Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 1

Johan Jeuring

Friday, 12 December 2014, 08:30–10:30

Please keep in mind that there are often many possible solutions and that these example solutions may contain mistakes.

Questions

A group chat in Whatsapp looks as follows:

These group chats are stored on Whatsapp servers, and sent to the computers (phones, tablets) of the members of the groupchat. The internal representation of the above group chat (or at least a part of it) might look as follows:

```
GROUPCHAT
```

```
NAME    Whitmans Chat
MEMBERS Alice, France, Jack, Ned, Peter, Zissou
MESSAGES
  MESSAGE
    NAME Alice
    TIME 7:01 PM, March 14, 2013
    CONTENT I'll never forget this country. I love even the way it smells END
 MESSAGE
    NAME Jack
    TIME 11:40 PM, March 14, 2013
    CONTENT mountains.jpg END
 MESSAGE
    NAME Peter
    TIME 7:01 PM, March 14, 2013
    CONTENT Amazing END
 MESSAGE
    NAME Ned
    TIME 7:03 PM, March 14, 2013
    CONTENT Wow U+1F60D END
 MESSAGE
    NAME Zissou
    TIME 11:39 AM, March 15, 2013
    CONTENT http:\\www.willis.com\ END
```

In this exercise we look at the language of group chats.

A group chat consists of the keyword `GROUPCHAT` followed by:

- the keyword `NAME` followed by a name,
- the keyword `MEMBERS` followed by a non-empty list of members, separated by comma's,
- the keyword `MESSAGES` followed by a list of messages, where a message consists of the keyword `MESSAGE`, followed by the keyword `NAME`, a name, the keyword `TIME`, a time, the keyword `CONTENT`, some content, and the keyword `END`.

1 (12 points)**.** Give a concrete syntax (a context-free grammar) of this language for group chats. You may use nonterminal *Identifier* to recognise a single name, and *String* to recognise the content of a message (a string not containing "END"). •

Solution 1.

GroupChat → `"GROUPCHAT"` `"NAME"` *Name* `"MEMBERS"` *Members* `"MESSAGES"` *Message**
Name → *Identifier*$^+$
Members → *Name* `","` *Members* | *Name*

$$
\begin{array}{lll}
\textit{Message} & \rightarrow & \texttt{"MESSAGE"}\ \texttt{"NAME"}\ \textit{Name}\ \texttt{"TIME"}\ \textit{Time}\ \texttt{"CONTENT"}\ \textit{String}\ \texttt{"END"} \\
\textit{Time} & \rightarrow & \textit{Hours}\ \texttt{":"}\ \textit{Minutes}\ \textit{APM}\ \texttt{","}\ \textit{Month}\ \textit{Day}\ \texttt{","}\ \textit{Year} \\
\textit{Hours} & \rightarrow & \textit{Natural} \\
\textit{Minutes} & \rightarrow & \textit{Natural} \\
\textit{APM} & \rightarrow & \texttt{"AM"} \mid \texttt{"PM"} \\
\textit{Month} & \rightarrow & \texttt{"January"} \mid \ldots \mid \texttt{"December"} \\
\textit{Day} & \rightarrow & \texttt{"0"} \mid \ldots \mid \texttt{"31"} \\
\textit{Year} & \rightarrow & \textit{Natural}
\end{array}
$$

Here is the above example sentence:

$$
\begin{array}{lll}
\textit{ex} & = & \texttt{"GROUPCHAT NAME "} +\!\!+ \textit{name} +\!\!+ \texttt{" MEMBERS "} +\!\!+ \textit{members} +\!\!+ \texttt{" MESSAGES "} +\!\!+ \textit{messages} \\
\textit{name} & = & \texttt{"Whitmans Chat"} \\
\textit{members} & = & \texttt{"Alice, France, Jack, Ned, Peter, Zissou"} \\
\textit{messages} & = & \textit{message1} +\!\!+ \texttt{"\textbackslash n"} +\!\!+ \textit{message2} +\!\!+ \texttt{"\textbackslash n"} +\!\!+ \textit{message3} +\!\!+ \texttt{"\textbackslash n"} +\!\!+ \textit{message4} \\
\textit{message1} & = & \texttt{"MESSAGE NAME Alice TIME 7:01 PM, March 14, 2013 "} +\!\!+ \\
 & & \texttt{"CONTENT I'll never forget this country. I love even the way it smells END"} \\
\textit{message2} & = & \texttt{"MESSAGE NAME Jack TIME 11:40 PM, March 14, 2013 CONTENT mountains.jpg END"} \\
\textit{message3} & = & \texttt{"MESSAGE NAME Peter TIME 7:01 PM, March 14, 2013 CONTENT Amazing END"} \\
\textit{message4} & = & \texttt{"MESSAGE NAME Ned TIME 7:03 PM, March 14, 2013 CONTENT Wow U+1F60D END"} \\
\textit{message5} & = & \texttt{"MESSAGE NAME Zissou TIME 11:39 AM, March 15, 2013 "} +\!\!+ \\
 & & \texttt{"CONTENT http:\textbackslash\textbackslash www.willis.com END"}
\end{array}
$$

