Exercise 6: Generating Expressions

So far, we have seen how grammars can be used to define the syntax of a language, and (more powerfully) to define the *recursive structure* of terms in the language that suggest how to perform operations on them. In this exercise, you'll learn about a new form of computing with grammars: using their recursive structure to *generate* expressions in a language.

(Fun fact: the Haskell testing library `quickcheck` does something similar, though much more sophisticated, when generating inputs for property-based tests.)

Starter code

- Ex6.hs

Note that this week's exercise is entirely in Haskell; we take advantage of its natural laziness (and memoization—see lecture!) to express our code purely in terms of familiar list operations, without worrying about memory blow up as we generate large numbers of expressions.

Task 1: A small arithmetic expression grammar

Consider the following grammar:

```
<expr> = <number>
       | (+ <expr> <expr>)
       | (* <expr> <expr>)

<number> = 1 | 2 | 3 | 4
```

That is, it consists of arbitrarily-nested addition and multiplication expressions, where each numeric literal is a digit between 1 and 4. On this exercise, we will call the `<number>` case for `<expr>` a *base case* because it is not recursive, while the `(+ <expr> <expr>)` and `(* <expr> <expr>)` cases *recursive* because it expands an `<expr>` in terms of other `<expr>`s. Note that the parentheses in the recursive cases are *required* (like in Racket); this helps disambiguate expressions, since each `+` or `*` operations must be enclosed in its own set of parentheses. Since we're operating in Haskell, we don't have a "datum" datatype like we would in Racket. Instead, we represent these expressions as strings, for example: `"3"` and `"(+ (* 4 2) (+ 2 1))"`.

We define the **rank** of an expression in this language is the maximum depth of the nesting of its subexpressions. We can define this recursively based on the grammar:

- A `<number>` has rank 0.
- An expression of the form `(+ <expr> <expr>)` has rank one greater than the *maximum* of the rank of its two subexpressions. The same holds for `(* <expr> <expr>)`.

Rank gives us a way of comparing the size or complexity of our generated expressions. This proves useful when generating expressions, as this notion gives us a way of ordering expressions as we generate them.

Now, turn to the starter code and complete the functions listed under Task 1.

Task 2: Generalize!

Now let's generalize our work in the previous section to work on different types of grammars. Formally, we define a **binary-recursive grammar** to be a pair of two lists, where

- The first list contains all of the literals in the grammar (these are the base cases).
- The second list contains *binary functions* that take in two arbitrary expressions in the grammar and return another one (these are the recursive rules of the grammar).

For example, the arithmetic expression grammar given in Task 1 is a binary-recursive grammar. Here's another example; note that even though the base cases are listed as two separate rules, this is only for human readability.

```
<expr> = <animal>
       | <emotion>
       | <expr> and <expr>!
       | <expr> or <expr>?

<animal> = cats | dogs | birds
<emotion> = love | terror | hunger
```

The *rank* of an expression for a binary-recursive grammar is defined in the same way as part 1; for example, `"cats and love!"` has rank 1, and `"dogs or terror? or birds? and love and hunger!!"` has rank 3. (The punctuation mark at the end is required to disambiguate the nesting structure here. Makes you miss parentheses, right?)

Now, read through and complete Task 2 in the starter code. We've already provided you with the crucial type `BinaryGrammar`; study it carefully to make sure that you understand it. You may find it helpful to review the previous labs where we first talked about types in Haskell, and how you defined functions on them using pattern-matching.

For course-related questions, please contact **david at cs.toronto.edu**.