Department of Information and Computing Sciences
Utrecht University

# INFOB3TC – Solutions for the Exam

Andres Löh

Monday, 7 December 2009, 09:00–12:00

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

## 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 $w$. •

*Solution* 1.

(a) Without abbreviations:

$$\begin{aligned} S &\to C\ \mathtt{x}\ C\ \mathtt{x}\ C\ \mathtt{x}\ C \\ C &\to \varepsilon \mid X\ C \\ X &\to \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \end{aligned}$$

With EBNF-abbreviations:

$$\begin{aligned} S &\to X^*\ \mathtt{x}\ X^*\ \mathtt{x}\ X^*\ \mathtt{x}\ X^* \\ X &\to \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \end{aligned}$$

(b) Without abbreviations:

$$\begin{aligned} S &\to C \mid C\ \mathtt{x}\ C \mid C\ \mathtt{x}\ C\ \mathtt{x}\ C \\ C &\to \varepsilon \mid Y\ C \\ Y &\to \mathtt{y} \mid \mathtt{z} \end{aligned}$$

With EBNF-abbreviations:

$$
\begin{aligned}
S &\rightarrow Y^*\ \mathtt{x}?\ Y^*\ \mathtt{x}?\ Y^* \\
Y &\rightarrow \mathtt{y} \mid \mathtt{z}
\end{aligned}
$$

(c) The intersection of the two languages is the empty language. A grammar for the empty language is, for example, the empty grammar (no productions) or

$$S \rightarrow S$$

where no derivations of terminal strings from the start symbol are possible.

○

## Grammar analysis and transformation

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

$$
\begin{aligned}
S &\rightarrow S\mathtt{a}S\mathtt{a} \\
S &\rightarrow S\mathtt{a}S\mathtt{b}S\mathtt{a} \\
S &\rightarrow \mathtt{b}
\end{aligned}
$$

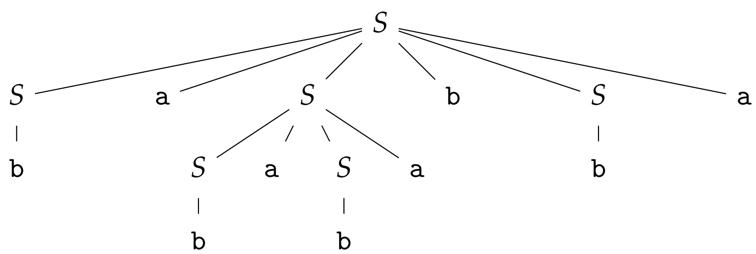
**2** (10 points)**.** For each of the following words, answer the question whether it is in $L(G)$. If yes, give a parse tree. If not, argue informally why the word cannot be in the language.

(a) `bababbba`

(b) `babababa` ●

*Solution* 2.

(a) The word `bababbba` is in $L(G)$, as can be witnessed by the following parse tree:




(b) The word `babababa` is not in $L(G)$. It contains an odd number of *a*s, but it is easy to prove via induction on the derivations in grammar $G$ that any word derived from $G$ must have an even number of *a*s.

It is also possible to argue via the length of the word (length 10 is only possible by applying the first production three times, and then derive `b` for all remaining occurrences of $S$; one can argue that this does not produce the desired word) or by analyzing derivation sequences and argue that all sufficiently long words must either contain the substring `aa` or `bb`. However, these arguments are more complicated than going via the number of `a`s and it is easier to forget a case.

