Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Exam

Andres Löh

Monday, 7 December 2009, 09:00–12:00

Preliminaries

- The exam consists of 4 pages (including this page). Please verify that you got all the pages.
- Write your **name** and **student number** on all submitted work. Also include the total number of separate sheets of paper.
- For each task, the maximum score is stated. The total amount of points you can get is 100.
- Try to give simple and concise answers. Write readable. Do not use pencils or pens with red ink.
- You may answer questions in Dutch or English.
- When writing Haskell code, you may use Prelude functions and functions from the *Data.List*, *Data.Maybe*, *Data.Map*, *Control.Monad* modules. Also, you may use all the parser combinators from the `uu-tc` package. If in doubt whether a certain function is allowed, please ask.

Good luck!

1

Context-free grammars

1 (10 points). Let $A = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Give context-free grammars for the following languages over the alphabet A:

(a) $L_1 = \{w \mid w \in A^*, \#(\mathtt{x}, w) \geqslant 3\}$

(b) $L_2 = \{w \mid w \in A^*, \#(\mathtt{x}, w) < 3\}$

(c) $L_1 \cap L_2$

Here, $\#(c, w)$ denotes the number of occurrences of a terminal c in a word w. •

Grammar analysis and transformation

Consider the following context-free grammar G over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with start symbol S:

$$S \rightarrow S\mathtt{a}S\mathtt{a}$$
$$S \rightarrow S\mathtt{a}S\mathtt{b}S\mathtt{a}$$
$$S \rightarrow \mathtt{b}$$

2 (10 points). For each of the following words, answer the question whether it is in $L(G)$. If yes, give a parse tree. If not, argue informally why the word cannot be in the language.

(a) `babababba`

(b) `bababababa` •

3 (11 points). Simplify the grammar G by transforming it in steps. Perform as many as possible of the following transformations: removal of left recursion, left factoring, and removal of unreachable productions. •

Alternative definitions of parser combinators

In the following tasks, you are not supposed to make use of the internal implementation of parser combinators.

4 (4 points). Define $(<\$>)$ in terms of *succeed* and $(<*>)$. •

5 (5 points). Let

$$anySymbol :: Parser\ s\ s$$

be a parser that consumes any single symbol in the input and returns it. The parser only fails if the end of the input has been reached. Define

$$symbol :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s$$

in terms of *anySymbol*, *succeed*, $(\gg\!=)$ and *empty*. •

Combinators for permutations

6 (4 points)**.** Write a parser combinator

$$perms2 :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ (a, b)$$

such that *perms2 p q* parses *p* followed by *q*, or *q* followed by *p*, and returns the results in a pair. Pay attention to the order in which the results are returned! •

7 (10 points)**.** Now write a parser combinator

$$perms3 :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ c \rightarrow Parser\ s\ (a, b, c)$$

where *perm3 p q r* parses any permutation of *p*, *q* and *r*.

If you find a way of improving the efficiency of the resulting parser, explain (for example, in terms of the underlying grammar) what has to be done. It is not necessary to give the resulting parser, however. •

Parsing logical propositions

Here is a grammar for logical propositions with start symbol *P*:

$$\begin{array}{rl} P \rightarrow & P \wedge P \\ | & P \vee P \\ | & P \Rightarrow P \\ | & \neg\, P \\ | & Ident \\ | & \texttt{(}\ P\ \texttt{)} \\ | & \texttt{1} \\ | & \texttt{0} \end{array}$$

Propositions can be composed from the constants true (1) and false (0) by using negation, conjunction, disjunction and implication, and parentheses for grouping.

Furthermore, propositions can contain variables – the nonterminal *Ident* represents an identifier consisting of one or more letters.

A corresponding abstract syntax in Haskell is:

$$\begin{array}{lll} \mathbf{data}\ P = & And & P\ P \\ | & Or & P\ P \\ | & Implies & P\ P \\ | & Not & P \\ | & Var & String \\ | & Const & Bool \end{array}$$

8 (10 points)**.** Resolve the operator priorities in the grammar as follows: negation ($\neg$) binds stronger that implication ($\Rightarrow$), which in turn binds stronger than conjunction ($\wedge$), which in turn binds stronger than disjunction ($\vee$). Furthermore, implication associates to the right, whereas conjunction and disjunction associate to the left. Give the resulting grammar. •

9 (11 points)**.** Give a parser that recognizes the grammar from Task 8 and produces a value of type P:

$$parseP :: Parser\ Char\ P$$

You can assume that the symbols $\neg$, $\Rightarrow$, $\wedge$, and $\vee$ are just characters. You can use *chainl* and *chainr*, but if you want more advanced abstractions such as *gen* from the lecture notes, you have to define them yourself. You may assume that spaces are not allowed in the input. •

10 (10 points)**.** Define an algebra type and a fold function for type P. •

11 (10 points)**.** Using the algebra and fold (or alternatively directly), define an evaluator for propositions:

$$evalP :: P \to Env \to Bool$$

The environment of type *Env* should map free variables to Boolean values. You can either use a list of pairs or a finite map with the following interface to represent the environment:

$$
\begin{array}{ll}
\textbf{data}\ Map\ k\ v & \text{— abstract type, maps keys of type } k \text{ to values of type } v \\
empty & :: Map\ k\ v \\
(!) & :: Ord\ k \Rightarrow Map\ k\ v \to k \to v \\
insert & :: Ord\ k \Rightarrow k \to v \to Map\ k\ v \to Map\ k\ v \\
delete & :: Ord\ k \Rightarrow k \to Map\ k\ v \to Map\ k\ v \\
member & :: Ord\ k \Rightarrow k \to Map\ k\ v \to Bool \\
fromList & :: Ord\ k \Rightarrow [(k, v)] \to Map\ k\ v
\end{array}
$$

•

12 (5 points)**.** Implement a tautology checker for propositions of type P:

$$tautology :: P \to Bool$$

A proposition is a tautology if and only if it evaluates to *True* regardless of the values of any of its free varaibles.

It may be helpful to use the following function *assignments* that produces a list of all possible Boolean assignments for a list of identifiers:

$$
\begin{array}{l}
assignments :: [String] \to [[(String, Bool)]] \\
assignments\ []\ \ \ \ \ \ \ = [[]] \\
assignments\ (n : ns) = [(n, x) : xs \mid x \leftarrow [True, False], xs \leftarrow assignments\ ns]
\end{array}
$$

You can use *evalP* – even if you have not implemented it – in the definition of *tautology*. •

13 (meta question)**.** How many out of the 100 possible points do you think you will get for this exam? •