Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment 1 – Part 2

Jeroen Bransen, João Pizani

Deadline: Friday, 9 December 2016 23:59

The goal of this assignment is to write a parser 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.

The specification of version 2.0 of the iCalendar format is given at

`http://tools.ietf.org/html/rfc5545`

and is quite extensive. In this assignment, we will implement only a **small subset** of the features which should be enough to parse simple iCalendar files. There is a bonus exercise for implementing extra features.

Parser combinators

Like in the first assignment, 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, we 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 8 for more information. You are allowed to directly implement all your parsers using that library, but I recommend that you start with `uu-tc` and only switch to `uu-parsinglib` when everything works.

[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 ICalendar.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 explicitly encouraged. The use of all existing libraries is allowed (as long as the program still compiles as above). Use Hoogle to search for functions with the types that you need.
- Copying solutions from the internet is not allowed.
- We prefer teams of size two, but a one person team size is allowed. One person of the team should be responsible for uploading the assignment, but BOTH names should be on ALL files uploaded.
- 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 `ICalendar.hs`. This is the *starting framework* file in which you should code the answers to the programming questions. Some datatypes and type signatures might already be defined. The rest is up to you.
- The outputs of your solution will be compared with a "model solution" by *DomJudge*. You can submit as many attempts as you want until the deadline. The submission considered for grading is the *last correct submission before the deadline*.

Reusing code from Part 1

One part of parsing a calendar is parsing dates and times. Hopefully, you did a good job at the previous assignment and can *reuse* the parser for `DateTime`. Nothing changed: the grammar of `DateTime` is exactly the same, so if you had a working parser, you can just copy/paste it.

In the starting framework module for this assignment (`ICalendar.hs`) we define the DateTime type *again* (to make the file "self-contained"), so please copy/paste *only your parser*, and *not* the old `DateTime` datatype.

Events and full calendar file

We now turn our attention to the definition of events, and of a full calendar file.

First of all, the concrete syntax of an *event* 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 followed by line feed, represented in Haskell as `"\r\n"`, and `text` is a string of characters containing neither carriage return nor line feed. No extra whitespace is allowed anywhere in an event.

Note that on Windows, the `readFile` function translates `"\r\n"` automatically to `"\n"`. When testing your implementation, make sure you read the hints in 2.

Now, having a definition for events, we can go on and define the grammar for a full iCalendar file as follows:

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.

1 (4 pt, difficult)**.** Define a parser

```
parseCalendar :: Parser Char Calendar
```

that can parse a complete iCalendar file. Note that you have a choice here. You can directly define the parser directly on strings, or you can write a lexical analyzer (a.k.a. scanner or lexer) that first transforms the input into a stream of tokens. Using a lexer, your design becomes simpler – you don't have to handle whitespace in the second phase – but it may also require a little extra work.

An important part of defining a parser is defining which datatype does the parser *target*, that is, values of which type are produced by the parser. For this and the following exercises, consider the type `Calendar` to be defined as follows:

```
data Calendar = Calendar {prodId :: String
                         ,events :: [VEvent]}
  deriving Eq


data VEvent = VEvent {dtStamp     :: DateTime
                     ,uid         :: String
                     ,dtStart     :: DateTime
                     ,dtEnd       :: DateTime
                     ,description :: Maybe String
                     ,summary     :: Maybe String
                     ,location    :: Maybe String}
  deriving Eq
```

Important: Do NOT copy the `Calendar` type that you developed in the previous assignment. It is important that all students use the same target type, so submissions can be graded automatically by DomJudge.

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

```
readCalendar :: FilePath -> IO (Maybe Calendar)
```

that uses `openFile` and `hGetContents` to read a file of the given name and then parses it. There is a small example file `bastille.ics` (in the *examples* directory of this assignment's package) that you should be able to parse using this function. A couple other example files are also in the same directory, most of which should be readable.

Many iCalendar files that you will find on the internet will NOT be handled by this simple parser. However, there is a bonus exercise to improve the parser and make it recognize more complex examples.

On Windows machines, Haskell translates `"\r\n"` automatically to `"\n"`. However, the iCalendar format explicitly defines that newlines should be `"\r\n"`, so you should turn this automatic translation off using `hSetNewlineMode` and `noNewlineTranslation`.

