Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 1

Sean Leather

Thursday, 15 December 2010, 08:30–10:30

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

Questions

Context-Free Grammars

1 (20 points)**.** Consider the following language definitions:

(a) $L_1 = \{ a\,w \mid a \in A^2 \wedge w \in A^* \wedge |w| > 0 \}$ where $A = \{ \mathtt{t}, \mathtt{u}, \mathtt{v} \}$

(b) L_2 is the language defined by the following grammar over the alphabet $\{ \mathtt{a}, \mathtt{b}, \mathtt{z} \}$:

$$S \to R\,\mathtt{a} \mid S\,\mathtt{a} \mid \mathtt{z}$$
$$R \to \mathtt{b}\,R \mid \mathtt{b}\,S$$

(c) $L_3 = \{ \texttt{<}t\texttt{>}\,c\,\texttt{</}t\texttt{>} \mid t \in L_1 \wedge c \in L_2 \}$

If possible, give another definition of the language using one of the following approaches: an enumeration, a context-free grammar, or a predicate. Do not use the same approach that is used in the question.

If you cannot give any alternative definitions, explain why the other approaches do not work.

Note that $|w|$ denotes the length of the word w.

•

Solution 1.

(a) Note that $A^2 = A\,A = \{ a_1\,a_2 \mid a_1 \in A \wedge a_2 \in A \}$. L_1 can be defined by the following context-free grammar:

$$S \to W\,W\,W^+$$
$$W \to \mathtt{t} \mid \mathtt{u} \mid \mathtt{v}$$

(b) $L_2 = \{ B\,\mathtt{z}\,A \mid B \in \{\mathtt{b}\}^* \wedge A \in \{\mathtt{a}\}^* \wedge (|B| > 0 \supset |A| > 0) \}$ where $\supset$ is logical implication.

Also, here is a transformation of the grammar that makes it clearer how the predicate is derived.

$$S \to R\,\mathtt{a}\,U? \mid \mathtt{z}\,U?$$
$$U \to \mathtt{a}\,U?$$
$$R \to \mathtt{b}\,R \mid \mathtt{b}\,S$$

$$S \to R\,\mathtt{a}\,U? \mid \mathtt{z}\,U?$$
$$U \to \mathtt{a}\,U?$$
$$R \to \mathtt{b}\,T$$
$$T \to R \mid S$$

$$S \to \mathtt{b}\,T\,\mathtt{a}\,U? \mid \mathtt{z}\,U?$$
$$U \to \mathtt{a}\,U?$$
$$T \to \mathtt{b}\,T \mid S$$

$$S \to \mathtt{b}\,T\,\mathtt{a}\,(\mathtt{a}^+)? \mid \mathtt{z}\,(\mathtt{a}^+)?$$
$$T \to \mathtt{b}\,T \mid S$$

$$S \to \mathtt{b}\,T\,\mathtt{a}\,\mathtt{a}^* \mid \mathtt{z}\,\mathtt{a}^*$$
$$T \to \mathtt{b}\,T \mid S$$

$$S \to \mathtt{b}\,T\,\mathtt{a}^+ \mid \mathtt{z}\,\mathtt{a}^*$$
$$T \to \mathtt{b}\,T \mid S$$

(c)
- L_3 cannot be defined by an enumeration because the language is infinite.
- L_3 cannot be defined by a context-free grammar because the grammar cannot specify that the first t and second t must be equivalent. Alternatively stated, a grammar cannot enumerate all possible words for t, because L_1 is infinite.

○

Grammar Analysis and Transformation

2 (30 points)**.** Consider the following context-free grammar over the alphabet $\{\,\mathtt{?}, \mathtt{/}, \mathtt{a}, \mathtt{b}\,\}$:

$$S \to \mathtt{?}\ T \mid \mathtt{a} \mid \mathtt{b}$$
$$T \to S \mid S\ \mathtt{/}\ S$$

(a) This grammar presents a variant of the "dangling `else`" problem. Describe the problem in full, including an example of a problematic sentence for this grammar and an explanation of why the example is problematic.

(b) If you were the designer of this language, how would you solve this problem? There are multiple solutions. Describe one possible solution in full, including any new grammars as necessary. Show how your example, or some modification thereof, is no longer problematic.

•

Solution 2.

(a) The grammar is ambiguous. Consider this example:

Example: `? ? a / b`

It can be parsed in two different ways, producing two different parse trees. If we use parentheses to indicate nesting in the parse tree, then we can view the two results as follows:

- `?` (`?` (`a` `/` `b`))
- `?` ((`?` `a`) `/` `b`)

Since different parse trees probably have different meanings, we want to avoid this ambiguity.

