Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for the Exam

Johan Jeuring

Monday, 13 December 2010, 10:30–13:00

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 $A = \{\mathtt{x}, \mathtt{y}\}$. Give context-free grammars for the following languages over the alphabet A:

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

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

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

Solution 1.

(a) A context-free grammar for L_1 is given in the solution to exercise 2.10 in the lecture notes. Establish an inductive definition for L_1. An example of a grammar that can be derived from the inductive definition is:

$$E \to \varepsilon \mid \mathtt{x}E\mathtt{y} \mid \mathtt{y}E\mathtt{x} \mid EE$$

There are many other solutions.

(b) Using E, we define the the following grammar for L_2:

$$\begin{aligned} L &\to \mathtt{x}E \mid E\mathtt{x} \mid E\mathtt{x}E \\ L &\to \mathtt{x}L \mid L\mathtt{x} \mid LL \end{aligned}$$

Again, there are many other solutions.

○

1

Grammar analysis and transformation

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

$$\begin{aligned} S &\rightarrow S\mathtt{ab} \mid S\mathtt{a} \mid A \\ A &\rightarrow \mathtt{a}A \mid \mathtt{a}S \end{aligned}$$

2 (5 points)**.** Is the word `aababab` in $L(G)$? If yes, give a parse tree. If not, argue informally why the word cannot be in the language. •

Solution 2. The word `aababab` is not in $L(G)$, because $L(G)$ does not contain any finite words. Any production has a non-terminal in the right-hand side, so it is impossible to produce a word that only contains terminals. ○

3 (10 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:

$$\begin{aligned} S &\rightarrow S\mathtt{ab} \mid S\mathtt{a} \mid A \\ A &\rightarrow \mathtt{a}A \mid \mathtt{a}S \end{aligned}$$

We can first left-factor the grammar:

$$\begin{aligned} S &\rightarrow S\mathtt{a}R \mid A \\ R &\rightarrow \mathtt{b} \mid \varepsilon \\ A &\rightarrow \mathtt{a}T \\ T &\rightarrow A \mid S \end{aligned}$$

Now, we remove left recursion:

$$\begin{aligned} S &\rightarrow AZ? \\ Z &\rightarrow \mathtt{a}RZ? \\ R &\rightarrow \mathtt{b} \mid \varepsilon \\ A &\rightarrow \mathtt{a}T \\ T &\rightarrow A \mid S \end{aligned}$$

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

New parser combinators

4 (4 points)**.** A price in dollars can be written in two ways: \$10, or 10\$. So the currency may appear either before, or after the amount. Write a parser combinator *beforeOrAfter* that takes two parsers p and q as argument, and parses p either before or after q:

$$\mathit{beforeOrAfter} :: \mathit{Parser}\ \mathit{Char}\ a \rightarrow \mathit{Parser}\ \mathit{Char}\ b \rightarrow \mathit{Parser}\ \mathit{Char}\ (a, b)$$

•

2

Solution 4.

$$\begin{aligned} beforeOrAfter\ p\ q\ &=\ (,)\qquad <\$> p <*> q \\ &<|> flip\ (,) <\$> q <*> p \end{aligned}$$
$$testBOAl = parse\ (beforeOrAfter\ (symbol\ \texttt{'\$'})\ natural)\ \texttt{"\$10"}$$
$$testBOAr = parse\ (beforeOrAfter\ (symbol\ \texttt{'\$'})\ natural)\ \texttt{"10\$"}$$

○

5 (6 points)**.** Binding constructs are used in many languages, here are some examples:

$$\begin{aligned} &x &&:= 3 \\ &(x,y) &&\leftarrow pairs \\ &f &&= \lambda x \rightarrow x \end{aligned}$$

A binding consists of a pattern to the left of a binding token, and then a value to which the pattern is bound. I want to define a parser *pBind* that parses such bindings. *pBind* takes a parser for patterns, a parser for the binding token, and a parser for the value to which the pattern is bound, and returns the pair of values consisting of the pattern and the value. For the above examples, this would be $(x,3)$, $((x,y), pairs)$, and $(f, \lambda x \rightarrow x)$, respectively. Define the parser *pBind*. •

Solution 5.

$$pBind :: Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ c \rightarrow Parser\ s\ (a,c)$$
$$pBind\ lhs\ btoken\ rhs = (,) <\$> lhs <* btoken <*> rhs$$
$$testBind = parse\ (pBind\ identifier\ (token\ \texttt{":="})\ natural)\ \texttt{"x:=3"}$$

○

An efficient choice combinator

6 (10 points)**.** The choice parser combinator is defined by

$$(<|>) :: Parser\ s\ a \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a$$
$$(p <|> q)\ xs = p\ xs +\!\!+\ q xs$$

The ++ in the right-hand side of this definition is a source of inefficiency, and might lead to rather slow parsers. We will use a standard approach to removing this efficiency: replace append (++) by composition (.). To do this, we need to turn our parser type into a so-called accumulating parser type. The new parser type looks as follows:

$$\textbf{type}\ AccParser\ s\ a = [s] \rightarrow [(a,[s])] \rightarrow [(a,[s])]$$

