In programming language lingo, the substitution cache is usually called the environment. Henceforth, we will use that terminology instead.

1 Re-Implementing Environments

We have seen one way of implementing environments, which is a datatype representing a stack:

```
(define-datatype SubCache SubCache?
  [mtSub]
  [aSub (name symbol?) (value FA-value?) (sc SubCache?)])
```

Let’s take a slightly different view of environments. Environments are really just a mapping from identifiers to values. But they are a particular kind of mapping: at every instant, when we look up the value of an identifier in an environment, we want to get at most one value for it. That is, environments are just (partial) functions! In a language like Scheme, we can implement those directly as functions without having to go through a data structure representation. First, let’s define a predicate for our new representation of environments:

```
(define (env? x)
  (procedure? x))
```

This predicate is not exact, but it’ll suffice for our purposes. Using this representation, we have a different way of implementing `aSub`:

```
;; aSub: symbol FA-value env → env
(define (aSub bound-name bound-value env)
  (lambda (want-name)
    (cond
      [(symbol=? want-name bound-name) bound-value]
      [else (lookup want-name env)])))
```

The function `aSub` must return an environment, and since we’ve chosen to represent environments by Scheme functions (`lambdas`), the function `aSub` must return a function. This explains the `lambda`.

An environment is a function that consumes one argument, a `want-name`, the name we’re trying to look up. It checks whether the name that name is the one bound by the current procedure. If it is, it returns the bound value, otherwise it continues the lookup process. How does that work?

```
;; lookup : symbol env → FA-value
(define (lookup name env)
  (env name))
```

An environment is just a procedure expecting an identifier’s name, so to look up a name, we simply apply it to the name we’re looking for.

The implementation of the initial value, `mtSub`, is simply a function that consumes an identifier name and halts in error:

```
;; mtSub : () → env
(define (mtSub)
  (error))
```
(lambda (name)
  (error 'lookup "no binding for identifier")))

These changes do affect the definition of FA-value:

(define-datatype FA-value FA-value?
  [numV (n number?)
   [closureV (param symbol?)
     (body FA?)
     (cache env?)])

However, the core interpreter itself remains unchanged:

;; interp : FAE env → FA-value
;; evaluates FAE expressions by reducing them to their corresponding values
(define (interp expr env)
  (cases FAE expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l env) (interp r env))]
    [sub (l r) (numV– (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (param body)
      (closureV param body env)]
    [app (fun-expr arg-expr)
      (local ([define fun-val (interp fun-expr env)]
        [define arg-val (interp arg-expr env)])
        (cases FA-value fun-val
          [closureV (cl-param cl-body cl-env)
            (interp cl-body
              (aSub cl-param arg-val cl-env))]
          [else (error 'interp "can only apply functions")))]))])

2 A New Representation for FAE Functions

Let’s consider our representation of numbers. We made the decision that FAE numbers be represented as Scheme numbers. Scheme numbers handle overflow automatically by growing as large as necessary. If we want to have FAE numbers overflow in a different way (by performing modular arithmetic, say, as Java’s numbers behave), we might need to provide our own implementation of arithmetic that captures our desired overflow modes, and use this to implement FAE arithmetic.

Because numbers are not so interesting compared to the other things we’ll be studying, we won’t be conducting such an exercise in this course. The important point relevant to this course is that by writing an interpreter ourselves, we get the power to make these kinds of choices. A related choice, which is relevant to this course, is the representation of functions.

What other representations are available for FAE functions (fun)? Currently, our interpreter uses a datatype. We might also use strings or vectors; vectors would gain little over a datatype, and it’s not quite clear how to use a string. One Scheme type that ought to be useful, however, is Scheme’s own procedure mechanism, lambda. Let’s consider how that might work.

First, we need to change our representation of function values. We will continue to use a datatype, but only to serve as a wrapper of the actual function representation (just like the numV clause only wraps the actual number). That is,

(define-datatype FA-value FA-value?
  [numV (n number?)
   [closureV (p procedure?)])

We will need to modify the fun clause of the interpreter. When we implemented environments with procedures, we embedded a variant of the original lookup code in the redefined aSub. Here we do a similar thing: We want FAE
function application to be implemented with Scheme procedure application, so we embed the original `app` code inside the Scheme procedure representing a FAE function.

```scheme
[fun (param body)
  (closureV (lambda (arg-val)
    (interp body
      (aSub param arg-val env)))))
]
```

That is, we construct a `closureV` that wraps a real Scheme closure. That closure takes a single value, which is the value of the actual parameter. It then interprets the body in an extended environment that binds the parameter to the argument’s value.

