Exercise 4: Eager Evaluation and Strictness Analysis

In lecture, we contrasted strict and non-strict denotational function call semantics, and saw how most programming languages perform *left-to-right eager evaluation* of a function call's arguments. On the other hand, we saw that Haskell uses a different strategy to evaluate function calls, which we colloquially call "lazy evaluation" because it defers the evaluation of arguments until they are required to evaluate the function body.

In this exercise, you'll learn more about this idea by implementing an interpreter that performs eager evaluation. We will also look at an idea called strictness analysis.

You should not use the built-in `eval` interpreter in any part of this exercise, or any other exercise.

Deadline: October 8, 2019 before 10:00pm

Starter code

- ex4.rkt

Task 1: Calculator (Part IV) Eager Evaluation

To explore eager evaluation, let's begin by expanding the binary arithmetic grammar from the previous weeks to introduce expressions for function definition and function calls. Our new grammar is as follows:

```
Lambda Calculus with Binary Arithmetic

<expr> = NUM
       | ID
       | (<op> <expr> <expr>)
       | (if (<comp> <expr> <expr>) <expr> <expr>)
       | (let* ((ID <expr>) ...) <expr>)
       | (lambda (ID ...) <expr>)       ; function expression (function definition)
       | (<expr> <expr> ...)            ; function call
<comp> = = | > | <
<op>   = + | - | * | /
```

Where `NUM` represents any integer literal in base 10, `ID` is a variable name (a symbol in Racket), and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings. Likewise, the comparison operations `=`, `>` and `<` have their standard mathematical meanings as well.

Evaluating Function Expressions (Defining Functions)

A function expression evaluates to a *closure*, or a data structure containing two things:

- The function expression.
- The environment at the time the function expression is evaluated. This is to implement lexical scoping.

We will store function expressions as a *list* containing three elements:

- The first element will be the token `'closure`, and will be used to identify closures during function call
- The second element will be the function body expression `<expr>`

- The third element will be the environment

For example, `(lambda (x) (+ x 1))` should evaluate to `'(closure (lambda (x) (+ x 1)) ENVIRONMENT)`.

Where `ENVIRONMENT` is the environment (hash table) at the time that the function expression is evaluated.

Evaluating Function Calls (Calling Functions)

A function call should be evaluated eagerly, left-to-right, like this:

- First, evaluate the function expression. (Normally, we would check that the value of the function expression is a closure. For this exercise, you may assume that the value of the function expression will always be a closure.)
- Evaluate each of the argument expressions, left-to-right
- Evaluate the function body, extending the appropriate environment with the additional argument bindings

Note: If you use the wrong environment in the last step, your interpreter will be dynamically scoped rather than lexically scoped.

What to do

Your task is to modify the interpreter `eval-calc`, starting from the code from last week's exercise, to support function definitions and function calls.

Task 2: Strictness Analysis

Lazy evaluation strategy incurs a (possibly significant) space cost, and so a fundamental optimization the Haskell compiler performs is *strictness analysis*, which is a static analysis used to determine whether a function parameter is *always* required to evaluate the function body. If this is the case, whenever the function is called, Haskell can eagerly evaluate the argument corresponding to this parameter, and pass in just its value to the function, reducing the space cost.

In this task, you'll learn more about this idea by performing a basic strictness analysis on a (different) small language.

Language grammar

Here is a grammar describing the language we'll use on this exercise.

```
<prog> = <function-def> ..1   # One or more function definitions
<function-def> = ( define ( ID ID ... ) <expr> )  # The first ID is the function name,
                                                   # and any others are function parameters.
<expr> = NUMBER
       | ID
       | ( if <expr> <expr> <expr> )  # Same syntax as Racket
       | ( ID <expr> ... )            # Function call, where the first expression MUST
                                      # be an identifier.
```

As usual, `NUMBER` refers to a numeric literal, while `ID` refers to any valid Racket identifier. This language has one built-in identifier `+`, which refers to the arithmetic addition operation. Since there are no booleans in the language, interpret `if` as checking whether its condition is non-zero for "true" and zero for "false".

Note that this language is restricted in that the first position of a function call expression *must* be an identifier: one of `+`, a globally-defined function name, or a function parameter.