Marking
String instead of Identifier (or something like that) in Name: -1
Integers for the numbers (hours, minutes, days, years) in the language: -1
Comma at the end of the members list: -1
Members not separated by comma's: -1
Keywords forgotten: -1 per keyword
Requiring two digits where one digit is also possible: -1
String instead of list of months: -1
Letters instead of AM and PM: -1
No definition of Time or Date: -2 per concept

$\circ$

The abstract syntax of the language for groups chats is given by the following (data)types:

$$
\begin{array}{lll}
\textbf{type}\ \textit{GroupChat} & = & (\textit{Name}, \textit{Members}, \textit{Messages}) \\
\textbf{type}\ \textit{Name} & = & [\textit{String}] \\
\textbf{type}\ \textit{Members} & = & [\textit{Name}] \\
\textbf{type}\ \textit{Messages} & = & [\textit{Message}] \\
\textbf{type}\ \textit{Message} & = & (\textit{Name}, \textit{Time}, \textit{Content}) \\
\textbf{type}\ \textit{Time} & = & (\textit{Hours}, \textit{Minutes}, \textit{APM}, \textit{Date})
\end{array}
$$

```
type Hours    = Int
type Minutes  = Int
data APM      = AM | PM deriving Show
type Date     = (Day, Month, Year)
type Day      = Int
type Month    = Int
type Year     = Int
type Content  = String
```

2 (12 points)**.** Define a parser *pGroupChat* :: *Parser Char GroupChat* that parses sentences from the language of groupchats. •

Solution 2.

```
spaces       = greedy (satisfy isSpace)
tokensp s    = token s    <* spaces
identifiersp = identifier <* spaces
integersp    = integer    <* spaces

pGroupChat :: Parser Char GroupChat
pGroupChat = (,,)
    <$>  tokensp "GROUPCHAT"
    <*   tokensp "NAME"
    <*>  pName
    <*   tokensp "MEMBERS"
    <*>  pMembers
    <*   tokensp "MESSAGES"
    <*>  many pMessage

pName     :: Parser Char Name
pName      = many identifiersp

pMembers  :: Parser Char Members
pMembers   = listOf pName (tokensp ",")

pMessage  :: Parser Char Message
pMessage   = (\_n t c -> (n,t,c))
   <$>  tokensp "MESSAGE"
    <*   tokensp "NAME"
    <*>  pName
    <*   tokensp "TIME"
    <*>  pTime
    <*   tokensp "CONTENT"
    <*>  untilEND

untilEND  :: Parser Char Content
untilEND   = (const [] <$> tokensp "END")
   <<|>      ((:) <$> satisfy (const True) <*> untilEND)
```

```
pTime      :: Parser Char Time
pTime      = (λhours minutes apm month day year → (hours, minutes, apm, (day, month, year)))
   <$>       integersp
    <*       tokensp ":"
   <*>       integersp
   <*>       (AM <$ tokensp "AM" <|> PM <$ tokensp "PM")
    <*       tokensp ","
   <*>       pMonth
   <*>       integersp
    <*       tokensp ","
   <*>       integersp
months     = ["January","February","March","April","May","June"
             ,"July","August","September","October","November","December"
             ]
pMonth     :: Parser Char Month
pMonth     = foldr1 (<|>) (zipWith (λx y → const x <$> y) [1..12] (map tokensp months))
— Parser test cases
test       = pGroupChat ex
```

Marking
No abstract syntax for AM/PM: -1
Incorrect parsing of months, and computation of corresponding integer: -2
Minor errors: -1
Tokens not (or partially) parsed: -2(-1)
Does not follow the concrete syntax: -4(-2)
Types as value constructors: -2
String instead of a parser for strings: -2

∘

3 (12 points)**.** Whatsapp only offers chats and groupchats. I can imagine it would be useful to have a hierarchy of chats. For example, Utrecht University might start a chat, with seven subchats for the seven faculties. So people can chat at the university level, or at their own faculty level. Each faculty chat consists of faculty wide chats, but also of chats at the various departments of the faculty, and so on. In this exercise I encode a slightly simplified version of this situation. A *GroupChatTree* is either a single *GroupChat*, or it collects a number of *GroupChatTree*'s under a particular name.

```
data GroupChatTree = Fork Name [GroupChatTree]
                   | Single GroupChat
```

