CS 111 Advanced Tutorial 3: A Simple Interpreter

In this tutorial, we'll look at some of the basics of how interpreters work.

First, some stuff we didn't talk about in class

We told you that you can make a list containing the number 1, 2, and 3 by saying: `(list 1 2 3)`. That makes the list by running the `list` procedure and passing it the numbers 1, 2, and 3. The procedure then creates the new list and returns it.

But there's a simpler way of doing it. You can just say:

```
'(1 2 3)
```

The single quote mark at the beginning tells Racket that everything between the parentheses is a list constant: it should just take all that data and package it up as a list. Because it knows that everything up to the matching close parent is data for the list, you can also use this to put lists in lists:

```
'((1 2 3) (4 5 6))
```

The above means a list with two elements, the first of which is the list `(1 2 3)` and the second of which is the list `(4 5 6)`. And note that you don't put additional quote marks inside; the one quote at the beginning is all.

Quoting is easier but less general than calling list. If you say `(list a b c)`, you get a list with the values of the variables `a`, `b`, and `c`, respectively. Since you're calling a procedure, all the procedure's arguments get run too, and so you can use this to make a list whose contents gets computed rather than being fixed. But if you say `'(a b c)`, it doesn't look up the values of the variables `a`, `b`, and `c`, and put those in this list. Instead, it makes a list of the *strings* `a`, `b`, and `c`. Except they're a special kind of string called a *symbol*. The only different between strings and symbols for our purposes is how you type them. You don't have to put double-quotes around symbols when they're inside quoted lists. If you want to type one outside of a quoted list, just put a single quote on it. So to type the symbol `a` outside a list, just say `'a`.

Why am I telling you this? Because this is one of the things that makes people who like Racket like it a lot. Because notice that I can take any piece of Racket code whatsoever:

```
(define a 10)
```

And put a quote in front of it:

```
'(define a 10)
```

And now it's a data structure! That means it's really easy to write metaprograms like interpreters, compilers, and debuggers in Racket because it's easy to represent Racket source code as data.

Let's write an interpreter!

Okay, so now that we know how to represent source code as a nice data structure, let's write some code to manipulate it. In particular, we'll write a very simple interpreter for a subset of Racket.

Dictionaries

First, we need to have some way of keeping track of the values of the variables in our interpreted program. So we need a **data structure** that tracks the values of the different variables. We've been calling this data structure "the dictionary" in class, so let's stick with that.

We've written some code for you to represent a dictionary as a list of lists. There are better representations, but we're just using lists and structs right now in class, so this will have to do. There's one sublist for each variable and the sublist has the name of the variable first, and its value second.

If you look in the `rkt` file, you'll find definitions for `lookup`, which takes a variable name (a symbol) and a dictionary and returns the value of the named variable in the dictionary:

> (`lookup` *variable-name dictionary*)
> Returns the value of *variable-name* within *dictionary*

We've also made a dictionary you can use and put it in the variable `dictionary`. Next week, you'll write code that makes new dictionaries, but for the moment, the one we gave you is enough to do this assignment.

How an interpreter works

An interpreter takes a data structure that represents your code, and it walks through it, doing what it says to do. Since we're interpreting a subset of Racket, that means our code is always an *expression* and running the code means getting the value of the expression. Moreover, we know that expressions come in different flavors with different rules for running them:

- Primitive expression
  - Constants are their own values
  - Variable references return the value of the variable in the dictionary
- Compound expressions
  - Procedure calls
    Run all the subexpressions, take the value of the first one (a procedure) and call it with the values of the others
  - Special forms
    We'll get to these later

So our interpreter is going to be called with a data object representing an expression, it will look at the object to see which *kind* of expression it's running, and then use the right rule for that kind of expression.

Preview

Let's step through how this is going to run. We're going to call our interpreter `evaluate` because that's kind of tradition. When we say `(evaluate '(+ 1 2))`, the following sequence of calls is going to happen:

- `evaluate` realizes it's argument is a complex expression and so calls `(evaluate-complex-expression '(+ 1 2))`.
- `evaluate-complex-expression` realizes it's argument is a procedure call and so calls `(evaluate-procedure-call '(+ 1 2))`.