(b) Here are two possible solutions:

a) Change the grammar to be unambiguous. Here, we only allow "closed" terms the left of a `/`.

$$S \rightarrow C \mid O$$
$$C \rightarrow \texttt{?}\ C\ \texttt{/}\ C \mid \texttt{a} \mid \texttt{b}$$
$$O \rightarrow \texttt{?}\ S \mid \texttt{?}\ C\ \texttt{/}\ O$$

The example now only has one parse tree:

`?` (`?` (`a` `/` `b`))

b) Extend the grammar with a closing symbol:

$$S \rightarrow \texttt{?}\ T\ \texttt{!} \mid \texttt{a} \mid \texttt{b}$$
$$T \rightarrow S \mid S\ \texttt{/}\ S$$

Then, to disambiguate the example, we can insert the new symbol in the appropriate place.

- `?` (`?` (`a` `/` `b`) `!`) `!`

- `?` (`(?` `a` `!`) `/` `b`) `!`

○

3 (20 points)**.** Transform the grammar below into a minimal grammar from which we can immediately derive the simplest and most efficient parser.

$S \to S\ a\ R$
$S \to T$
$T \to a\ R$
$R \to b\ c$
$P \to S\ S$

Name each transformation step and show the grammar after that transformation. You may use any of the following transformations:

Inline nonterminal	Remove duplicate productions
Introduce nonterminal	Remove left-recursion
Introduce $\cdot^*$	Remove unreachable production
Introduce $\cdot^+$	Left-factoring
Introduce $\cdot?$	

•

Solution 3.

Begin with initial grammar:

$S \to S\ a\ R$
$S \to T$
$T \to a\ R$
$R \to b\ c$
$P \to S\ S$

Remove unreachable production:

$S \to S\ a\ R$
$S \to T$
$T \to a\ R$
$R \to b\ c$

Introduce nonterminal:

$S \to S\ T$
$S \to T$
$T \to a\ R$
$R \to b\ c$

Inline nonterminal:

$S \to S\ T$
$S \to T$
$T \to a\ b\ c$

Remove left-recursion:

$S \to T\ Z \mid T$
$Z \to T\ Z \mid T$
$T \to a\ b\ c$

Introduce $\cdot^+$:

$S \to T\ Z \mid T$
$Z \to T^+$
$T \to a\ b\ c$

Inline nonterminal:

$S \to T\ T^+ \mid T$
$T \to a\ b\ c$

Introduce $\cdot^+$:

$S \to T^+$
$T \to a\ b\ c$

Inline nonterminal (optional):

$S \to (a\ b\ c)^+$

∘

Parsing Grammar Descriptions

4 (30 points)**.** There are many other notations for describing context-free (EBNF) grammars than the one we have used in the course. The grammar below describes one of those notations.

$$
\begin{array}{lll}
\mathit{Production} & \to & \mathit{Name}\ \texttt{=}\ \mathit{Expression}?\ \texttt{.} \\
\mathit{Expression} & \to & \mathit{Alternative}\ (\texttt{|}\ \mathit{Alternative})^* \\
\mathit{Alternative} & \to & \mathit{Term}^+ \\
\mathit{Term} & \to & \mathit{Name} \mid \mathit{Token}\ (\ldots\ \mathit{Token})? \mid \mathit{Group} \mid \mathit{Option} \mid \mathit{Repetition} \\
\mathit{Group} & \to & \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
\mathit{Option} & \to & \texttt{[}\ \mathit{Expression}\ \texttt{]} \\
\mathit{Repetition} & \to & \texttt{\{}\ \mathit{Expression}\ \texttt{\}}
\end{array}
$$

In this grammar for a language of grammars, production rules are described with a name and an optional expression and end with a dot. An expression is a nonempty

sequence of alternatives with a `|` separator. An alternative is a nonempty sequence of terms. A term can be either a name, a single token, a range of tokens with minimum and maximum separated by a two dots, a subexpression, an optional expression, and a possibly empty sequenced expression.

(a) The grammar for *Name* is a word with an initial uppercase character from the Latin alphabet followed by any number of alphabetic or numeric terminals. Give the necessary productions, the type (either as a datatype or a type synonym), and the parser (by using combinators you know and/or defining a new one) for *Name*.

(b) The grammar for *Token* is a sequence of alphanumeric or whitespace terminals between double quotes. Given the necessary productions, the type, and the parser for *Token*.

(c) Give the abstract syntax for the grammar of productions in the form of a family of Haskell types.

(d) Define an efficient parser using the above grammar and abstract syntax.

(e) Define the algebra type and fold for the datatype used for *Production*.

(f) Using the above fold, define the following semantic functions.

   a) The function *tokens* collects all possible tokens of a production into a list of tokens. Assume that you can enumerate all tokens in a range with a function *enumRange* which takes two tokens and results in a list of tokens.

   b) The function *productions* splits the alternatives of a production into multiple productions. It produces a list of productions where each new production has the same name as the input production and only one of the alternatives.

•

Solution 4.