○

**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. •

*Solution* 3. This is the original grammar:

$$\begin{aligned}
&S \to S\mathtt{a}S\mathtt{a}\\
&S \to S\mathtt{a}S\mathtt{b}S\mathtt{a}\\
&S \to \mathtt{b}
\end{aligned}$$

We can first left-factor the grammar:

$$\begin{aligned}
&S \to S\mathtt{a}SR\\
&S \to \mathtt{b}\\
&R \to \mathtt{a}\\
&R \to \mathtt{b}S\mathtt{a}
\end{aligned}$$

Now, we remove left recursion:

$$\begin{aligned}
&S \to \mathtt{b}Z?\\
&Z \to \mathtt{a}SRZ?\\
&R \to \mathtt{a}\\
&R \to \mathtt{b}S\mathtt{a}
\end{aligned}$$

Of course, it is possible to remove left recursion first and perform left factoring later. We then get

$$\begin{aligned}
&S \to \mathtt{b}Z?\\
&Z \to \mathtt{a}S\mathtt{a}Z? \mid \mathtt{a}S\mathtt{b}S\mathtt{a}Z?
\end{aligned}$$

after removal of left recursion, and left factoring then yields:

$$\begin{aligned}
&S \to \mathtt{b}Z?\\
&Z \to \mathtt{a}SR\\
&R \to \mathtt{a}Z? \mid \mathtt{b}S\mathtt{a}Z?
\end{aligned}$$

If desired, it is possible to perform more operations, but that was not expected.

○

## 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 $(<*>)$. •

*Solution* 4. The map function on parsers can be defined as a derived combinator simply as follows:

$$
\begin{aligned}
&(<\$>) :: (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \\
&f <\$> p = \mathit{succeed}\ f <*> p
\end{aligned}
$$

○

**5** (5 points)**.** Let

$$\mathit{anySymbol} :: \mathit{Parser}\ s\ s$$

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

$$\mathit{symbol} :: \mathit{Eq}\ s \Rightarrow s \to \mathit{Parser}\ s\ s$$

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

*Solution* 5.

$$\mathit{symbol}\ x = \mathit{anySymbol} \gg\!= \lambda y \to \textbf{if}\ x == y\ \textbf{then}\ \mathit{succeed}\ y\ \textbf{else}\ \mathit{empty}$$

Of course, we can write *return* instead of *succeed*. We can even use **do**-notation:

$$
\begin{aligned}
\mathit{symbol}\ x = \textbf{do}\ & \\
& y \leftarrow \mathit{anySymbol} \\
& \textbf{if}\ x == y\ \textbf{then}\ \mathit{return}\ y\ \textbf{else}\ \mathit{empty}
\end{aligned}
$$

Although returning $y$ is preferable over returning $x$ (exercise: why?), returning $x$ is ok as a solution.

○

## Combinators for permutations

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

$$\mathit{perms2} :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ (a, b)$$

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! •

*Solution* 6. Written such that the symmetry becomes most obvious:

$$\begin{aligned} \mathit{perms2}\ p\ q \ &= \ (\lambda x\ y \to (x, y)) <\$> p <*> q \\ &<|> (\lambda y\ x \to (x, y)) <\$> q <*> p \end{aligned}$$

The main difficulty is that we have to reorder the results so that the types match.

$\circ$

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

$$\mathit{perms3} :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ c \to \mathit{Parser}\ s\ (a, b, c)$$

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. $\bullet$

*Solution* 7. The first approach is probably the following:

$$\begin{aligned} \mathit{perms3}\ p\ q\ r \ &= \ (\lambda x\ y\ z \to (x, y, z)) <\$> p <*> q <*> r \\ &<|> (\lambda x\ z\ y \to (x, y, z)) <\$> p <*> r <*> q \\ &<|> (\lambda y\ x\ z \to (x, y, z)) <\$> q <*> p <*> r \\ &<|> (\lambda y\ z\ x \to (x, y, z)) <\$> q <*> r <*> p \\ &<|> (\lambda z\ x\ y \to (x, y, z)) <\$> r <*> p <*> q \\ &<|> (\lambda z\ y\ x \to (x, y, z)) <\$> r <*> q <*> p \end{aligned}$$

