# Programming Languages – Interpreter Lab 2

So far we have implemented numbers, math, subexpressions, variables, definitions, and an ifzero statement. What we will add today are user-defined functions. We do this first by adding support for creating new functions (closures), then we will add support for calling those functions.

Mini-Racket will make a few simplifications to the way normal Racket works:

- In Mini-Racket, all functions will have exactly one argument. This removes the need for lambdas to have parentheses around the variable list. So Mini-Racket lambdas will look like, for example, `(lambda x (+ x 1))` instead of `(lambda (x) (+ x 1))`.

- In normal Racket, there are two versions of define: one that defines a variable, such as `(define x 3)`, and one that defines a function, such as `(define (add1 x) (+ x 1))`. The second version is just a shortcut for writing `(define add1 (lambda (x) (+ x 1))`.

- In Mini-Racket, we will only have one version of define that works with both variables and functions. For example, in Mini-Racket, we will write `(define add1 (lambda x (+ x 1)))`.

- Normal Racket implements function calls by just having the name of the function be the first item in a list, as in `(add1 5)`. In Mini-Racket, to make function calls easier to recognize, we will have an explicit "call" expression, as in `(call add1 5)`.

1. In Mini-Racket (and normal Racket), lambda expressions always evaluate to closures. In normal Racket, we never discussed how closures are implemented "under the hood" because we never had to. But in Mini-Racket, we must have an internal data structure to store them. We will represent a closure in Mini-Racket by a list of four items: the symbol `'closure`, the lambda's variable, the body of the lambda expression, and the environment in which the closure appears --- we need this last piece needed to implement lexical scope.

    a. Write the function `(lambda? expr)` which detects if an `expr` is a lambda expression, which looks like `(lambda var body)` in Mini-Racket.

    b. Write the function `(eval-lambda expr env)` which creates a closure from `expr` (which should be a Mini-Racket lambda expression). `eval-lambda` should simply create a list of the four components described above.

2. To call a function, we first need to write `mini-apply`, which takes a closure and an argument value and applies the body of that closure to the argument.

    a. Write the function `mini-apply`, which follows the rules of environment diagrams to call a function. (Make a new frame mapping the argument to its value, point the frame's pointer to the closure's frame, evaluate the closure's body in the new environment.)

    b. Write `(call? expr)` which detects if an expr is a function all expression. Note that in Mini-Racket, user-defined functions are called as `(call <func> <argument>)`

    c. Write `(eval-call? expr env)`. Remember, expr will look like `(call <func> <arg>)`. So evaluate the `func` to get a closure, then evaluate the `arg` to get a value, then apply the closure to the value and return the result.

    d. Celebrate! You now have a Turing-complete language that is (technically) just as powerful as Racket, Java, C, C++, or Python.

    *(over)*

3. Try these things:

   Hint: Since the Mini-Racket interpreter will kick you out with an error message if you make a syntax mistake --- which will cause you to lose all your previous work inside of Mini-Racket, I recommend writing these functions in a separate text editor and pasting them in as you get them working. You can paste in multiple lines at once.

   a. Define some functions like `add1`, `sub1`. Experiment with calling them.

   b. Turns out Mini-Racket gives you recursion for free! Write a function of an argument n that sums up the numbers from 1 to n and returns the result. (See examples in the code for Fibonacci and factorial).

   c. Write a function called `even?` that returns 1 if its argument is even and 0 if it's odd. You may assume the argument is 0 or positive.

   d. Write a function to explore the Collatz conjecture. First, write a function called "`collatz`" that does the following:

      `collatz`(n) = n/2 if n is even; and 3n+1 if n is odd.

      The Collatz conjecture states that starting with any integer n, repeating calling the `collatz` function on n will eventually reach the number 1.

      Write a function called `count-collatz` that takes an argument n and counts the number of times `collatz` has to be called on n to get to 1.