Define the algebra type, and the *fold* for the datatype *GroupChatTree*. You may assume that the type *GroupChat* is a constant type such as *Int* and *String*, that is, you don't have to define a *fold* for *GroupChat*. •

Solution 3.

```
type GroupChatTreeAlgebra a = (Name -> [a] -> a, GroupChat -> a)
foldGroupChatTree :: GroupChatTreeAlgebra a -> GroupChatTree -> a
foldGroupChatTree (fork, single)    = fold where
  fold (Fork name groupchattrees) = fork name (map fold groupchattrees)
  fold (Single groupchat)         = single groupchat
— some examples to test the solutions
gcCS :: GroupChat
gcCS = (["Computer", "Science"]
       , [["Johan", "Jeuring"], ["Joao", "Pizani"]]
       , [(["Johan", "Jeuring"], atime, "Hi!"), (["Johan", "Jeuring"], atime, "there")]
       )
gcW  :: GroupChat
gcW  = (["Mathematics"]
       , [["Rob", "Bisseling"], ["Jan", "Hoogendijk"]]
       , [(["Rob", "Bisseling"], atime, "Hello"), (["Jan", "Hoogendijk"], atime, "world!")]
       )
gcWI :: GroupChatTree
gcWI = Fork ["WI"] [Single gcCS, Single gcW]
atime = (7, 24, PM, (27, 11, 2014))
```

Marking
Algebra not a **type**: -2
Algebra completely wrong: -4
No type argument of algebra: -2
Two type arguments to the algebra: -2
Algebra just a single type: -1
Not using parameter in list of recursive calls in algebra: -3
Recursive parameter instead of *GroupChat*: -2
Type *fold* incorrect: -2
No parens in pattern matching: -1
No recursive def of *fold*: -6
No map in recursive def of *fold*: -3
Resulttype of *fold* not a type variable: -2
Minor mistakes: -1
fold... instead of fold in the recursive call (without the algebra argument): -2

○

4 (9 points)**.** Define a function *nrOfMessagesGCT* :: *GroupChatTree* → *Int* that returns the number of messages present in a *GroupChatTree*. Define function *nrOfMessagesGCT* as a *fold* on the datatype *GroupChatTree*.

•

Solution 4.

```
nrOfMessagesGCT :: GroupChatTree → Int
nrOfMessagesGCT = foldGroupChatTree (\_xs → sum xs, nrOfMessagesGC)
nrOfMessagesGC                  :: GroupChat → Int
nrOfMessagesGC (_, _, messages) = length messages
test1 = nrOfMessagesGCT gcWI
```

Marking
Using *foldGroupChatTree* but no definition of *fork* and *single*: +2 (-7)
Using *foldGroupChatTree* but incorrect definition of *fork* and *single*: +2(-7)
Using *Fork n xs* as argument to *fork* (interpreting arguments correctly): -2
Algebra and tree argument order swapped: -1
Double recursion: -5
No *fold* but explicitly recursive definition: -6
Wrongly typed fork function: -3
const 1 instead of *nrOfMessagesGC*: -3

◦

5 (9 points)**.** Define a function *messagesMember* :: *GroupChatTree* → *Member* → *Messages* that returns the messages of a particular member in a *GroupChatTree*. Define function *messagesMember* as a *fold* on the datatype *GroupChatTree*. •

Solution 5.

```
messagesMember :: GroupChatTree → Name → Messages
messagesMember = foldGroupChatTree messagesMemberAlgebra
  where messagesMemberAlgebra = (\_xs m → concat (map ($m) xs)
                                ,λ(_, _, ms) → λm → filter (λ(n, t, c) → n == m) ms)
test2 = messagesMember gcWI ["Johan","Jeuring"]
```

Marking
Using *foldGroupChatTree* but no definition of *fork* and *single*: +2 (-7)
Using *foldGroupChatTree* but incorrect definition of *fork* and *single*: +2(-7)
Not passing the name on recursively: -3
Forget the dollar when passing on the name recursively: -1
Member name and recursive results swapped in algebra: -1
Member name and recursive results swapped in fold: -3
Double recursion: -5

◦

6 (9 points).

(a) Give an example of a grammar that can be left-factorized
Solution 6.

$S \to \mathtt{a}\ S\ \mathtt{b} \mid \mathtt{a}\ S\ \mathtt{a} \mid \mathtt{a}$

Marking

○

(b) Left-factorize this grammar
Solution 6.