However, this parser is in clear need for left-factoring. The grammar corresponding to the parser above is:

$$\begin{aligned} S \to\ & P\ Q\ R \\ |\ & P\ R\ Q \\ |\ & Q\ P\ R \\ |\ & Q\ R\ P \\ |\ & R\ P\ Q \\ |\ & R\ Q\ P \end{aligned}$$

which can be left-factored to

$$\begin{aligned} S &\to P\ X \mid Q\ Y \mid R\ Z \\ X &\to Q\ R \mid R\ Q \\ Y &\to P\ R \mid R\ P \\ Z &\to P\ Q \mid Q\ P \end{aligned}$$

If someone wrote this, it was sufficient. But now, $X$, $Y$, and $Z$ are permutations of two elements, so it is relatively easy to write the parser in an efficient, left-factored way:

$$\begin{aligned} \mathit{perms3}\ p\ q\ r \ &= \ (\lambda x\ (y, z) \to (x, y, z)) <\$> p <*> \mathit{perms2}\ q\ r \\ &<|> (\lambda y\ (x, z) \to (x, y, z)) <\$> q <*> \mathit{perms2}\ p\ r \\ &<|> (\lambda z\ (x, y) \to (x, y, z)) <\$> r <*> \mathit{perms2}\ p\ q \end{aligned}$$

$\circ$

## Parsing logical propositions

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

$$
\begin{aligned}
P \rightarrow\ & P \wedge P \\
\mid\ & P \vee P \\
\mid\ & P \Rightarrow P \\
\mid\ & \neg\, P \\
\mid\ & \mathit{Ident} \\
\mid\ & \texttt{(}\ P\ \texttt{)} \\
\mid\ & \texttt{1} \\
\mid\ & \texttt{0}
\end{aligned}
$$

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:

$$
\begin{aligned}
\textbf{data}\ P = \ & \mathit{And} && P\ P \\
\mid\ & \mathit{Or} && P\ P \\
\mid\ & \mathit{Implies} && P\ P \\
\mid\ & \mathit{Not} && P \\
\mid\ & \mathit{Var} && \mathit{String} \\
\mid\ & \mathit{Const} && \mathit{Bool}
\end{aligned}
$$

**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. •

*Solution* 8. We split $P$ into several nonterminals, corresponding to the different priority levels:

$$
\begin{aligned}
P &\rightarrow P_1 \\
P_1 &\rightarrow P_1 \vee P_2 \mid P_2 \\
P_2 &\rightarrow P_2 \wedge P_3 \mid P_3 \\
P_3 &\rightarrow P_4 \Rightarrow P_3 \mid P_4 \\
P_4 &\rightarrow \neg\, P_4 \mid \mathit{Ident} \mid \texttt{(}\ P\ \texttt{)} \mid \texttt{1} \mid \texttt{0}
\end{aligned}
$$

There should not be any surprises in the resulting grammar. I think it's actually unusual to have implication bind as strong. After placing the exam it occurred to me that it probably makes more sense to have it bind weakest. But then again, it does not make any difference for the difficulty of the assignment.

○

**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. •

*Solution* 9. This is a rather direct transcription using *chainl* and *chainr*:

```
parseP = p1
p1     = chainl p2 (Or      <$ symbol '∨' )
p2     = chainl p3 (And     <$ symbol '∧' )
p3     = chainr p4 (Implies <$ symbol '⇒')
p4     = Not <$ symbol '¬' <*> p4
     <|> Var <$> some (satisfy isLetter)
     <|> parenthesised parseP
     <|> Const True  <$ symbol '1'
     <|> Const False <$ symbol '0'
```

Using *identifier*, $many_1$ or $greedy_1$ for the *Var* case is also ok.

