COMP 325 — Interpreter 3

Functional Programming

Fall 2017

For your third interpreter you'll be implementing a purely functional language for basic numerical computing.

Expressions

The language should support the following expressions.

- Binary Arithmetic: +,-,*,/,modulus
- Numeric Comparison: ==, <. >, <=, >=
- Boolean Operators: and,or,not (binary only. standard unary not)
- Conditionals: if and cond
- First Class, n-ary Functions: lambda expressions
- `let` and multi-binding `let*`

We've covered most of these in class so there isn't much new here.

Programs

Users write programs in the language using the familiar *main* function paradigm. The program is a series of top-level function and constant definitions. Within that set of definitions there should exactly one named *main* which is the program executed by the run-time system (i.e. Pyret). Arguments to the program can be passed from the "CLI". To simplify matters we'll only allow numerical or boolean arguments, no passing functions from the command-line.

Semantics

The language utilizes some now familiar semantics as well as something new: *currying*. Functions should use eager argument evaluation and have statically scoped identifiers. All operators have run-time type checking and boolean operators should short circuit. The new semantic element allows us to leverage statically scoped closures in such a way that function semantics are reduced to unary functions.

Currying Functions

Currying reduces n-ary functions to a series of nested unary functions. For example,

```
(def (foo a b c)
   (+ a (+ b c)))
```

Would curry to

```
(def (foo a)
  (lambda (b)
    (lambda(c)
      (+ a (+ b c)))))
```

Which is equivalent to

```
(def foo
  (lambda (a)
    (lambda (b)
       (lambda (c)
          (+ a (+ b c))))))
```

This does a couple of things. First, it allows for the desugaring of n-ary functions. Second, it enables a rough equivalent partial evaluation of n-ary functions. With partial evaluation, users can pass m fewer than n arguments to the function and effectively get the $(n - m)$-ary function back in which the first m arguments of the original function are bound. For example passing 5 and 2 to *foo* results in the equivalent of this function.

```
(def addseven
   (lambda (c)
      (+ 5 (+ 2 c))))
```

When currying is at play, the result isn't, strictly speaking, a partially evaluated function because the return value is a unary function. Doing a partial evaluation of *foo* the ternary function with a single argument would result in a binary function. When we first curry *foo* we get a unary function, which returns a unary function. For example, *(foo 5)* gives the equivalent to these two results:

```
## partial eval
(lambda (b c)
  (+ 5 (+ b c)))

## curried foo
(lambda (b)
   (lambda (c)
     (+ 5 (+ b c))))
```

If currying turns all functions into unary functions then not only do we need to rewrite function definitions, but we need to rewrite function applications. When *foo* is curried as show above then *(foo 1 2 3)* needs to be rewritten as *(((foo 1) 2) 3)* so that the arguments are bound to the same identifiers they would have been had the function not been automatically curried.

Logistics

The completed interpreter is due on **Monday 10/23**. The sections of the interpreter are graded differently this go around. Adjust your work habits accordingly.

Area	Points
Desugar	25
Interp	20
Parse	15
Data Definitions	10
Style and Comments	5
	75 total

You are expected to have all required expressions represented in all parts of your design. If it's not there and at least stubbed, you will lose points. At this point we should be able to lay out the skeleton of top level cases. Sufficiency of testing is covered for the section your testing. The core language should be as minimal as possible without going overboard. The quality of your sugar versus non-sugar choices will be evaluated as part of your Data Definitions grade. The style and comment part of your grade accounts for good coding practices like proper indentation, avoiding printed line wrapping, good identifier and function names, documentation, and commenting of logic.