$S \to \mathtt{a}\ T$
$T \to S\ U \mid \varepsilon$
$U \to \mathtt{b} \mid \mathtt{a}$

Marking

○

(c) Explain why left-factorization may be a useful grammar transformation
Solution 6.
When you translate a context-free grammar that can be left-factorized into a combinator parser, the parser performs the same actions twice for the part that can be left-factorized. If the parser is of a particular recursive structure, this might lead to an exponential time required for parsing.

Marking

○

•

7 (9 points).

(a) Give an example of a grammar that is left-recursive
Solution 7.

$S \to SS \mid \mathtt{a}$

Marking

○

(b) Remove this left-recursion

Solution 7.

$$S \to \mathtt{a} \mid \mathtt{a}\ Z$$
$$Z \to S \mid S\ Z$$

Marking

○

(c) Explain why removing left-recursion may be a useful grammar transformation

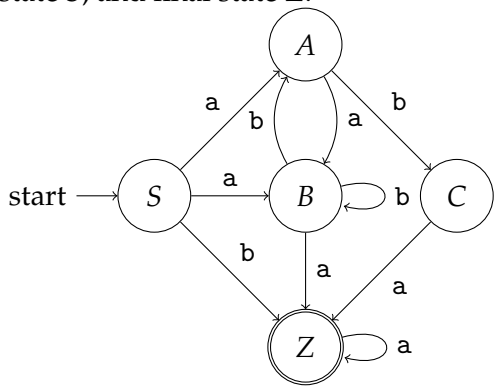
Solution 7. If you directly translate a left-recursive context-free grammar to a parser combinator, the parser combinator does not terminate on any input. Removing left-recursion removes this cause of non-termination.

Marking
Loop in the grammar instead of the parser: -1
Ensures termination instead of avoids non-termination: -1

○

•

Consider the following NFA (Nondeterministic Finite-state Automaton), with start state S, and final state Z.



8 (6 points)**.** Construct a regular grammar with the same language. •

Solution 8.

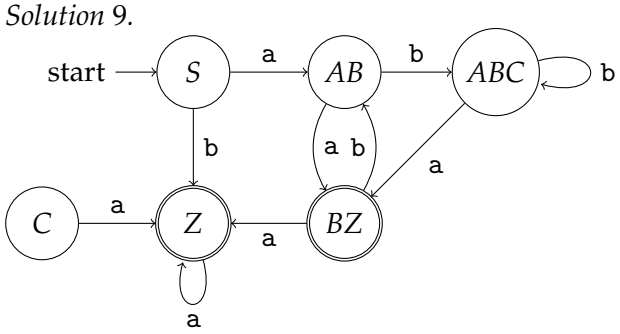
$S \to \mathtt{a}\ A$
$S \to \mathtt{a}\ B$
$S \to \mathtt{b}\ Z$
$A \to \mathtt{a}\ B$
$A \to \mathtt{b}\ C$
$B \to \mathtt{b}\ A$
$B \to \mathtt{b}\ B$
$B \to \mathtt{a}\ Z$
$C \to \mathtt{a}\ Z$
$Z \to \mathtt{a}\ Z$
$Z \to \varepsilon$

Marking
No productions for Z: -2
Production $Z \to \varepsilon$ forgotten: -1

○

9 (6 points)**.** Construct a DFA (Deterministic Finite-state Automaton) with the same language (you may draw a DFA). •

Solution 9.



Note that state C is not reachable from the start state, so it may safely be removed.

Marking
No final states: -1
Minor errors (single forgotten transition): -1
Still non-deterministic: -4

○

10 (6 points)**.** Suppose we have two context-free grammars $G_1 = (T_1, N_1, R_1, S_1)$ and $G_2 = (T_2, N_2, R_2, S_2)$, where the intersection of N_1 and N_2 is empty. Define $G = (T_1 \cup T_2, N_1 \cup N_2 \cup \{S\}, R_1 \cup R_2 \cup \{S \to S_1\ S_2\}, S)$, where S is the new startsymbol.

(a) What is the language of G?

(b) This construction does not work for regular grammars. Why not?

(c) Describe the construction of a grammar with the same language as G, which is regular if both G_1 and G_2 are regular.

•

Solution 10.

(a) $L\,(G) = \{\, x\, y \mid x \leftarrow L\,(G_1), y \leftarrow L\,(G_2)\,\}$.

(b) The resulting grammar is not regular, since it is of the form $S \to S_1\ S_2$, and hence it has two instead of one non-terminals in a right-hand side of a production.

(c) See Theorem 8.10 in the lecture notes: we obtain a regular grammar for G if we replace in G_1 every production of the form $T \to \mathtt{x}$ and $T \to \varepsilon$ by $T \to \mathtt{x}\ S_2$ and $T \to S_2$, respectively.

Marking
$G = G_1\ G_2$: -1
Using automata instead of productions in c): -1

◦