Strictness

Now we'll define the key notion of *strictness* for this language. Let E be an expression (i.e., `<expr>`) in this language, and x be an identifier not equal to `+`. The intuition we want to capture is whether evaluating E *requires (in all cases)* evaluating x. Formally, we define whether x *is* ***strict*** *in* E using structural recursion:

- If E is a number, x is *not* strict in E.
- If E is an identifier, x is strict in E if and only if x equals E.
- If E is an `if` expression, x is strict in E if and only if at least one of the following hold:
  1. x is strict in the condition of E.
  2. x is strict in *both* the "then expression" and "else expression" of E. (Both are required because the definition of strictness requires x to be necessary in all possible cases.)
- If E is a call to `+`, x is strict in E if and only if it is strict in at least one argument of the function call.
- If E is a call of a function identifier f, then x is strict in E if and only if at least one of the following hold:
  1. x equals f (i.e., E is calling x itself).
  2. x is strict in an argument subexpression of E, and that argument corresponds to a parameter that is strict in the body of f.

The second condition in the last rule is the trickiest, but before tackling that, let's look at some examples of the other cases.

```
5    ; x is not strict in this
x    ; x is strict in this
y    ; x is not strict

(if x 1 2)  ; x is strict
(if y x x)  ; x is strict
(if y x z)  ; x is NOT strict

(+ x 1 2 3)  ; x is strict
(x 1 2 3)    ; x is strict
```

Now let's return to function declarations. Suppose we have a function declaration `(define (f x1 x2 ... xn) body)`, and consider parameter `x1`. We say that `x1` is a **strict parameter** of `f` if and only if `x1` is strict in `body` (using the definition of strictness from above). Intuitively, this means that whenever `f` is called, the value of the argument passed to `x1` is *required* to evaluate the body of `f`.

So then another way of putting the second condition of the last strictness rule above is:

2. x is strict in an argument subexpression of E, and that argument corresponds to a strict parameter of f.

Here's an example:

```
(define (f a b) a)  ; a is a strict parameter of f, and b isn't.

(f x 1)             ; x is strict in this function call.
(f 1 x)             ; x is NOT strict in this function call.
```

Update (October 4)

Because strictness analysis needs to efficiently connect arguments to their corresponding parameters, we record the strict parameters of a function using a list of integers representing their index in the function's parameter list. Here are some examples:

```
(define (f1 a b) a)           ; (list 0), since only a is strict

(define (f2 a b c) (+ b c))   ; (list 1 2), since b and c are strict
```

```
(define (f3 a b c d) 10)      ; (list) or empty, since no parameters are strict
```

Finally, the result of strictness analysis is a strictness map: a map of function identifiers to their corresponding strict parameter list. For example, the strictness map corresponding to the three function definitions from above is:

```
(hash 'f1 (list 0)
      'f2 (list 1 2)
      'f3 (list))
```

Language restrictions

As usual, you may assume that inputs to your strictness analyzer are syntactically valid, and also obey the following additional semantic restrictions:

1. Function definitions cannot be recursive.
2. The identifier `+` is never shadowed or re-bound.
3. A function body can only contain the following identifiers:
   - `+`
   - parameters of that function
   - function identifiers that have been defined *above* the current function definition (so as usual, the top-down order of function definitions matters)
4. There are no type errors (so only functions are called, functions get the right number of type of arguments, only numbers are passed to `+`, etc.).

Since the strictness analysis is new to you, we don't want you to have to worry about checking for errors as well. :)

What to do

You will write a function that performs a (static) *strictness analysis* on an input program in the form described above, i.e., a list of function definitions. The output of your function is a *mapping* of function names to a list of indexes of the strict parameters of that function (0-indexed, so the first parameter has index 0), in increasing order. Every function defined in the program should appear in the map, even if it has no strict parameters (in this case, the corresponding value is an empty list).

See the starter code for some sample tests. In addition to the main function we're actually testing (`analyze-strictness/analyzeStrictness`), we've provided one key helper that should prove useful. However, you're free to modify and/or remove that helper if you prefer a different approach.

Just for fun

Your interpreter in task 1 probably will not support recursion. For example, this expression computes the factorial of 5 using a recursive function.

```
(let*
  ((fac (lambda (n)
    (if (= n 0)
        1
        (* n (fac (- n 1)))))))
  (fac 5))
```

Your interpreter should not be able to evaluate this expression. In any case, `let*` in Racket does not support recursion. However, `letrec` in Racket *does* support recursion:

```
(letrec
  ((fac (lambda (n)
    (if (= n 0)
```

```
      1
      (* n (fac (- n 1)))))))
(fac 5))
```

You do **not** need to implement `letrec` for this exercise. However, if you figure out how to do so before the midterm, show me and I will give you a cookie.