Let’s examine the process of interpreting the following small program. Just because it’s syntactically more convenient, we will use with to write out our example. Consider the evaluation of:

{with {x 3}
  {with {y 4}
    {with {z 5}
      (+ x (+ y z))}}}

= {with {y 4}
  {with {z 5}
    (+ 3 (+ y z))}}

= {with {z 5}
  (+ 3 (+ 4 z))}

= (+ 3 (+ 4 5))

at which point it reduces to an arithmetic problem. To reduce it, though, the interpreter had to apply substitution three times: once for each with. This is slow! How slow? Well, if the program has length $n$ (measured in atomic expressions, say), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least $O(n^2)$. That seems rather wasteful; surely we can do better.

How do we avoid redundancy in computations? The classic computer science technique is to use caching. Caches are not only low-level units of hardware; anything that stores results of prior or potential computations can be called a cache. In this case, we want a cache of substitutions.

Concretely, here’s the idea. Initially, we have no substitutions to perform, so the cache is empty. Every time we encounter a substitution (in the form of a with or application), we augment the cache with one more entry, recording the identifier’s name and the value (if eager) or expression (if lazy) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.

Doing this breaks a key invariant we had established earlier: namely that any identifier the interpreter encounters is of necessity free, for had it been bound, it would have been replaced by substitution. But we’re not using substitution any longer! Thus, we are likely to indeed encounter identifiers during interpretation. How do we replace these identifiers with their values? Why, of course, we consult the cache!

Let’s provide a data definition for a substitution cache:

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

where SubCache stands for a “substitution cache”. A SubCache has two forms: its either empty (mtSub) or non-empty (represented by an aSub structure). The latter contains a reference to the rest of the cache in its third field.

The interpreter obviously needs to consume both an expression and a substitution cache. Therefore, its contract becomes

```
;; interp : FWAE SubCache → FWAE
```

It will need a helper function that looks up the value of identifiers in the cache. Its code is:

---

1 Puzzle: Can it be worse than that? Prove it either way.
2 "Empty sub"—get it?
;; lookup : symbol SubCache → FWAE

(define (lookup name sc)
  (cases SubCache sc
    [mtSub () (error "no binding for identifier")]
    [aSub (bound-name bound-value rest-sc)
      (if (symbol=? bound-name name)
        bound-value
        (lookup name rest-sc))])))

With that fanfare, we can now present the interpreter:

(define (interp expr sc)
  (cases FWAE expr
    [num (n) expr]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num- (interp l sc) (interp r sc))]
    [with (bound-id named-expr bound-body)
      (interp bound-body
        (aSub bound-id
          (interp named-expr sc)
          sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body) expr]
    [app (fun-expr arg-expr)
      (local (define fun-val (interp fun-expr sc))
        (interp (fun-body fun-val)
          (aSub (fun-arg fun-val)
            (interp arg-expr sc)
            sc)))]))

To make sure this is correct, we recommend that you first study its behavior on programs that have no identifiers—i.e., verify that the arithmetic rules do the right thing—and only then proceed to the rules that involve identifiers.