Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 1

Johan Jeuring

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

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

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. •

Solution 1. Concrete syntax for genealogical data:

$$
\begin{array}{lcl}
\textit{FamilyTree} & \rightarrow & \textit{Single} \mid \textit{Child} \\
\textit{Single} & \rightarrow & \texttt{"BEGIN"}\ \textit{Person}\ \texttt{"END"} \\
\textit{Person} & \rightarrow & \textit{Name}^{*}\ \textit{Date}\ \textit{Date}\,? \\
\textit{Child} & \rightarrow & \texttt{"BEGIN"}\ \textit{Person}\ \texttt{"FATHER"}\ \textit{FamilyTree}\ \texttt{"MOTHER"}\ \textit{FamilyTree}\ \texttt{"END"} \\
\textit{Date} & \rightarrow & Z\ \textit{Month}\ Z \\
\textit{Month} & \rightarrow & \texttt{"January"} \mid \texttt{...} \mid \texttt{"December"}
\end{array}
$$

where Z is a nonterminal generating integers. Here are some example sentences:

$$
\begin{array}{lcl}
\textit{fts} & = & \texttt{"BEGIN Hans Baas 12 January 1980 END"} \\
\textit{ftc} & = & \texttt{"BEGIN Grietje Huizen 4 December 1953 FATHER "} + \textit{ftf} + \texttt{" MOTHER "} + \textit{ftm} + \texttt{" END"} \\
\textit{ftf} & = & \texttt{"BEGIN Gert Huizen 11 February 1926 30 March 1987 END"} \\
\textit{ftm} & = & \texttt{"BEGIN Anna Buurten 13 July 1929 END"}
\end{array}
$$

Marking
Family tree is a list instead of a tree: -2
Missing syntax for BEGIN, END, FATHER, MOTHER: -2
A list of parents instead of two parents: -2
Also two fathers or mothers are possible: -2
Different order of parents: -2
Exactly two names: -2
Just one name: -2
Months represented as strings: -2
Months represented as two digits: -2
All components of date represented as string: -2
Missing definitions: -2 (depending on size)

○

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. •

Solution 2.

```
spaces        = greedy (satisfy isSpace)
tokensp s     = token s    <* spaces
identifiersp  = identifier <* spaces
integersp     = integer    <* spaces

today   ::   Date
today   =    (19, 12, 2013)

pFamilyTree  ::   Parser Char FamilyTree
pFamilyTree  =    pSingle
                 <|> pChild

pSingle ::   Parser Char FamilyTree
pSingle =    Single
          <$  tokensp "BEGIN"
          <*> pPerson
          <*  tokensp "END"

pPerson ::   Parser Char Person
pPerson =    (λnames birth mdeath → (names, birth, mdeath))
          <$> many pName
          <*> pBirth
          <*> option (Just <$> pDeath) Nothing

pChild   ::  Parser Char FamilyTree
pChild   =   (λp f m → Child p f m)
       <$    tokensp "BEGIN"
       <*>   pPerson
       <*    tokensp "FATHER"
       <*>   pFamilyTree
       <*    tokensp "MOTHER"
       <*>   pFamilyTree
       <*    tokensp "END"

pBirth, pDeath, pDate :: Parser Char Date
pBirth   =   pDate
pDeath   =   pDate
pDate    =   (λd m y → (d, m, y))
      <$>    integersp
      <*>    pMonth
      <*>    integersp

months = ["January", "February", "March", "April", "May", "June"
         , "July", "August", "September", "October", "November", "December"
         ]
```

```
pMonth  :: Parser Char Month
pMonth  =  foldr1 (<|>) (zipWith (λx y → const x <$> y) [1 .. 12] (map tokensp months))
pName = identifiersp
-- Parser test cases
testfts  = pFamilyTree fts
testftc  = pFamilyTree ftc
testftf  = pFamilyTree ftf
testftm  = pFamilyTree ftm
```

Marking
The parser does not follow the concrete syntax: -2
Missing or incorrect parsers for tokens: -2
Optional date parser incorrect (type error, or missing default value): -2
Parser for name incorrect: -2
Parser for month incorrect: -2
Abstract syntax value not constructed correctly: -2
Wrong usage of parser combinators: -2 or -4

○

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*. ●

Solution 3.

```
type FamilyTreeAlgebra a = (Person → a
                           ,Person → a → a → a
                           )
foldFamilyTree :: FamilyTreeAlgebra a → FamilyTree → a
foldFamilyTree (single, child) = fold
  where fold (Single p)     = single p
        fold (Child p f m)  = child p (fold f) (fold m)
```

Marking
The maximum score for the algebra is 4, and for the *fold* 8.
For the algebra part:
Recursive occurrences not replaced by variables: -4
Type variable not bound by argument: -1
Using *String* instead of *Person*: -2
For the *fold* part:
Type does not correspond to algebra: -2

