Substitution Caches

[[[PLAI Chapter 5 (called "deferred substitutions" there)]]]

Evaluating using substitutions is very inefficient -- at each scope, we copy a piece of the program AST. This includes all function calls, which implies an impractical cost (function calls should be *cheap*!).

To get over this, we want to use a cache of substitutions.

Basic idea: we begin evaluating with no cached substitutions, and collect them as we encounter bindings.

Implies another change for our evaluator: we don't really substitute identifiers until we get there, so when we reach an identifier it is no longer an error -- we must consult the substitution cache at that point.

Initial Implementation of Cache Functionality

First, we need a type for a substitution cache. For this we will use a list of lists of two elements each -- a name and its value ***FLANG***:

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))
```

We need to have an empty substitution cache, a way to extend it, and a way to look things up:

```
(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend id expr sc)
  (cons (list id expr) sc))

(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (cond [(null? sc) (error 'lookup "no binding for ~s" name)]
        [(eq? name (first (first sc))) (second (first sc))]
        [else (lookup name (rest sc))]))
```

Actually, the reason to use such list of lists is that Racket has a built-in function called `assq' that will do this kind of search (`assq' is a search in an association list using `eq?' for the key comparison).

Example:

> (assq 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)

> (assq 7 (list (list 1 2) (list 3 4) (list 5 6)))

#f

This is a version of `**lookup**' that uses `**assq**':

```
(: lookup : Symbol SubstCache -> FLANG)
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))
```

Formal Rules for Cached Substitutions

The formal evaluation rules are now different. Evaluation carries along a **"substitution cache"** that begins its life as ***empty***: so `**eval**' needs an **extra argument**.

Lookup rules:
We begin by writing the rules that deal with the cache, and use the above function names for simplicity -- the behavior of the three definitions can be summed up in a single rule for `lookup':

```
lookup(x,empty-subst)      = error!
lookup(x,extend(x,E,sc))   = E
lookup(x,extend(y,E,sc))   = lookup(x,sc)  if `x' is not `y'
```

Evaluation rules:
Now, we can write the new rules for `**eval**'

```
eval(N,sc)                 = N
eval({+ E1 E2},sc)         = eval(E1,sc) + eval(E2,sc)
eval({- E1 E2},sc)         = eval(E1,sc) - eval(E2,sc)
eval({* E1 E2},sc)         = eval(E1,sc) * eval(E2,sc)
eval({/ E1 E2},sc)         = eval(E1,sc) / eval(E2,sc)
eval(x,sc)                 = lookup(x,sc)
eval({with {x E1} E2},sc)  = eval(E2,extend(x,eval(E1,sc),sc))
eval({fun {x} E},sc)       = {fun {x} E}
eval({call E1 E2},sc)
         = eval(Ef,extend(x,eval(E2,sc),sc))
                            if eval(E1,sc) = {fun {x} Ef}
         = error!           otherwise
```

1. Note that there is no mention of `**subst**' -- the whole point is that we don't really do substitution, but use the cache instead. The `**lookup**' rules and the places where `**extend**' is used -- replace `**subst**', and therefore specify our scoping rules.

2. Also note that the rule for `**call**' is still very similar to the rule for `**with**', but it looks like we have lost something -- the interesting bit with substituting into `**fun**' expressions (i.e., whether we should or should not substitute).

Evaluating with Substitution Caches

Implementing the new `eval' is easy now -- it is extended in the same way that the formal `eval' rule is extended:

```
(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [else (error 'eval
                      "`call' expects a function, got: ~s"
                      fval)]))]))
```

Again, note that we don't need `subst' anymore, but the rest of the code (the data type definition, parsing, and `arith-op') is exactly the same.

Finally, we need to make sure that `eval' is initially called with an ***empty cache***. This is easy to change in our main `run' entry point:

```
(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s"
                   result)])))
```

The full code (including the same tests, but not including formal rules for now) follows. Note that one test does not pass…

```
#lang pl

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name)  (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)
(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend name val sc)
  (cons (list name val) sc))

(: lookup : Symbol SubstCache -> FLANG)
```

```
(define (lookup name sc)
  (let ([cell (assq name sc)])
    (if cell
      (second cell)
      (error 'lookup "no binding for ~s" name))))

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
  (: Num->number : FLANG -> Number)
  (define (Num->number e)
    (cases e
      [(Num n) n]
      [else (error 'arith-op "expects a number, got: ~s" e)]))
  (Num (op (Num->number expr1) (Num->number expr2))))