3 (1.5 pt)**.** Define a printer

```
printCalendar :: Calendar -> String
```

that generates a string representation from an abstract `Calendar` object. Try to make the layout of the generated string nice and readable. For instance, put every property on a line ended by a carriage return and line feed. Since the printer might change the layout, we will not in general have the property that parsing a file and printing it results in the original string again. However, the other direction should hold. For any value `c` of type `Calendar`,

```
run parseCalendar (printCalendar c) == Just c
```

Here's a possible (not the only one) result of printing `bastille.ics` – note the changes in layout:

```
BEGIN:VCALENDAR
PRODID:-//hacksw/handcal//NONSGML v1.0//EN
VERSION:2.0
BEGIN:VEVENT
SUMMARY:Bastille Day Party
UID:19970610T172345Z-AF23B2@example.com
DTSTAMP:19970610T172345Z
DTSTART:19970714T170000Z
DTEND:19970715T040000Z
END:VEVENT
END:VCALENDAR
```

4 (1.5 pt, medium)**.** Write a function (or several functions) that for a value of type `Calendar` answers the following questions:

- How many events are there in the calendar?
- Are there any events (and if yes, which) happening at a given date and time? An event should **not** be counted if the searched time matches exactly the end time.
- Are there any overlapping events?
- How much time (in minutes) is spent in total for events with a given summary?

Hint: You can choose whatever form of computation you find easiest. In particular, you can choose to define your own datatypes, and whether you want to define one or multiple functions to collect the data. In any case, remember the standard recipe for defining functions on datatypes!

5 (2 pt, difficult)**.** Write a viewer that can visualize a calendar in ASCII graphics. It is recommended to use a *pretty printing* library for this, for example `Text.PrettyPrint`[2].

Write a function

[2] `http://hackage.haskell.org/package/pretty`

```
ppMonth :: Year -> Month -> Calendar -> String
```

that, given a month and year, gives a visual overview of all appointments in that month. For example, when called with the `rooster_infotc.ics` example for November 2012, it should give an output like:

```
1             | 2             | 3             | 4             | 5             | 6             | 7
--------------+---------------+---------------+---------------+---------------+---------------+--------------
8             | 9             | 10            | 11            | 12            | 13            | 14
              |               |               |               | 12:15 - 14:00 |               |
              |               |               |               | 14:15 - 16:00 |               |
              |               |               |               | 14:15 - 16:00 |               |
--------------+---------------+---------------+---------------+---------------+---------------+--------------
15            | 16            | 17            | 18            | 19            | 20            | 21
08:00 - 09:45 |               |               |               | 12:15 - 14:00 |               |
10:00 - 11:45 |               |               |               | 14:15 - 16:00 |               |
10:00 - 11:45 |               |               |               | 14:15 - 16:00 |               |
--------------+---------------+---------------+---------------+---------------+---------------+--------------
22            | 23            | 24            | 25            | 26            | 27            | 28
08:00 - 09:45 |               |               |               | 12:15 - 14:00 |               |
10:00 - 11:45 |               |               |               | 14:15 - 16:00 |               |
10:00 - 11:45 |               |               |               | 14:15 - 16:00 |               |
--------------+---------------+---------------+---------------+---------------+---------------+--------------
29            | 30            |               |               |               |               |
08:00 - 09:45 |               |               |               |               |               |
10:00 - 11:45 |               |               |               |               |               |
10:00 - 11:45 |               |               |               |               |               |
```

You do not need to stick exactly to this format and you may choose a different representation. Of course, representations that look more like a real calendar will get more points, for example when the columns are fixed to the days of the week and the first day of the month starts in the correct column.

Hint: It is not trivial to let both the columns and rows vary in size. It is thus a good idea to give the columns a fixed width and let the rows vary in height. Think also about what to do for appointments that span over multiple days.

Bonus exercises

6 (bonus, 0.5 pt)**.** Instead of appearing at a single line, text values can also span over multiple lines by starting the next line with a single space or tab character. For example:

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//hacksw/handcal//NONSGML v1.0//EN
BEGIN:VEVENT
UID:12345@example.com
DTSTAMP:20111205T170000Z
DTSTART:20111205T170000Z
DTEND:20111205T210000Z
SUMMARY:This is a very long description that
 spans over multiple lines.
END:VEVENT
END:VCALENDAR
```

Adapt your lexical analyzer or your parser to support multiple line texts. Using a lexer might involve less work here, so if you did not use one until now, reconsider...

7 (bonus, 0.5 pt)**.** Until now we have only considered a small subset of the iCalendar format. Extend your solution such that it can handle larger examples from the *examples* directory. The main goal is to be able to parse more files, so you should only change your parser, but not change the `Calendar` datatype.

8 (bonus, 0.5 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`[3] library there is a `Interleaved` module which allows you to specify this in a nice way.

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

Hint: The `Interleaved` module was recently moved to a new library (`uu-interleaved`), which is already available when you install `uu-parsinglib`. The full name of the module is `Control.Applicative.Interleaved`.

9 (bonus, 1 pt, medium)**.** The iCalendar format can also contain timezone information. Implement support for timezones in your solution.

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