Not recursive: -6
foldFamily instead of *fold* in the recursive call: -2
A type variable instead of *Person* in the algebra not resolved in the *fold*: -2

○

4 (12 points)**.** Define a function *oldest* :: *FamilyTree* → *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* → *Int*, which returns the age of a person in number of days. ●

Solution 4.

```
— imprecise implementation of age:
age :: Person → Int
age (n, (bd, bm, by), md) = let (rd, rm, ry) = case md of
                                                 Just (dd, dm, dy) → (dd, dm, dy)
                                                 Nothing           → today
                            in (ry − by) ∗ 365 + (rm − bm) ∗ 30 + (rd − bd)
oldest :: FamilyTree → Person
oldest = foldFamilyTree (single, child)
  where single p       = p
        child p fo mo = maxWith age [p, fo, mo]
maxWith :: Ord a ⇒ (b → a) → [b] → b
maxWith f [x]      = x
maxWith f (x : xs) = if f x > f xs′ then x else xs′
  where xs′ = maxWith f xs
```

Marking
Only the *single* case is correct: (+)2
Age instead of person as result: -2
Person in child is not taken into account: -2
Double recursion: -8

○

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*. •

Solution 5.

```
names' = foldFamilyTree (single, child)
  where single (n, bd, dd)        = λns → n : ns
        child (n, bd, dd) nf nm = λns → let {ns' = nf ns; ns'' = nm ns'}
                                         in n : ns''
```

Marking
Father and mother are not used: -6
The accumulator is passed on incorrectly: -6
Wrong recursive usage of functions: -4
Double recursion: -8

○

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

$$A \to B\mathtt{b} \mid A\mathtt{+}A \mid AB\mathtt{a}$$
$$B \to \mathtt{c}A \mid \varepsilon$$

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"). •

Solution 6. Initial grammar:

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

Inline B:

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

Remove unreachable production:

$$A \to \texttt{c}A\texttt{b} \mid \texttt{b} \mid A\texttt{+}A \mid A\texttt{c}A\texttt{a} \mid A\texttt{a}$$

Remove left-recursion:

$$\begin{aligned} A &\to \texttt{c}A\texttt{b} \mid \texttt{c}A\texttt{b}Z \mid \texttt{b} \mid \texttt{b}Z \\ Z &\to \texttt{+}A \mid \texttt{+}AZ \mid \texttt{c}A\texttt{a} \mid \texttt{c}A\texttt{a}Z \mid \texttt{a} \mid \texttt{a}Z \end{aligned}$$

Introduce $Z?$ (corresponds to left-factoring):

$$\begin{aligned} A &\to \texttt{c}A\texttt{b}Z? \mid \texttt{b}Z? \\ Z &\to \texttt{+}AZ? \mid \texttt{c}A\texttt{a}Z? \mid \texttt{a}Z? \end{aligned}$$

Marking
One simple transformation (such as Introduce ·?) correct: (+)2
One transformation wrong: -2
Left-recursion removal used correctly: +5

○

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

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

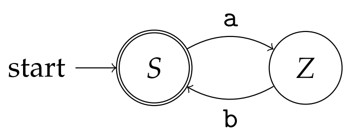
(b) $\{\,\texttt{a}^n\ \texttt{b}^n \mid n \textbf{ in } N\}$

(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. •

Solution 7.

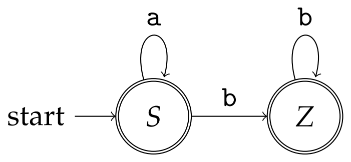
(a) This language is regular.



(b) The language is not regular. We use the regular pumping lemma to prove this.

Let $n \in \mathbb{N}$.
Take $s = xyz$, with $x = \mathtt{a}^n$, $y = \mathtt{b}^n$, and $z = \varepsilon$. The sentence $s = \mathtt{a}^n\ \mathtt{b}^n$ is an element of the language.
Let u, v, w be such that $y = uvw$ with $v \not\equiv \varepsilon$, that is, $u = \mathtt{b}^p$, $v = \mathtt{b}^q$ and $w = \mathtt{b}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then $xuv^2wz = \mathtt{a}^n\mathtt{b}^{p+2q+r}{=}\mathtt{a}^n\mathtt{b}^{n+q}$.
Since $q > 0$, this is not a sentence in the language of the grammar.
Using the negative version of the regular pumping lemma, we conclude that this language is not regular.

(c) This language is regular.



Marking
Maximum score for a) and c) is 2.5, and for b) 10 (2.5 + 7.5).
ε is not accepted: -1
Automaton accepting $(a^*\ b^*)^*$: -1.5
Regular expression instead of DFA: -1.5
Edges without labels: -0.5
Clumsy or incomplete proof: -1 to -2.5

○