Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment P1

Jeroen Bransen, João Pizani, Trevor McDonell,
Alejandro Serrano, David van Balen

November 21, 2023

As a result of the first assignment of the course, we will end up building a parser (and a few so-called "semantic functions") for files in (a simplified version of) 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.

Parser combinators

For this task, you are supposed to use parser combinators as discussed in the lectures. These are contained in a Haskell package called `uu-tc` which is available from Hackage. The documentation is available at `http://hackage.haskell.org/package/uu-tc`.

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`. We recommend that you `import ParseLib.Abstract` for easier-to-understand error messages.

General remarks

Here are a few remarks:

- Make sure your program compiles (with an installed `uu-tc` package). Please do not submit attempts which don't even compile.
- 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 stack or cabal), except for parsing libraries. Use Hoogle to search for functions with the types that you need.
- Copying solutions from the internet is not allowed.
- Textual answers to tasks can be included as comments in the source file submitted.
- We **strongly** 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. Submission is done through Blackboard.
- In the same page where you got this PDF file there is a *starting framework* file in which you should write the answers to the programming questions. Some datatypes and type signatures are already defined. The rest is up to you to define. The project consists of a library, where you'll do the assignments, and an executable, which you can use to test.

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 datatypes:

```
data DateTime = DateTime {date  :: Date
                         ,time  :: Time
                         ,utc   :: Bool}
  deriving (Eq,Ord)
data Date = Date {year :: Year
                 ,month :: Month
                 ,day  :: Day}
  deriving (Eq,Ord)
newtype Year = Year   {runYear  :: Int} deriving (Eq,Ord)
newtype Month = Month {runMonth :: Int} deriving (Eq,Ord)
newtype Day  = Day    {runDay   :: Int} deriving (Eq,Ord)
data Time = Time {hour   :: Hour
                 ,minute :: Minute
                 ,second :: Second}
  deriving (Eq,Ord)
newtype Hour = Hour {runHour :: Int} deriving (Eq,Ord)
newtype Minute = Minute {runMinute :: Int} deriving (Eq,Ord)
newtype Second = Second {runSecond :: Int} deriving (Eq,Ord)
```

1 (3 pt)**.** 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 (1.5 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 `https://en.wikipedia.org/wiki/Gregorian_calendar` for more information about the number of days per month. Make sure that you handle leap years (`https://en.wikipedia.org/wiki/Leap_year`) 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`. Also, you are allowed to use time libraries!

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. In `text`, `crlf` may be present, but always followed by a space. See the multiline.ics calendar for example, which has `crlf` in the summary. No extra whitespace is allowed anywhere in an event. Note that on Windows, the `readFile` function translates `"\r\n"` automatically to `"\n"`. The starting template contains a workaround.

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.

6 (4 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. Which elements of the concrete syntax are important, and which elements are irrelevant?

7 (4 pt)**.** Define the datatype `Token` and the following functions in order to be able to recognize valid calendars and output a structured representation of them in the form of the `Calendar` datatype:

```
scanCalendar :: Parser Char [Token]
parseCalendar :: Parser Token Calendar
```

The recognition is done in two steps: first, *lexing* (done in `scanCalendar`), which serves to ignore whitespace and build a list of *tokens* (smallest possible pieces of meaningful information). Then the second step is *parsing* (done in `parseCalendar`), which takes the list of tokens and builds a structured representation of valid calendars.

In the starting framework we have defined a `recognizeCalendar` function which runs the lexer and the parser in order.

8 (1.5 pt)**.** Define a printer

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

that generates a string representation from an abstract `Calendar` object. Each line may only be 42 characters long: If a `text` is too large, you should split it as in the multiline example. Since the printer might change the layout or order of properites, parsing a file and printing it will not always result in the original string. However, the other direction should hold: For any value `c` of type `Calendar`,

```
recognizeCalendar (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
```

9 (1.5 pt)**.** 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?

Note: For this exercise, you may ignore the *timeutc* flag, and assume that the local time is UTC.

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.

10 (2 pt)**.** Write a viewer that can visualize a calendar in ASCII graphics. It is recommended to use a *pretty printing* library for this, for example `boxes` (available at `http://hackage.haskell.org/package/boxes`). Write a function

```
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.