Define the parser combinators *symbol* and <|> using the *AccParser* type:

$$symbol :: Char \rightarrow AccParser\ Char\ Char$$
$$(<|>) :: AccParser\ s\ a \rightarrow AccParser\ s\ a \rightarrow AccParser\ s\ a$$

•

Solution 6.

$$
\begin{aligned}
&\mathit{symbol} && :: \mathit{Char} \rightarrow \mathit{AccParser}\ \mathit{Char}\ \mathit{Char}\\
&\mathit{symbol}\ c\ [\,] && = \mathit{id}\\
&\mathit{symbol}\ c\ (y : \mathit{ys}) && = \textbf{if}\ c \equiv y\ \textbf{then}\ ((c, \mathit{ys}):)\ \textbf{else}\ \mathit{id}
\end{aligned}
$$

$$
\begin{aligned}
&(<|>) :: \mathit{AccParser}\ s\ a \rightarrow \mathit{AccParser}\ s\ a \rightarrow \mathit{AccParser}\ s\ a\\
&p <|> q = \lambda \mathit{xs} \rightarrow p\ \mathit{xs}\ .\ q\ \mathit{xs}
\end{aligned}
$$

$$
\mathit{testAcc} = (\mathit{symbol}\ \texttt{'a'} <|> \mathit{symbol}\ \texttt{'b'})\ \texttt{"ab"}\ [\,]
$$

◦

Parsing polynomials

In secondary school mathematics you have encountered polynomials. A polynomial is an expression of finite length constructed from variables and constants, using only the operations of addition, subtraction, multiplication, and non-negative integer exponents. Here are some examples of polynomials: $x + 2$, $x^2 + 3x - 4$, $(x + 2)^2$, and $x^5 - 2y^7 + z$. In this exercise, you will develop a parser for polynomials.

Since it is hard to represent superscripts in a string, we assume that before the above polynomials are parsed, they are transformed into the following expressions: `x+2`, `x^2+3x-4`, `(x+2)^2`, and `x^5-2y^7+z`. So integer exponents are turned into normal constants preceded by a `^`.

Here is a grammar for polynomials with start symbol P:

$$
\begin{aligned}
P \rightarrow\ & \mathit{Nat}\\
|\ & \mathit{Identifier}\\
|\ & P\texttt{+}P\\
|\ & P\texttt{-}P\\
|\ & PP\\
|\ & P\texttt{^}\mathit{Nat}\\
|\ & \texttt{(}P\texttt{)}
\end{aligned}
$$

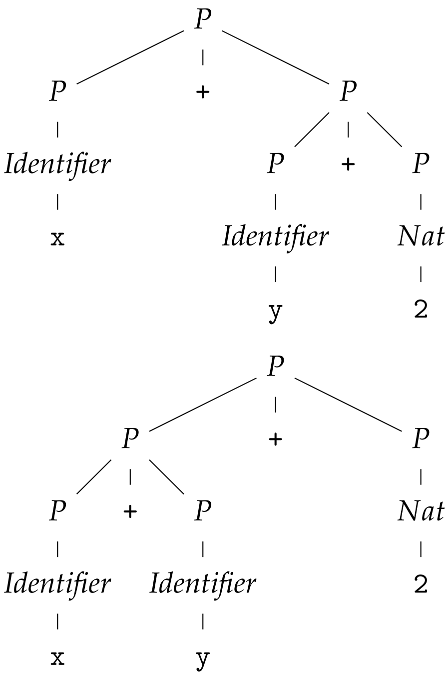
Polynomials can be composed from constant naturals (described by means of the nonterminal *Nat*), variables (identifiers, described by means of the nonterminal *Identifier*), by using addition, subtraction and multiplication (which is invisible), by exponentiation with a natural number, and parentheses for grouping.

A corresponding abstract syntax in Haskell is:

$$
\begin{aligned}
\textbf{data}\ P = &\ \mathit{Const}\ \mathit{Int}\\
|\ & \mathit{Var}\quad \mathit{String}\\
|\ & \mathit{Add}\quad P\ P\\
|\ & \mathit{Sub}\quad P\ P\\
|\ & \mathit{Mul}\quad P\ P\\
|\ & \mathit{Exp}\quad P\ \mathit{Int}\ \textbf{deriving}\ \mathit{Show}
\end{aligned}
$$

7 (5 points)**.** Is the context-free grammar for polynomials ambiguous? Motivate your answer. ●

Solution 7. The grammar is ambiguous: we can construct two different parse trees for the sentence `x+y+2`.



