input. Of all the results the parser returns, we are interested in the *first* result that is a *complete* parse, i.e., where the remaining list of input symbols is empty. If such a result exists, it is returned. Otherwise, `run` should return `Nothing`.

3 (1 pt). Define a printer

```
printDateTime :: DateTime -> String
```

that turns a date and time value back into SAN notation. The idea is that for any value `dt` of type `DateTime` we have that

```
run parseDateTime (printDateTime dt) == Just dt
```

i.e., that printing the date and time and then parsing it again succeeds and results in the same abstract representation of the date and time. Similarly, for valid SAN strings `s` we should have that

```
parsePrint s == Just s
```

where

```
parsePrint s = printDateTime <$> run parseDateTime s
```

4 (0 pt). Test your parser for date and time on a couple of examples, by parsing and printing examples:

```
*Main> parsePrint "19970610T172345Z"
Just "19970610T172345Z"
*Main> parsePrint "19970715T040000Z"
Just "19970715T040000Z"
*Main> parsePrint "19970715T40000Z"
Nothing
*Main> parsePrint "20111012T083945"
Just "20111012T083945"
*Main> parsePrint "20040230T431337Z"
Just "20040230T431337Z"
```

As you might have noticed, the last example demonstrates that our grammar (and hence, our parser) also accepts some *invalid* values: for instance, hours can only range from `0` to `23`, but currently all 2-digit integers are accepted.

5 (2 pt). Write a function

```
checkDateTime :: DateTime -> Bool
```

that verifies that a `DateTime` represents a valid date and time. Any 4-digit value should be accepted as a valid year, and years in "BC" (*Before Christ*) are ignored. Valid months are in the range `1 - 12`, and valid days are in the range `1 - 28`, `1 - 29`, `1 - 30` or `1 - 31`, depending on the month. Valid times are those where the hour is in the range `0 - 23` and minute and seconds are in the range `0 - 59`.

Refer to

`http://en.wikipedia.org/wiki/Month#Julian_and_Gregorian_calendars`

for more information about the number of days per month. Make sure that you handle leap years in the correct way!

```
*Main> let parseCheck s = checkDateTime <$> run parseDateTime s
*Main> parseCheck "19970610T172345Z"
Just True
*Main> parseCheck "20040230T431337Z"
Just False
*Main> parseCheck "20040229T030000"
Just True
```

Tip: Write functions to work with the datatypes produced by the parser, even for the simple ones. For example, if you need to subtract two values of `Year`, define a function `subYears :: Year -> Year -> Year`.

Events and full calendar file

We will now extend our definition to events. The concrete syntax of events is as follows:

```
event       ::= BEGIN:VEVENT crlf
                eventprop*
                END:VEVENT crlf
eventprop   ::= dtstamp | uid | dtstart | dtend | description | summary | location
dtstamp     ::= DTSTAMP:      datetime crlf
uid         ::= UID:          text     crlf
dtstart     ::= DTSTART:      datetime crlf
dtend       ::= DTEND:        datetime crlf
description ::= DESCRIPTION:  text     crlf
summary     ::= SUMMARY:      text     crlf
location    ::= LOCATION:     text     crlf
```

Here *crlf* is a Carriage Return and Line Feed, represented in Haskell as `"\r\n"`, and *text* is a string of characters not containing carriage return or line feed. No extra whitespace is allowed anywhere in an event.

Note that on Windows, *readFile* translates `"\r\n"` automatically to `"\n"`. If you want to test your implementation at this point, make sure you read the hints in **??**.

Finally, the concrete syntax of a full iCalendar file is defined as:

```
calendar ::= BEGIN:VCALENDAR crlf
             calprop*
             event*
             END:VCALENDAR crlf
```

calprop	::=	*prodid* | *version*
prodid	::=	`PRODID:` *text crlf*
version	::=	`VERSION:2.0` *crlf*

Here is an informal explanation of the syntax: An iCalendar file consists of a standard header and a sequence of events. Both the standard header and the event consist of a set of properties. The properties are name-value pairs, separated by a colon, each on a separate line ended by a carriage return and line feed. Events and the main calendar object are blocks surrounded by `BEGIN` and `END` lines.

Some of the properties are required and most properties must appear exactly once. The order in which the properties must appear within an event is not defined. In the header both *prodid* and *version* are required and must appear exactly once. In an event the properties *dtstamp*, *uid*, *dtstart* and *dtend* are required and must appear exactly once. *description*, *summary* and *location* are optional but must not appear more than once.

6 (3 pt)**.** Define Haskell datatypes (or type synonyms) to describe the abstract syntax of an iCalendar file. Call the type for a whole iCalendar file `Calendar`.

Hint: The abstract syntax **does not need** to have the same structure as the concrete syntax. Read the informal explanation of the format several times, and think about the best way to represent the calendar.

Bonus exercises

7 (bonus, 1 pt, medium)**.** With the parser combinators from the `uu-tc` library, there is no elegant way to define parsers for properties that must appear exactly once but can appear in any order. In the `uu-parsinglib`[2] library there is a `MergeAndPermute` module which allows you to specify this in a nice way.

Implement all parsers in your solution using the `uu-parsinglib`.

[2]`http://hackage.haskell.org/package/uu-parsinglib`