○

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

*Solution* 10. We just apply the systematic translation:

```
type PAlgebra r = ( r → r → r,    — And
                    r → r → r,    — Or
                    r → r → r,    — Implies
                    r → r,        — Not
                    String → r,   — Var
                    Bool → r)     — Const
foldP :: PAlgebra r → P → r
foldP (and, or, implies, not, var, const) = f
  where f (And     x1 x2) = and     (f x1) (f x2)
        f (Or      x1 x2) = or      (f x1) (f x2)
        f (Implies x1 x2) = implies (f x1) (f x2)
        f (Not     x    ) = not     (f x)
        f (Var     x    ) = var     x
        f (Const   x    ) = const   x
```

No surprises here.

○

**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:

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

•

*Solution* 11. We assume

$$\textbf{type}\ Env = Map\ String\ Bool$$

The algebra is similar to the evaluator for expressions discussed in the lectures:

$$
\begin{array}{l}
evalAlgebra :: PAlgebra\ (Env \to Bool) \\
evalAlgebra = (\lambda x_1\ x_2\ e \to x_1\ e\ \&\&\ x_2\ e, \\
\qquad\qquad\qquad \lambda x_1\ x_2\ e \to x_1\ e \parallel x_2\ e, \\
\qquad\qquad\qquad \lambda x_1\ x_2\ e \to not\ (x_1\ e) \parallel x_2\ e, \\
\qquad\qquad\qquad \lambda x\quad\ e \to not\ (x\ e), \\
\qquad\qquad\qquad \lambda x\quad\ e \to e\,!\,x, \\
\qquad\qquad\qquad \lambda x\quad\ e \to x) \\
evalP = foldP\ evalAlgebra
\end{array}
$$

The environment never changes, so defining

$$
\begin{array}{l}
evalAlgebra :: Env \to PAlgebra\ Bool \\
evalAlgebra\ e = ((\&\&), (\parallel), (\lambda x_1\ x_2 \to not\ x_1 \parallel x_2), not, (e!), id)
\end{array}
$$

is simpler and ok as well.

∘

**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:

$$
\begin{aligned}
&\mathit{assignments} :: [\mathit{String}] \to [[(\mathit{String}, \mathit{Bool})]] \\
&\mathit{assignments}\ []\qquad\quad = [[]] \\
&\mathit{assignments}\ (n : ns) = [(n, x) : xs \mid x \leftarrow [\mathit{True}, \mathit{False}], xs \leftarrow \mathit{assignments}\ ns]
\end{aligned}
$$

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

*Solution* 12. We need a way to discover the free variables in $P$:

$$
\begin{aligned}
&\mathit{freeVarsAlgebra} :: \mathit{PAlgebra}\ [\mathit{String}] \\
&\mathit{freeVarsAlgebra} = ((+\!\!+), (+\!\!+), (+\!\!+), \mathit{id}, (:[]), \mathit{const}\ [])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{freeVars} :: P \to [\mathit{String}] \\
&\mathit{freeVars} = \mathit{foldP}\ \mathit{freeVarsAlgebra}
\end{aligned}
$$

This algebra simply collects all the variables. Now we can define the tautology checker:

$$
\mathit{tautology}\ p = \mathit{all}\ (\lambda e \to \mathit{evalP}\ p\ (\mathit{fromList}\ e))\ (\mathit{assignments}\ (\mathit{freeVars}\ p))
$$

For all the assignments corresponding to the free variables, the proposition has to evaluate to true. The function *all* is a standard function from the prelude and can be defined as follows:

$$
\begin{aligned}
&\mathit{all} :: (a \to \mathit{Bool}) \to [a] \to \mathit{Bool} \\
&\mathit{all}\ p = \mathit{and}\ .\ \mathit{map}\ p
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{and} :: [\mathit{Bool}] \to \mathit{Bool} \\
&\mathit{and} = \mathit{foldr}\ (\&\&)\ \mathit{True}
\end{aligned}
$$

○

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

*Solution* 13. I ask this question because it is the first time I'm setting an exam for the course, and I'm genuinely interested how difficult the students perceive the exam to be, and how good the students are in judging their own performance. Obviously, the answer to this question has no relevance for the final result. ○