# Embedded Domain-Specific Languages
# FLOLAC 2014
# Exercises, Part 1

## 1 Introduction

This short collection of exercises explores a little around deep and shallow embeddings of domain-specific languages. The domain in this case is of formal languages, grammars, and (if we get that far) parsers. In this sense, a *formal language* is nothing more than a set of finite strings, each of which is a finite (possibly empty) sequence of characters.

## 2 Grammars

Here is a simple datatype of grammars for formal languages:

```
data Grammar :: ∗ where
  Empty :: Grammar
  Unit  :: Grammar
  Single :: Char → Grammar
  Conc  :: Grammar → Grammar → Grammar
  Union :: Grammar → Grammar → Grammar
```

The intended interpretation is as follows:

- the grammar *Empty* represents the empty language $\{\}$
- the grammar *Unit* represents the language consisting of precisely one string, the empty string: $\{""\}$
- for any character $c$, the grammar *Single c* represents the language $\{[c]\}$ consisting of precisely one string, the string consisting of the single character $c$
- if $x$ and $y$ are two grammars representing languages $X$ and $Y$ respectively, then *Conc x y* is the grammar representing the language containing strings of the form $s +\!\!+ t$ where $s \in X$ and $t \in Y$
- if $x$ and $y$ are two grammars representing languages $X$ and $Y$ respectively, then *Union x y* is the grammar representing the language $X \cup Y$

1. Write a grammar representing car registration plates. In the UK, these are currently of the form "AB 34 EFG", but before that they were of the form "A 234 EFG"; I believe that in Taiwan they are currently of the forms "ABC 5678" and "AB 3456". Make sure your definition type-checks, even though you can't do anything else with it yet.

# 3 Recognition

One thing you might want to do with your grammar is to use it to decide whether a given string is within the language it represents; we call this 'recognition'. We model this as a function *recog* that takes a *Grammar* and a *String* and returns a *Bool*, indicating whether or not the string is in the language.

$$recog :: Grammar \to String \to Bool$$

2. Define *recog*.

3. Use *recog* to check your grammar of registration plates: apply it to various strings to see whether it gives you the results you expected.

As it happens, the class of grammars discussed above itself forms a little domain-specific language, and the grammar you defined for registration plates is a little program in that DSL. The datatype *Grammar* is a deep embedding of the language, and your function *recog* is a semantics for the DSL.

4. Define recognition instead as a shallow embedding. That is, represent grammars directly by their semantics (of type $String \to Bool$), and define five functions on that semantics;

$$\begin{aligned}
&\textbf{type } Grammar = String \to Bool \\
&empty :: Grammar \\
&unit \;\; :: Grammar \\
&single :: Char \to Grammar \\
&conc \;\; :: Grammar \to Grammar \to Grammar \\
&union :: Grammar \to Grammar \to Grammar
\end{aligned}$$

The recognition function above is a little awkward, especially as regards the *Conc* constructor: in order to recognise whether a string $s$ is in the formal language represented by $Conc\ x\ y$, it's not enough to ask whether $s$ is in the languages represented by $x$ and by $y$—you have to ask about different strings ($s1$ and $s2$ such that $s1 +\!\!+ s2 = s$) instead. More convenient is to define a function

$match :: Grammar \to String \to Maybe\ (String, String)$

that returns a little more information. Here, the datatype *Maybe* represents optional values:

**data** $Maybe\ a = Just\ a \mid Nothing$

The idea is that $match\ g\ s$ takes a grammar $g$ and a string $s$, and determines whether any prefix of $s$ matches $g$. If $s$ is not in the formal language represented by $g$ it returns *Nothing*, instead of *False*. But if $s$ is in the formal language. it returns $Just\ (s1, s2)$, where $s1$ and $s2$ are two strings such that $s1 +\!\!+ s2 = s$, and $s1$ matches $g$. That's more useful for *Conc* cases, because it tells you where to start matching the second language.

5. Define *match*, twice: once as a semantics for the deep embedding of *Grammar*, and once as a shallow embedding.

# 4 Generation

Here's another semantics for grammars: interpret them as 'generators', that generate the formal language they represent. For example, applying a generator to your language of registration plates should generate a (very long!) list of possible registration plates.

6. Define the function

   $generate :: Grammar \to [String]$

   as a semantics for the deep embedding.

7. As an alternative, take the semantics $[String]$ as a shallow embedding, and redefine the five grammar constructors on this type.

If you answered the question above naively, it will work but will not be very interesting. For example, on UK registration plates, it might generate "AA 00 AAA", then "AA 00 AAB", and so on, and you'll have to wait for $26^3 = 17576$ elements to pass before you see anything as exciting as "AA 01 AAA"; and you'll have to wait an awful lot longer before you see the first of the form "A 000 AAA". For generating strings in the union of two formal languages, you might want to 'interleave' the generators of the two languages; for generating strings in the concatenation of the two languages, you might want to 'diagonalise'. I will explain these two techniques in class.

8. Redefine *generate* to use interleaving and diagonalisation.

# 5 Parsing

Recognising, matching, and generating are all very well, but often one wants to do more than merely ask whether a string is in a formal language—if it is, one wants to use it for some purpose, building some data structure based on the string. In programming languages, that process is known as 'parsing', although it is a bit of a misnomer.

It's two small steps from grammars to parsers. The first step is to define a parametrised type *Parser a* instead of an unparametrised type *Grammar*; a parser of type *Parser a* is like a *Grammar*, representing a set of strings, but it will also allow you to extract a value of type *a* from any one of those strings. For example, a parser of type *Parser Float* will allow you to recognise strings representing floating point numbers, but also to extract the floating point number in question. The second step is to add one more constructor, which we call *Using*, to modify extracted values. So the datatype is as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Parser} :: * \to * \ \textbf{where}\\ &\quad \mathit{Fail} \quad :: \mathit{Parser}\ a\\ &\quad \mathit{Succeed} :: a \to \mathit{Parser}\ a\\ &\quad \mathit{Char} \quad :: \mathit{Char} \to \mathit{Parser}\ \mathit{Char}\\ &\quad \mathit{Seq} \quad :: \mathit{Parser}\ a \to \mathit{Parser}\ b \to \mathit{Parser}\ (a, b)\\ &\quad \mathit{Choice} :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a\\ &\quad \mathit{Using} \quad :: \mathit{Parser}\ a \to (a \to b) \to \mathit{Parser}\ b\end{aligned}$$

The constructors correspond to those of *Grammar* in the obvious way, except for *Using* which is new.

9. Define a semantics

   $$\mathit{parse} :: \mathit{Parser}\ a \to \mathit{String} \to \mathit{Maybe}\ (a, \mathit{String})$$

   for the deeply embedded DSL of parsers above. This is like *match* for grammars, except that instead of returning a successful match $\mathit{Just}\ (s1, s2)$, it returns the extracted value in place of $s1$. (Hint: the definitions you need are in my lecture notes.)

10. Use the semantic domain

    $$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to \mathit{Maybe}\ (a, \mathit{String})$$

    as a shallow embedding of the parser DSL instead of an interpretation of a deep embedding; define the six parser constructors as functions on this domain.

Jeremy Gibbons
University of Oxford
July 2014