Exercise 2: Recursive Data Types and Abstract Syntax Trees

In this exercise, you'll check your understanding of two programming language concepts we covered in lecture by implementing some static code analyzers for these ideas. This exercise is an opportunity to practice writing functions on these using structural pattern-matching.

Deadline: September 24, 2019 before 10:00pm

Starter code

- ex2.rkt

Task 1: Calculator II

For this task, we will modify the `calculate` function from last week to allow simple if statements involving comparisons. We will modify our arithmetic expressions to have the following grammar:

```
Binary Arithmetic Expression Grammar

<expr> = NUM
       | (<op> <expr> <expr>)
       | (if (<comp> <expr> <expr>) <expr> <expr>)
<comp> = = | > | <
<op>   = + | - | * | /
```

where `NUM` represents any integer literal in base 10, and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings. Likewise, the comparison operations `=`, `>` and `<` have their standard mathematical meanings as well.

The semantics of an `if` expression is like in Racket. For example, the expression `(if (= a b) c d)`, where `a, b, c, d` are expressions, is evaluated as follows:

- evaluate the expressions `a` and `b`
- compare the resulting values of `a` and `b` using Racket `equal?`, `<` or `>`
- if the comparison succeeds, then return the value of `c`
- otherwise, return the value of `d`

Note that **only one branch of the if statement should be evaluated**. In particular, evaluating the expression `(if (= 1 1) 42 <gibberish>)` using your `calculate` function should return `42`, even if `<gibberish>` is invalid.

What to do

Your task is to complete the function `calculate` that returns the value of an expression. Start from your code from last week's exercise.

Task 2: Finding tail calls

For this task, we are **not** working with the arithmetic grammar. Instead, we will be working with a subset of Racket.

We discussed last week how recursive functions can blow the call stack if we aren't careful, and that some programming languages protect against this by implementing an optimization known as *tail call elimination*. The obvious first step to implementing such an optimization is to actually check for function calls in tail position—that's what you'll do in this task.

More formally, you'll write a function that's given an abstract syntax tree representing an expression, and returns a list of function call subexpressions that are in tail position with respect to the outer expression. We can formally define the *tail call position(s)* of a small subset of a functional language (like Racket) using structural recursion:

- An atomic value (an expression with no subexpressions, e.g. literal or identifier) has *no* tail call positions.
- An expression that is a function call is in tail call position. None of its subexpressions are in tail call position.
- An expression that is an `(if <cond> <then> <else>)` has the following tail call positions: the tail call positions of the `<then>` expression, *and* those of the `<else>` expression (in that order).
- The tail call positions of `(and <expr1> <expr2> ...)` is equal to the tail call positions of the *last* argument expression. If there are no argument expressions—`(and)`— this has *no* tail call positions. The same rule applies to `or` expressions.

Note that while tail positions exist for any type of expression, on this exercise we're only looking for *function calls* that are in tail position. So for example, in the expression `(if (f x) 10 20)`, both `10` and `20` are in tail position, but would not be returned by your function, since they aren't function calls.

Your function must be able to handle all expressions consisting of the (arbitrarily-nested combinations of) elements in the items above, but that's all. You don't need to handle other types of expressions, such as `define`, `lambda`, or `cond`.

You will implement this function in Racket. You'll find it helpful to first complete Lab 2, which introduces the notion of *datums* (our representation of ASTs in Racket).

Submission instructions

Submit the file `ex2.rkt` to MarkUs.