Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Exam

Johan Jeuring

Monday, 13 December 2010, 10:30–13:00

Preliminaries

- The exam consists of 5 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 90.
- Try to give simple and concise answers. Write readable. Do not use pencils or pens with red ink.
- You may answer questions in Dutch, English, or Swedish.
- 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 $A = \{\mathtt{x}, \mathtt{y}\}$. 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. •

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. •

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. •

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:

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

•

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

$$x \quad := 3$$
$$(x, y) \leftarrow \mathit{pairs}$$
$$f \quad = \lambda x \to x$$

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 $(f, \lambda x \to x)$, respectively. Define the parser *pBind*. •

An efficient choice combinator

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

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

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:

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

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

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

•

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:

$$\begin{array}{lll} P & \to & Nat \\ & | & Identifier \\ & | & P\texttt{+}P \\ & | & P\texttt{-}P \\ & | & PP \\ & | & P\texttt{\^{}}Nat \\ & | & \texttt{(}P\texttt{)} \end{array}$$

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. •

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. •

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. •

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

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 → Bool
```

•

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

```
evalP :: P → Env → 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:

```
data Map k v   — abstract type, maps keys of type k to values of type v
empty   :: Map k v
```

$$
\begin{aligned}
(!) &:: \mathit{Ord}\ k \Rightarrow \mathit{Map}\ k\ v \to k \to v \\
\mathit{insert} &:: \mathit{Ord}\ k \Rightarrow k \to v \to \mathit{Map}\ k\ v \to \mathit{Map}\ k\ v \\
\mathit{delete} &:: \mathit{Ord}\ k \Rightarrow k \to \mathit{Map}\ k\ v \to \mathit{Map}\ k\ v \\
\mathit{member} &:: \mathit{Ord}\ k \Rightarrow k \to \mathit{Map}\ k\ v \to \mathit{Bool} \\
\mathit{fromList} &:: \mathit{Ord}\ k \Rightarrow [(k, v)] \to \mathit{Map}\ k\ v
\end{aligned}
$$

•