1 Implementing Environments

Until now, we have not shown any implementation of environments. There are a number of ways we might consider implementing them, the most obvious of which is probably a list implementation:

\[
\text{(define-struct binding (var-name value))}
\]

\[
;; \text{extend-env: env } \times \text{ var-name } \times \text{ value } \rightarrow \text{ env}
\]
\[
\text{(define (extend-env env name value)}
\]
\[
\text{(cons (make-binding name value) env))}
\]

\[
;; \text{empty-env: env}
\]
\[
\text{(define empty-env empty)}
\]

\[
;; \text{lookup-env: env } \times \text{ var-name } \rightarrow \text{ value}
\]
\[
\text{(define (lookup-env env name)}
\]
\[
\text{(cond}
\]
\[
\text{[(empty? env)}
\]
\[
\text{(error \text{'undefined-identifier})]}
\]
\[
\text{[(symbol=? (binding-var-name (first env)) name)
}\]
\[
\text{(binding-value (first env))]}
\]
\[
\text{[else (lookup-env (rest env) name)])}}
\]

Let’s take a slightly different view of environments. Environments are really just a mapping from variable names to values. Or, in Scheme terms,

\[
;; \text{env: var-name } \rightarrow \text{ value}
\]

That is, environments are just functions! Let’s implement them as such:

\[
;; \text{extend-env: env } \times \text{ var-name } \times \text{ value } \rightarrow \text{ env}
\]
\[
\text{(define (extend-env env name value)}
\]
\[
\text{(lambda (want)}
\]

\[
\text{\text{.}}
\]
(cond
  [(symbol=? name want) value]
  [else (lookup-env env want)])

This may seem weird, but it is actually quite straightforward. Our contract for extend-env did not change, so (define (extend-env env name value) stays the same too. The function extend-env must return an environment, which we have decided to explicitly represent as a function. This explains the lambda. Environments are functions that consume one argument, a var-name, and therefore the function we return from extend-env consumes a var-name named want. You will notice that the rest of extend-env looks very similar to lookup-env. By embedding the functionality of lookup-env in extend-env, the new lookup-end is just function application:

;; lookup-env: env × var-name → value
(define (lookup-env env name)
  (env name))

The implementation for empty-env is simply a function that consumes a var-name and returns an error:

;; empty-env: env
(define empty-env
  (lambda (want)
    (error "error undefined-identifier")))

2 A New Representation for Rip Procedures

Let’s consider our representation of numbers in Rip. We made the decision that Rip numbers be represented as Scheme numbers. Scheme numbers handle overflow automatically. If we want to have Rip handle overflow in a special way, we might need to provide our own implementation of arithmetic that captures our desired overflow modes, and use this to implement Rip arithmetic.

This class, however, is not about representing numbers in a computer. The important point is that by writing an interpreter ourselves, we get the power to make these kinds of choices. A choice more relevant to CS 173 is the issue of how to represent procedures.

Instead of strictly using structs, we might consider using strings, vectors, or even lambda. Let’s try the latter.

The procE clause of the interpreter will need to be modified. When we implemented environments with procedures, we embedded the original lookup code in extend-environment. Here we do a similar thing: We want Rip procedure application to be implemented with Scheme function application, so we embed the original do-app code into the lambda.
Our `appE` clause remains the same, but our `do-app` function gets much simpler; it becomes function application. The code is:

```scheme
(define (do-app lamV argV)
  (lamV argV))
```

A `procE` expression now evaluates to a Scheme function that takes a Rip value as its argument. Procedure application in Rip is now just function application in Scheme.

There is a very important question to consider: Where did the closure go? Don’t we need to remember the environment extant when the procedure is created? Certainly, and we do this here by relying on Scheme’s static scoping. Scheme’s `lambda` actually creates a closure! The `env` in the code for the `procE` clause will be the environment that exists when the function is created, not when it is invoked. If Scheme were dynamically-scoped, this approach would not work.

### 3 Meta Interpreters

It wasn’t that difficult to represent Rip procedures with Scheme functions. There’s a reason for this: Rip and Scheme have the same semantics.

**Definition 1 (meta interpreter)** A meta interpreter is an interpreter that uses the language features (i.e., something more than string rewriting) of the interpreting language.

**Definition 2 (syntactic interpreter)** A syntactic interpreter is an interpreter that does not rely on the language features of the interpreting language.

Our first Rip substitution interpreter was nearly a syntactic interpreter. The only language feature it borrowed from Scheme was numbers. Our current interpreter is a meta interpreter—we use Scheme closures to implement Rip closures, Scheme function application for Rip procedure application, Scheme numbers for Rip numbers, and Scheme addition for Rip addition.

With a good match between the interpreted language and the interpreting language, writing a meta interpreter can be very easy. With a bad match, though, it can be very hard. With a syntactic interpreter, implementing each semantic feature will be pretty hard, though you don’t necessarily have to worry about getting a good language match.

Meta interpreters often allow you to easily gain insight into the subtleties of a language. This is not often true of syntactic interpreters. Meta interpreters, we will see later in the
course, can be used to automatically generate compilers for new languages we write.

In fact, ignoring the switch from parens to curls, our current interpreter can be classified as something more specific than a meta interpreter:

**Definition 3 (meta-circular interpreter)** A meta-circular interpreter is a meta interpreter in which the interpreting and interpreted language are the same.

Other than for educational purposes, are there any uses of a meta-circular interpreter? Yes. We saw that a meta interpreter is good for testing your understanding of a language. A meta-circular interpreter is good for testing a language that you write for bugs. By writing a meta-circular interpreter in a language that you designed, you are likely to find problems or inconsistencies that you hadn’t considered before. In fact, one definition of a truly powerful language is that makes it easy to write a meta-circular interpreter in it.