- `evaluate-procedure-call` recursively calls evaluate on each of the subexpressions of the call: +, 1, and 2.
    - `(evaluate '+)` calls `(evaluate-primitive-expression '+)`, which calls `(lookup '+ dictionary)`.
        - The value given for the variable named `'+` in the dictionary is just Racket's normal + procedure, so this returns Racket's + procedure.
    - `(evaluate 1)` calls `(evaluate-primitive-expression 1)`, which just returns 1.
    - `(evaluate 2)` calls `(evaluate-primitive-expression 2)`, which just returns 2.
- `evaluate-procedure-call` now calls the value of the first expression (Racket's normal boring + procedure) with 1 and 2 as arguments, and gets 3 as a result.

Interpreting a primitive expression

In our data structure for representing expressions, a constant expression like 10 is just represented as the number 10. Similarly, a string is represented by just the string itself. But references to variables are represented as a symbol holding the name of the variable. You can test whether something is a symbol or not using the `symbol?` procedure.

Write a procedure `evaluate-primitive-expression`, that takes an object representing a primitive expression (that is, either a symbol or something else like a number) and returns its value using the rules above for primitive expressions. *Hint: the code for this is far simpler than the English text trying to explain what to write.*

By the way, the reason we call this thing `evaluate-primitive-expression` rather than `interpret-primitive-expression` is just that it's traditional to call interpreters like this `evaluate` or `eval`. Evaluation is essentially a synonym for interpreting.

You can test it by running things like

- `(evaluate-primitive-expression 1)`
  To test evaluating constants. This should return 1!
- `(evaluate-primitive-expression 'b)`
  To test evaluating variable references. This should return the value of b in the `dictionary`, which is 2.

Interpreting an arbitrary expression

Any expression that isn't a primitive expression is a complex expression and has parentheses in the source code. And when you quote that code, you get a list. So any complex expression is represented as a list.

Write a procedure `evaluate`, that takes an object representing an expression (complex or primitive) and uses either `evaluate-primitive-expression` or `evaluate-complex-expression` to run it, depending on whether it's a list. Note that we haven't written `evaluate-complex-expression` yet, so you can't test it yet.

Interpreting procedure calls

For the moment, we're just going to have our interpreter handle procedure calls, but not other kinds of complex expressions. So write `evaluate-complex-expression` to just call `evaluate-procedure-call`.

```
(define (evaluate-complex-expression exp)
   (evaluate-procedure-call exp))
```

We'll add support for things like `if` and `lambda` next week.

Okay, so now we want to write `evaluate-procedure-call`. This is going to take a list representing the procedure call expression and run that expression. So for example, if the source code was `(+ 1 2)`, `evaluate-procedure-call` will get called with the list `'(+ 1 2)`, i.e. a three element list whose elements are the symbol +, and the numbers 1 and 2, respectively:

```
;; evaluate-procedure-call: list -> any
;; Take the list representing a procedure call, run it, and return the
;; result.
```

Great! How do we run a procedure call? First we run all the subexpressions. Since our call is a list, the subexpressions are the elements of the list. So we need to run all the elements of the list and get their results back. So ask yourself:

- What is the name of a procedure you've written that you can call with an arbitrary expression and that will run it and return the result?
- What procedure did we teach you about in class that lets you run a procedure on all the elements of a list and give you back a list of the results?
- What special form did we tell you about to take the result of an expression and stash it in a local variable?

Start by writing a definition for `evaluate-procedure-call` that takes the list passed to it and runs all the elements of the list and stashes a list of the results in a local variable.

Now you have a list of the results of all the subexpressions. The rest of the rule for running a procedure call is to take the result of the first subexpression, which had better be a procedure, and call it with the results of the rest as inputs.

You can use `first` to get the value of the first subexpression, and `rest` to get the values of the others. And you can use `apply`, which was discussed in class but we wouldn't blame you for having forgotten, to call a procedure and pass it arguments taken from a list. So put those together to finish off your definition of `evaluate-procedure-call`, and with it your interpreter.

Conclusion

That's all you have to do to write the minimum possible interpreter. It's basically just a glorified calculator. Once of its glaring deficiencies is that it doesn't have `lambda`, so you don't have any way of defining a procedure, and so you can't ever call anything except one of Racket's procedures that you stashed in the dictionary in advance. But we'll add `lambda` and `if` next week.