(a) The grammar for *Name*:

$$
\begin{array}{ll}
\mathit{Lower} & \longrightarrow \texttt{a...z} \\
\mathit{Upper} & \longrightarrow \texttt{A...Z} \\
\mathit{Alpha} & \longrightarrow \mathit{Lower} \mid \mathit{Upper} \\
\mathit{Digit} & \longrightarrow \texttt{0...9} \\
\mathit{AlphaNum} & \longrightarrow \mathit{Alpha} \mid \mathit{Digit} \\
\mathit{Name} & \longrightarrow \mathit{Upper}\ \mathit{AlphaNum}^*
\end{array}
$$

The type:

$$\textbf{type}\ \mathit{Name} = \mathit{String}$$

The parser:

```
pAlphaNum = satisfy isAlphaNum
pUpper    = satisfy isUpper


pName :: Parser Char Name
pName = (:) <$> pUpper <*> greedy pAlphaNum
```

(b) The grammar for *Token* (where _ means space):

$$\begin{aligned} \mathit{Char} &\rightarrow \mathit{AlphaNum} \mid \_ \\ \mathit{Token} &\rightarrow \texttt{"}\ \mathit{Char}^*\ \texttt{"} \end{aligned}$$

The type:

```
type Token = String
```

The parser:

```
pSpace = satisfy isSpace
pChar  = pAlphaNum <|> pSpace


pToken :: Parser Char Token
pToken = pack p (greedy pChar) p
  where p = symbol '"'
```

(c) The abstract syntax:

```
data Production = Production Name (Maybe Expression)
data Expression = Expression [Alternative]
data Alternative = Alternative [Term]
data Term       = Name Name
                | Range Token Token
                | Token Token
                | Group Expression
                | Option Expression
                | Repetition Expression
```

(d) The parser:

```
pProduction :: Parser Char Production
pProduction = Production <$> pName <* symbol '=' <*> optional pExpression <* symbol '.'
pExpression :: Parser Char Expression
pExpression = Expression <$> listOf pAlternative (symbol '|')
pAlternative :: Parser Char Alternative
```

```
pAlternative = Alternative <$> some pTerm

pTerm :: Parser Char Term
pTerm  =  Name <$> pName
      <|> Range <$> pToken <* token "..." <*> pToken
      <|> Token <$> pToken
      <|> Group <$> parenthesised pExpression
      <|> Option <$> bracketed pExpression
      <|> Repetition <$> braced pExpression
```

(e) The algebra type:

```
type ProductionAlgebra p e a t =
  (Name → Maybe e → p   — Production
  , [a] → e             — Expression
  , [t] → a             — Alternative
  , Name → t            — Name
  , Token → Token → t   — Range
  , Token → t           — Token
  , e → t               — Group
  , e → t               — Option
  , e → t               — Repetition
  )
```

The fold:

```
foldProduction :: ProductionAlgebra p e a t → Production → p
foldProduction alg@(prod, _, _, _, _, _, _, _, _) (Production n me)
  = prod n (fmap (foldExpression alg) me)

foldExpression :: ProductionAlgebra p e a t → Expression → e
foldExpression alg@(_, exp, _, _, _, _, _, _, _) (Expression as)
  = exp (map (foldAlternative alg) as)

foldAlternative :: ProductionAlgebra p e a t → Alternative → a
foldAlternative alg@(_, _, alt, _, _, _, _, _, _) (Alternative ts)
  = alt (map (foldTerm alg) ts)

foldTerm :: ProductionAlgebra p e a t → Term → t
foldTerm alg@(_, _, _, name, range, tok, group, opt, rep) t
  = case t of
      Name n       → name n
      Range t1 t2  → range t1 t2
      Token t      → tok t
      Group e      → group (foldExpression alg e)
      Option e     → opt (foldExpression alg e)
      Repetition e → rep (foldExpression alg e)
```

a) Given *enumTokens*,

$$enumRange :: Token \to Token \to [Token]$$

we can define the function *tokens*:

```
tokens :: Production → [Token]
tokens = foldProduction
  (const (maybe [] id)   — Production
  ,concat                — Expression
  ,concat                — Alternative
  ,const []              — Name
  ,enumRange             — Range
  ,(:[])                 — Token
  ,id                    — Group
  ,id                    — Option
  ,id                    — Repetition
  )
```

b) The function *productions*:

```
productions :: Production → [Production]
productions = foldProduction
    (  f             — Production
    ,  Expression    — Expression
    ,  Alternative   — Alternative
    ,  Name          — Name
    ,  Range         — Range
    ,  Token         — Token
    ,  Group         — Group
    ,  Option        — Option
    ,  Repetition    — Repetition
    )
  where
    f :: Name → Maybe Expression → [Production]
    f _    Nothing                = []
    f name (Just (Expression as)) =
      map (λa → Production name (Just (Expression [a]))) as
```

∘