○

8 (10 points)**.** Resolve the operator priorities in the grammar as follows: exponentiation (`^`) binds stronger than (invisible) multiplication, which in turn binds stronger than addition `+` and subtraction `-`. Furthermore, multiplication, addition and subtraction associate to the left. Give the resulting grammar. ●

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

$$
\begin{aligned}
P &\rightarrow P_1 \\
P_1 &\rightarrow P_1\texttt{+}P_2 \mid P_1\texttt{-}P_2 \mid P_2 \\
P_2 &\rightarrow P_2P_3 \mid P_3 \\
P_3 &\rightarrow P_4\texttt{\^{}}\mathit{Nat} \mid P_4 \\
P_4 &\rightarrow \mathit{Nat} \mid \mathit{Identifier} \mid \texttt{(}P\texttt{)}
\end{aligned}
$$

○

9 (10 points)**.** Give a parser that recognizes the grammar from Task 8 and produces a value of type P:

parseP :: *Parser Char P*

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. Remember to not define left-recursive parsers. •

Solution 9. This is a rather direct transcription of the grammar as a parser using *chainl*:

```
parseP = p1
p1     = chainl p2 ((Add <$ symbol '+') <|> (Sub <$ symbol '-'))
p2     = chainl p3 (Mul <$ epsilon)
p3     = Exp <$> p4 <* symbol '^' <*> natural <|> p4
p4     = Const <$> natural
      <|> Var <$> identifier
      <|> parenthesised parseP
```

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

The test-cases work as expected:

```
ex0 = "x+2"
ex1 = "x^2+3x-4"
ex2 = "(x+2)^2"
ex3 = "x^5-2y^7+z"
tex0 = parse parseP ex0
tex1 = parse parseP ex1
tex2 = parse parseP ex2
tex3 = parse parseP ex3
```

○

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

Solution 10. We apply the systematic translation:

```
type PAlgebra r = (Int → r,       — Const
                   String → r,    — Var
                   r → r → r,     — Add
                   r → r → r,     — Sub
                   r → r → r,     — Mul
                   r → Int → r)   — Exp
foldP :: PAlgebra r → P → r
foldP (const, var, add, sub, mul, exp) = f
  where f (Const x    ) = const x
        f (Var   x    ) = var   x
        f (Add   x1 x2) = add   (f x1) (f x2)
        f (Sub   x1 x2) = sub   (f x1) (f x2)
        f (Mul   x1 x2) = mul   (f x1) (f x2)
        f (Exp   x  i ) = exp   (f x)  i
```

No surprises here. ○

11 (5 points)**.** A constant polynomial is a polynomial in which no variables appear. So $4 + 2^2$ is constant, but $x + y$ and $x^2 - x^2$ are not.

Using the algebra and fold, determine whether or not a polynomial is constant:

$$\mathit{isConstant} :: P \to \mathit{Bool}$$

•

Solution 11.

$$\begin{array}{l}
\mathit{isConstant} = \mathit{foldP}\ (\mathit{const}\ \mathit{True} \\
\qquad\qquad\qquad\quad ,\mathit{const}\ \mathit{False} \\
\qquad\qquad\qquad\quad ,(\wedge) \\
\qquad\qquad\qquad\quad ,(\wedge) \\
\qquad\qquad\qquad\quad ,(\wedge) \\
\qquad\qquad\qquad\quad ,\lambda x\ i \to x \\
\qquad\qquad\qquad\quad) \\
\mathit{testIsConstantF} = \mathit{parse}\ (\mathit{isConstant} <\$> \mathit{parseP})\ \mathit{ex0} \\
\mathit{testIsConstantT} = \mathit{parse}\ (\mathit{isConstant} <\$> \mathit{parseP})\ \texttt{"3+4\^{}2"}
\end{array}$$

○

12 (5 points)**.** Using the algebra and fold (or alternatively directly), define an evaluator for polynomials:

$$\mathit{evalP} :: P \to \mathit{Env} \to \mathit{Int}$$

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

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

•

Solution 12. We assume

$$\textbf{type}\ \mathit{Env} = \mathit{Map}\ \mathit{String}\ \mathit{Int}$$

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

$$
\begin{aligned}
&evalAlgebra :: PAlgebra\ (Env \to Int) \\
&evalAlgebra = (\lambda x \quad\ \ e \to x, \\
&\qquad\qquad\qquad\ \ \lambda x \quad\ \ e \to e\,!\,x, \\
&\qquad\qquad\qquad\ \ \lambda x_1\ x_2\ e \to x_1\ e + x_2\ e, \\
&\qquad\qquad\qquad\ \ \lambda x_1\ x_2\ e \to x_1\ e - x_2\ e, \\
&\qquad\qquad\qquad\ \ \lambda x_1\ x_2\ e \to x_1\ e * x_2\ e, \\
&\qquad\qquad\qquad\ \ \lambda x\ \ i\ \ \ e \to (x\ e)^i \\
&\qquad\qquad\qquad\ ) \\
&evalP = foldP\ evalAlgebra
\end{aligned}
$$

The environment never changes, so defining

$$
\begin{aligned}
&evalAlgebra :: Env \to PAlgebra\ Int \\
&evalAlgebra\ e = (id, (e!), (+), (-), (*), (^{\cdot}))
\end{aligned}
$$

is simpler and ok as well. ○