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:

$$L \to \mathtt{x}E \mid E\mathtt{x} \mid E\mathtt{x}E$$
$$L \to \mathtt{x}L \mid L\mathtt{x} \mid LL$$

Again, there are many other solutions.

◦

Grammar analysis and transformation

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

$$S \to S\mathtt{ab} \mid S\mathtt{a} \mid A$$
$$A \to \mathtt{a}A \mid \mathtt{a}S$$

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:

$$S \to S\mathtt{ab} \mid S\mathtt{a} \mid A$$
$$A \to \mathtt{a}A \mid \mathtt{a}S$$

We can first left-factor the grammar:

$$S \to S\mathtt{a}R \mid A$$
$$R \to \mathtt{b} \mid \varepsilon$$
$$A \to \mathtt{a}T$$
$$T \to A \mid S$$

Now, we remove left recursion:

$$S \to AZ?$$
$$Z \to \mathtt{a}RZ?$$
$$R \to \mathtt{b} \mid \varepsilon$$
$$A \to \mathtt{a}T$$
$$T \to A \mid S$$

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:

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

•

Solution 4.

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

∘

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

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

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 $(x, 3)$, $((x, y), \mathit{pairs})$, and $(f, \lambda x \rightarrow x)$, respectively. Define the parser *pBind*. •

Solution 5.

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

∘

An efficient choice combinator

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

$$\begin{aligned}
&(<|>) :: \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a \\
&(p <|> q)\ \mathit{xs} = p\ \mathit{xs} +\!\!+ \mathit{qxs}
\end{aligned}$$

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:

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

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

$$\begin{aligned}
&\mathit{symbol} :: \mathit{Char} \rightarrow \mathit{AccParser}\ \mathit{Char}\ \mathit{Char} \\
&(<|>) :: \mathit{AccParser}\ s\ a \rightarrow \mathit{AccParser}\ s\ a \rightarrow \mathit{AccParser}\ s\ a
\end{aligned}$$

•

Solution 6.

```
symbol           :: Char → AccParser Char Char
symbol c []      = id
symbol c (y : ys) = if c == y then ((c, ys):) else id
(<|>) :: AccParser s a → AccParser s a → AccParser s a
p <|> q = λxs → p xs . q xs
testAcc = (symbol 'a' <|> symbol 'b') "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*:

```
P → Nat
  | Identifier
  | P+P
  | P-P
  | PP
  | P^Nat
  | (P)
```

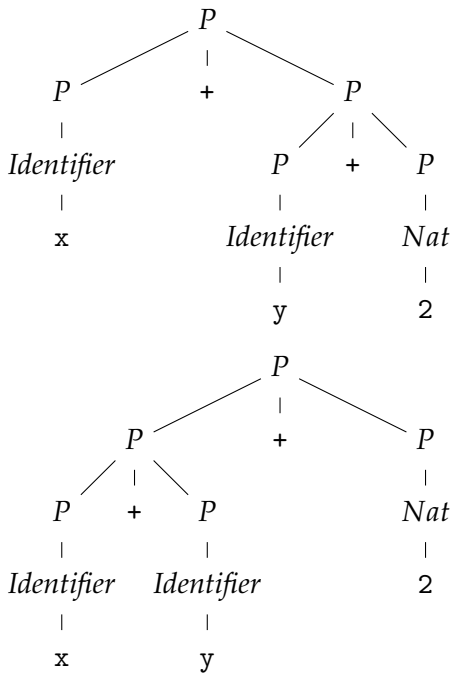
Polynomials can be composed from constant naturals (described by means of the nonterminal *Nat*), variables (identifiers, described by means of the nonterminal *Identifier*), by using addition, subtraction and multiplication (which is invisible), by exponentiation with a natural number, and parentheses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

A corresponding abstract syntax in Haskell is:

```
data P = Const Int
       | Var   String
       | Add   P P
       | Sub   P P
       | Mul   P P
       | Exp   P Int deriving Show
```

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:

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

○

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

$\mathit{parseP} :: \mathit{Parser\ Char\ P}$

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*:

$$
\begin{array}{lcl}
\mathit{parseP} & = & p_1 \\
p_1 & = & \mathit{chainl}\ p_2\ ((\mathit{Add} <\$\ \mathit{symbol}\ \texttt{'+'}) <|> (\mathit{Sub} <\$\ \mathit{symbol}\ \texttt{'-'})) \\
p_2 & = & \mathit{chainl}\ p_3\ (\mathit{Mul} <\$\ \ \mathit{epsilon}) \\
p_3 & = & \mathit{Exp} <\$> p_4 <*\ \mathit{symbol}\ \texttt{'\^{}'} <*> \mathit{natural} <|> p_4 \\
p_4 & = & \mathit{Const} <\$> \mathit{natural} \\
 & <|> & \mathit{Var} <\$> \mathit{identifier} \\
 & <|> & \mathit{parenthesised}\ \mathit{parseP}
\end{array}
$$

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

The test-cases work as expected:

$$
\begin{array}{l}
\mathit{ex0} = \texttt{"x+2"} \\
\mathit{ex1} = \texttt{"x\^{}2+3x-4"} \\
\mathit{ex2} = \texttt{"(x+2)\^{}2"} \\
\mathit{ex3} = \texttt{"x\^{}5-2y\^{}7+z"} \\
\mathit{tex0} = \mathit{parse}\ \mathit{parseP}\ \mathit{ex0} \\
\mathit{tex1} = \mathit{parse}\ \mathit{parseP}\ \mathit{ex1} \\
\mathit{tex2} = \mathit{parse}\ \mathit{parseP}\ \mathit{ex2} \\
\mathit{tex3} = \mathit{parse}\ \mathit{parseP}\ \mathit{ex3}
\end{array}
$$

○

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

Solution 10. We apply the systematic translation:

$$
\begin{array}{l}
\textbf{type}\ \mathit{PAlgebra}\ r = (\mathit{Int} \to r, \qquad -\!\!- \mathit{Const} \\
\qquad\qquad\qquad\qquad \mathit{String} \to r, \qquad -\!\!- \mathit{Var} \\
\qquad\qquad\qquad\qquad r \to r \to r, \qquad -\!\!- \mathit{Add} \\
\qquad\qquad\qquad\qquad r \to r \to r, \qquad -\!\!- \mathit{Sub} \\
\qquad\qquad\qquad\qquad r \to r \to r, \qquad -\!\!- \mathit{Mul} \\
\qquad\qquad\qquad\qquad r \to \mathit{Int} \to r) \qquad -\!\!- \mathit{Exp} \\
\mathit{foldP} :: \mathit{PAlgebra}\ r \to P \to r \\
\mathit{foldP}\ (\mathit{const}, \mathit{var}, \mathit{add}, \mathit{sub}, \mathit{mul}, \mathit{exp}) = f \\
\quad \textbf{where}\ f\ (\mathit{Const}\ x\ \ \ \ \ \ ) = \mathit{const}\ x \\
\qquad\qquad f\ (\mathit{Var}\ \ \ x\ \ \ \ \ ) = \mathit{var}\ \ \ x \\
\qquad\qquad f\ (\mathit{Add}\ \ x_1\ x_2) = \mathit{add}\ \ \ (f\ x_1)\ (f\ x_2) \\
\qquad\qquad f\ (\mathit{Sub}\ \ x_1\ x_2) = \mathit{sub}\ \ \ (f\ x_1)\ (f\ x_2) \\
\qquad\qquad f\ (\mathit{Mul}\ \ x_1\ x_2) = \mathit{mul}\ \ \ (f\ x_1)\ (f\ x_2) \\
\qquad\qquad f\ (\mathit{Exp}\ \ x\ \ i\ \ ) = \mathit{exp}\ \ \ (f\ x)\ \ i
\end{array}
$$

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:

$$isConstant :: P \to Bool$$

•

Solution 11.

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

◦

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

$$evalP :: P \to Env \to Int$$

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:

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

•

Solution 12. We assume

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

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

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

The environment never changes, so defining

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

is simpler and ok as well. ∘