Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Midterm Exam INFOB3TC

Jurriaan Hage

Tuesday, December 17th, 2019, 11.00-13.00

Preliminaries

- The exam consists of 8 pages (including this page). Please verify that you got all the pages.
- Fill out the answers **on the exam itself**.
- Write your **name** and **student number** here:

- For open questions, the maximum score is stated at the top of each question. The total number of points you can get is 90. You can compute your exam grade by dividing by 9 and rounding to one decimal after the decimal point.
- Try to give simple and concise answers. Write readable text. You may use Dutch or English.
- Answer multiple choice questions by writing the correct alternative in the box below the question. Sometimes multiple answers are correct: in that case you need to give the *best* answer.
- When writing grammar and language constructs, you may use any set, sequence, or language operations covered in the lecture notes.
- When writing Haskell code, you may use Prelude functions and functions from the following modules: *Data.Char*, *Data.List*, *Data.Maybe*, and *Control.Monad*. Unless indicated otherwise, you may use all the parser combinators from the `uu-tc` package. If you are in doubt whether a certain function is allowed, please ask.

Good luck!

1. An important part of database systems is the construction of forms. In such forms we have input fields that often satisfy formatting rules. For example, a Dutch zip code field should contain six characters, the first four of which must be digits, and the last two must be letters. In such systems, you can often describe the formatting of such fields with a formatting string, and the system will guarantee it will parse that field according to the specified format. To keep things manageable we use the following formatters (each formatter is a character): '0' stands for any digit, 'L' stands for any letter (lower or upper case), but will also convert all lower case characters it parses to upper case, similarly we have "l" for the other way around.

For Dutch zip codes, the format string would be "0000LL". If we apply the parser *zipCodeP* that follows this format string to the input "1900Ab" it will succeed and return the string "1900AB", having converted the acceptable character 'b' to the capital 'B'. The parser fails on inputs like "My Name", "ABCD99", and also "1900 AB".

Now, write a function *parseFormatted* :: *String* $\rightarrow$ *Parser Char String*, that given a format string, returns the parser that parses following the rules above. This will allow us to define

zipCodeP :: *Parser Char String*
zipCodeP = *parseFormatted* `"0000LL"`

You may use any parser combinator from the `uu-tc` package.

•

. . . / **10**

2. To increase the power of the formatters, someone suggests to add the character '9' which behaves like an optional '0', a digit may be there or not. Is support for '9' easy to add? Explain your answer. •

. . . /5

3. A grammar has the following productions:

$$T \quad \rightarrow \quad \texttt{x}T\texttt{y} \mid xy$$

If we add a single production to this grammar, we can derive the sentence `xxyyxxyy`. Which of the following productions could we add?

a) $T \rightarrow \texttt{x}T\texttt{yy}$

b) $T \rightarrow \texttt{y}T\texttt{x}$

c) $T \rightarrow TTT$

d) None of the above answers are correct.

•

. . . /5

4. Suppose we have a parser *pExpr* :: *Parser Char Expr*, where the datatype *Expr* has a constructor *Let Identifier Expr Expr*. What is the type of the following parser combinator?

```
pDecl = Let <$ token "let"
            <* identifier
            <* symbol '='
           <*> pExpr
            <* token "in"
           <*> pExpr
```

a) *Parser Char* $(Identifier \rightarrow Expr \rightarrow Expr \rightarrow Expr)$

b) *Parser Char* $((Identifier, Expr, Expr) \rightarrow Expr)$

c) *Parser Char Expr*

d) The parser *pDecl* is type incorrect

•

. . . /5

5. Given is the following context-free grammar:

$$\begin{array}{lcl} S & \to & \texttt{ddS} \mid \texttt{db} \mid \texttt{b}A\texttt{c} \mid \texttt{b}A\texttt{d} \\ A & \to & \texttt{aS} \mid \texttt{a}A \end{array}$$

Give a left-factorised version of this context-free grammar. •

. . . / **10**

6. Given is the following context-free grammar for boolean formulas:

$B \to B \mathbin{/\!\backslash} B$
$B \to B \mathbin{\backslash\!/} B$
$B \to \texttt{(}\ B\ \texttt{)}$
$B \to \mathit{ff}$
$B \to \mathit{tt}$

This grammar is ambiguous and does not correctly enforce the priority that conjunction ('and') $/\!\backslash$ has over disjunction ('or') $\backslash\!/$. Give a non-ambiguous context-free grammar in which the above priorities have been correctly introduced. •

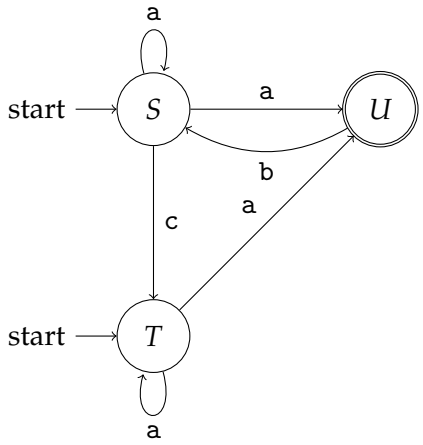
. . . / **10**

7. Construct an NFA for the language L over the alphabet $A = \{a, b\}$ containing all sequences over A^*, except those that contain the contiguous subsequence *bbb*. So, *abba* and *bababb* are in L, but *bbb* and *abbaaabbbb* are not.

•

. . . /**10**

Consider the following NFA, with start states S and T, and final state U.



8. Construct a DFA (Deterministic Finite Automaton) that accepts the same language. You should use the subset construction, but should exclude states that are unreachable from the start state or that can never lead to a final state. It is *essential* that in your answer it is clear what each of your DFA states corresponds to in the original NFA; do not forget to indicate the start and end states of the DFA. •

. . . /**12**

We are modelling a situation in which group chats can be organized in a hierarchical fashion. For example, a company may consist of four (in fact, any number of) divisions, and it can initiate a group chat in such a way that only members of the same division can chat with each other. A concrete group chat consists of a list of messages each message coming with the name of the member who sent it. For this, the following Haskell types have been defined:

data *GroupChatterbox* = *Fork Name* [*GroupChatterbox*] | *Single Name GroupChat*
type *GroupChat* = [(*Member*, *Message*)]

where the datatype *GroupChat* stores all relevant information about a single chat. The actual form of types like *Name*, *Message* and *Members* is not that relevant here, but if it helps think of them as *String*s.

9. Define the algebra type for the datatype *GroupChatterbox*. •

. . . /**6**

10. Now, give the type and the definition of the fold function associated with the datatype *GroupChatterbox*. •

. . . /**8**

11. Define a function *members* :: *GroupChatterbox* $\rightarrow$ [*Member*] that returns a list of *Member*s that participate in a group chat (with or without duplicate member names). Define it using a fold on the datatype *GroupChatterbox*. •

. . . /**5**

12. Define the function *twotimers* :: *GroupChatterbox* $\rightarrow$ [*Member*] that returns only those members that have sent messages in more than one *GroupChat*. If necessary, give a new definition of *members* to make this possible. Hint: use the function *nub* to remove duplicates from a list. •

. . . /**4**