Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment P1

Jeroen Bransen

Deadline: Sunday, 1 December 2013 23:59

The goal of this assignment is to write a parser (including 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.

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 a subset of the features which should be enough to parse simple iCalendar files. 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 16 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 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 with the above invocation).
- Copying solutions from the internet is not allowed.
- You may work alone or with one other person. A team must submit a single assignment and put both names on it.
- Textual answers to tasks can either be included as comments in a source file, or be submitted as text or PDF files. Microsoft Word documents are not accepted!

Date and time

We will build the abstract syntax and the parser bottom-up. We will start with date and time, then extend everything to a single event and finally to a full calendar file.

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 and time is UTC time or absolute time. If the `Z` is omitted the time should be interpreted as local time.

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

1 (0.5 pt)**.** Define Haskell datatypes to describe the abstract syntax of a date and time value. Call the datatype for this value `DateTime`. Note that you will probably have to declare multiple datatypes, but you may also decide to represent certain syntactic categories using type synonyms if you prefer.

Hint: it may be helpful to use `deriving (Show, Eq)` in your datatype declarations.

2 (1 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 you have introduced, too.

3 (0.25 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`.

4 (0.75 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
```

5 (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 not only valid date and times are accepted by our grammar (and hence, by our parser): for instance, hours can only range from `0` to `23`, but currently all 2-digit integers are accepted.

6 (0.5)**.** Write a function

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

that verifies that a `DateTime` represents a valid date and time. Any 4-digit value is accepted as a valid year, so years in "BC" 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
```

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 9.

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.

7 (1.25 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. Think about the best way to represent the calendar.

8 (2 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 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. With the latter, your design may be simpler – you don't have to deal with whitespace in the second phase – but it may also require more work.

9 (0.25 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` on the Wiki that you should be able to parse using this function. There are a couple of other examples available on the Wiki, most of which should be readable. Many iCalendar files that you will find on the internet will not be handled by this parser (yet, this can be done as a bonus exercise).

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`.

10 (1 pt)**.** Define a printer

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

that generates a string representation from an abstract iCalendar object. Try to make the layout of the generated string 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 valid) 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
```

11 (1 pt, medium)**.** Write a function (or several functions) that for a value of type `Calendar` answers the following questions:

- how many events are in the calendar?
- are there any events, and if yes which events, happening at a given date and time?

- are there any overlapping events?
- how much time, in seconds, is spend 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 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!

12 (1.5 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]. If you want to use another pretty printing library, it should be available from Hackage.

Write a function

```
ppMonth :: Int -> Int -> 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 therefore 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.

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

Bonus exercises

13 (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 th
 at spans over multiple lines.
END:VEVENT
END:VCALENDAR
```

Adapt your lexical analyzer or your parser to support multiple line texts.

14 (bonus, 0.5 pt)**.** Until now we have only considered a small subset of the iCalendar format. Extend your solution such that your solution can handle the larger examples from the wiki. The main goal is to be able to parse more files, so you may decide to only change your parser and throw away certain extra data.

15 (bonus, 1 pt, difficult)**.** `QuickCheck`[3] is a unit testing tool for QuickCheck, which you can use for testing the correctness of functions. QuickCheck does this by generating valid inputs for your functions and then validate properties, specified by the user, of the result of the function.

Write QuickCheck tests for at least the properties specified in the assignment. For this you will need to make some of your datatypes an instance of `Arbitrary`, see the QuickCheck documentation for more information.

16 (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`[4] 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`.

17 (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/QuickCheck`
[4] `http://hackage.haskell.org/package/uu-parsinglib`