Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Exam 1

Johan Jeuring

Thursday, 19 December 2013, 08:30–10:30

Preliminaries

- The exam consists of 7 pages (including this page). Please verify that you got all the pages.
- Fill out the answers **on the exam itself**.
- Write your **name** and **student number** here:

- The maximum score is stated at the top of each question. The total amount of points you can get is 90.
- Try to give simple and concise answers. Write readable text. Do not use pencils or pens with red ink. You may use Dutch or English.
- When writing grammar and language constructs, you may use any set, sequence, or language operations covered in the lecture notes.
- When writing Haskell code, you may use Prelude functions and functions from the following modules: *Data.Char*, *Data.List*, *Data.Maybe*, and *Control.Monad*. Also, you may use all the parser combinators from the `uu-tc` package. If you are in doubt whether a certain function is allowed, please ask.

Good luck!

Questions

In the following five exercises you will write a parser for (a part of) a language for describing genealogic information in the form of family trees, and you will define several functions for obtaining particular kinds of information from a family tree, such as the oldest person in a family tree, and all names appearing in a family tree.

Here are two examples of sentences from the language for family trees:

```
BEGIN Hans Baas 12 January 1980 END

BEGIN Grietje Huizen 4 December 1953
  FATHER BEGIN Gert Huizen 11 February 1926 30 March 1987 END
  MOTHER BEGIN Anna Buurten 13 July 1929 END
END
```

A sentence in a family tree consists of:

- a single person: the keyword `BEGIN`, followed by a list of names, followed by a birth date, and an optional (a person may still live) date of death, followed by the keyword `END`.
- or a person (as above, so starting with the keyword `BEGIN` etc.) together with his or her father **and** mother, each consisting of a family tree sentence, possibly containing more fathers and mothers. The father and mother are given after the (optional) date of death, starting with `FATHER` and `MOTHER`, respectively.

1 (12 points)**.** Give the concrete syntax (a context-free grammar) of this language for family trees. •

The abstract syntax of the language for family trees is given by the following datatypes:

```
data FamilyTree = Single Person
                | Child Person FamilyTree FamilyTree deriving Show
type Person     = (Name, Birth, Maybe Death)
type Name       = [String]
type Birth      = Date
type Death      = Date
type Date       = (Day, Month, Year)
type Day        = Int
type Month      = Int
type Year       = Int
```

2 (12 points)**.** Define a parser *pFamilyTree* :: *Parser Char FamilyTree* that parses sentences from the language of family trees. •

3 (12 points)**.** Define the algebra type, and the *fold* for the datatype *FamilyTree*. You may assume that the type `Person` is a constant type such as *Int* and *String*, that is, you don't have to define a *fold* for *Person*. ●

4 (12 points)**.** Define a function $oldest :: FamilyTree \rightarrow Person$ that returns the oldest person in a family tree. Define function *oldest* as a *fold* on the datatype *FamilyTree*. You may assume the existence of a function $age :: Person \rightarrow Int$, which returns the age of a person in number of days. ●

5 (12 points)**.** The function *names*, which returns all names appearing in a family tree, can be defined as follows:

```
names :: FamilyTree → [Name]
names = foldFamilyTree (single, child)
  where single (n, bd, dd)       = [n]
        child (n, bd, dd) nf nm = n : nf ++ nm
```

The operator $+\!\!+$ used in the definition of *child* makes this definition rather inefficient. We get a more efficient function by accumulating the list of names in a parameter. The type of the function then becomes:

$$names' :: FamilyTree \to [Name] \to [Name]$$

Define the function *names'* as a *fold* on the datatype *FamilyTree*. •

6 (15 points)**.** Consider the following context-free grammar over the alphabet $\{\,\texttt{a}, \texttt{b}, \texttt{c}, \texttt{+}\,\}$ with the start symbol A:

$$\begin{aligned} A &\to B\texttt{b} \mid A\texttt{+}A \mid AB\texttt{a} \\ B &\to \texttt{c}A \mid \varepsilon \end{aligned}$$

The operator + is associative.

Describe a sequence of transformations for simplifying this grammar. The resulting grammar should be minimal and suitable for deriving a parser (using parser combinators). The grammar should not be ambiguous and should not result in inefficiency or nontermination in the parser.

You may use any of the transformations in the following list or another transformation discussed during the lecture or in the lecture notes.

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

For each transformation step in the sequence, describe the transformation and give the transformed grammar. You may use at most two transformations in one step, but you must mention both of them (e.g. "Inline S and remove unreachable production"). •

7 (15 points)**.** Consider the following three languages:

(a) $\{\,(\texttt{ab})^n \mid n \textbf{ in } N\}$

(b) $\{\,\texttt{a}^n\ \texttt{b}^n \mid n \textbf{ in } N\}$

(c) $\{\,\texttt{a}^n\ \texttt{b}^m \mid n, m \textbf{ in } N\}$

For each of these languages, answer the question: is the language regular? If so, give a DFA accepting the language, if not, prove that it is not regular using the pumping lemma. •

(a)

(b)

(c)