These changes immediately foster two important questions:

1. Which environment do we extend when evaluating the body? The one that was active at the time of procedure definition, `env`, thereby preserving static scope. How do we know that’s the one we’ll get? Because Scheme obeys static scoping. That is, Scheme’s `lambda` does the hard work of making sure we get the “right” environment.

2. Doesn’t the body get interpreted when we define the function? No, it doesn’t. It only gets evaluated when something—hopefully the application clause of the interpreter—extracts the Scheme procedure from the `closureV` value and applies it to the value of the actual parameter.

Correspondingly, application becomes

```scheme
[app (fun-exp arg-exp)
  (local ((define fun-val (interp fun-exp env))
    (define arg-val (interp arg-exp env)))
    (cases FA-value fun-val
      [closureV (p)
        (p arg-val)]
      [else (error 'interp "can only apply functions")])])
]
```

That is, having reduced the function and argument positions to values, the interpreter extracts the Scheme procedure that represents the function, and applies it to the argument value.

Therefore, the entire interpreter becomes:

```
;; interp : FAE env → FA-value
;; evaluates FAE expressions by reducing them to their corresponding values
(define (interp expr env)
  (cases FAE expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l env) (interp r env))]
    [sub (l r) (numV- (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (param body)
      (closureV (lambda (arg-val)
        (interp body
          (aSub param arg-val env)))]
    [app (fun-exp arg-exp)
      (local ((define fun-val (interp fun-exp env))
        (define arg-val (interp arg-exp env)))
        (cases FA-value fun-val
          [closureV (p)
            (p arg-val)]
          [else (error 'interp "can only apply functions")])])])
```

In short, a `fun` expression now evaluates to a Scheme procedure that takes a FAE value as its argument. Function application in FAE is now just procedure application in Scheme.
3 Types of Interpreters

It wasn’t that difficult to represent FAE functions with Scheme procedures. There’s a reason for this: FAE and Scheme have the same semantics. This suggests a taxonomy in the world of interpreters.

Definition 1 (syntactic interpreter) A syntactic interpreter is one that uses the interpreting language to represent only terms of the interpreted language, implementing all the corresponding behavior explicitly.

Definition 2 (meta interpreter) A meta interpreter is an interpreter that uses language features of the interpreting language to directly implement behavior of the interpreted language.

While our first FAE substitution-based interpreter was very nearly a syntactic interpreter, we haven’t written any purely syntactic interpreters so far: even that interpreter directly used Scheme’s implementation of numbers. The interpreter we have studied today is a true meta interpreter—we use Scheme closures to implement FAE closures, Scheme procedure application for FAE function application, Scheme numbers for FAE numbers, and Scheme arithmetic for FAE arithmetic.

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 somewhat hard, but in return you don’t necessarily have to worry about getting a good language match. In particular, if there is a particularly strong mismatch between interpreting and interpreted language, it may take less effort to write a syntactic interpreter than a meta interpreter. (Consider, for instance, if we had wanted FAE to employ dynamic rather than static scope! How much work would we have to do to implement this using the Scheme procedure representation of FAE functions?)

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.

Meta-circular interpreters have a clear detriment: they inhibit understanding, or at least do nothing to enhance it. This is because, to gain an understanding of the language being interpreted—one of the reasons we write interpreters in the first place—you need to already thoroughly understand that language so you can understand the interpreter! Therefore, meta-circular interpreters are cute but not very effective educational tools (except, perhaps, when teaching one particular language). In this course, therefore, we will write interpreters that balance between syntactic and meta elements, using only those meta features that we have already understood well (such as Scheme closures). This is the only meta-circular interpreter we will write.

That said, meta-circular interpreters do serve in one very important role: they’re good at finding weaknesses in language definitions! For instance, if you define a new scripting language, no doubt you will get some of its features fairly right, such as those to parse data files or communicate over a network. But will you get the domain-independent parts—procedures, scope, etc.—right also? And how can you be sure? One good way is to try and write a meta, then meta-circular interpreter using the language. You will probably soon discover all sorts of deficiencies in the core language. The failure to apply this simple but effective experiment is partially responsible for the amazingly messy state of many scripting languages (Tcl, Perl, JavaScript, Python, etc.) for so long; only now are they getting powerful enough to actually support effective meta-circular interpreters.

In short, by writing a meta-circular interpreter, you are likely to find problems, inconsistencies and, in particular, weaknesses that you hadn’t considered before. In fact, some people would argue that a truly powerful language is one that makes it easy to write its meta-circular interpreter.2

---

1 Though a poor choice of meta language can make this much harder than necessary!
2 Indeed, if the Scheme meta-circular interpreter we studied today seems rather concise, try to find a meta-circular interpreter for Prolog. It looks like a cheat.