(: eval : FLANG SubstCache -> FLANG)
;; evaluates FLANG expressions by reducing them to expressions
(define (eval expr sc)
  (cases expr
    [(Num n) expr]
    [(Add l r) (arith-op + (eval l sc) (eval r sc))]
    [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
    [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
    [(Div l r) (arith-op / (eval l sc) (eval r sc))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (extend bound-id (eval named-expr sc) sc))]
    [(Id name) (lookup name sc)]
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (cases fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr sc) sc))]
         [else (error 'eval "`call' expects a function, got: ~s"
                              fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) empty-subst)])
    (cases result
      [(Num n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
```

```
                  {with {x 3}
                    {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
               {with {foo {fun {x} {+ x 1}}}
                 {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
               {with {f {fun {y} {+ x y}}}
                 {with {x 5}
                   {call f 4}}}}")
      => "???")
(test (run "{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4}")
      => 7)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

Dynamic and Lexical Scopes

This seems like it should work, and it even worked on a few examples, except for one which was hard to follow. Seems like **we have a bug**...

Now we get at a tricky issue that managed to be a problem for ***lots*** of language implementors, ***including the first version of Lisp***. Let's try to run the following expression -- try to figure out what it will evaluate to:

```
(run "{with {x 3}
        {with {f {fun {y} {+ x y}}}
          {with {x 5}
            {call f 4}}}}")
```

We expect it to return **7** (at least I do!), but instead it returns **9**... The question is -- ***should* it return 9**?

What we have arrived at is called "**dynamic scope**". Scope is determined by the dynamic run-time environment (which is represented by our substitution cache). This is ***almost always*** undesirable, as I hope to convince you.

Before we start, we define two options for a programming language:

- **Static Scope** (also called Lexical Scope): In a language with static scope, each identifier gets its value from the scope in which it was **defined** (not the one in which it is used).
- **Dynamic Scope:** In a language with dynamic scope, each identifier gets its value from the scope of its use (not its definition).

Racket uses lexical scope, our new evaluator uses dynamic, the old substitution-based evaluator was static etc.

Side-remark: Lisp began its life as a dynamically-scoped language. The artifacts of this were (sort-of) dismissed as an implementation bug. When Scheme was introduced, it was the first Lisp dialect that used strictly lexical scoping, and Racket is obviously doing the same. (Some Lisp implementations used dynamic scope for interpreted code and lexical scope for compiled code!) In fact, Emacs Lisp is the only *live* dialects of Lisp that is still dynamically scoped by default. Too see this, compare a version of the above code in Racket:

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

and the Emacs Lisp version (which looks almost the same):

```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
```

```
      (let ((x 5))
        (funcall f 4))))
```

which also happens when we use another function on the way:

```
  (defun blah (func val)
    (funcall func val))

  (let ((x 3))
    (let ((f (lambda (y) (+ x y))))
      (let ((x 5))
        (blah f 4))))
```

and note that renaming identifiers can lead to different code -- change that `val' to `x':

```
  (defun blah (func x)
    (funcall func x))

  (let ((x 3))
    (let ((f (lambda (y) (+ x y))))
      (let ((x 5))
        (blah f 4))))
```

and you get 8 because the argument name changed the `x' that the internal function sees!

Consider also this Emacs Lisp function:

```
  (defun return-x ()
    x)
```

which has no meaning by itself (`x' is unbound),

```
  (return-x)
```

but can be given a dynamic meaning using a `let':

```
  (let ((x 5)) (return-x))
```

or a function application:

```
  (defun foo (x)
    (return-x))

  (foo 5)
```

There is also a dynamically-scoped language in the course languages:

```
#lang pl dynamic

(define x 123)

(define (getx) x)

(define (bar1 x) (getx))
(define (bar2 y) (getx))

(test (getx) => 123)
(test (let ([x 456]) (getx)) => 456)
(test (getx) => 123)
(test (bar1 999) => 999)
(test (bar2 999) => 123)

(define (foo x) (define (helper) (+ x 1)) helper)
(test ((foo 0)) => 124)

;; and *much* worse:
(define (add x y) (+ x y))
(test (let ([+ *]) (add 6 7)) => 42)
```

Note how bad the last example gets: you basically cannot call any function and know in advance what it will do.

There are some cases where dynamic scope can be useful in that it allows you to "remotely" customize any piece of code. A good example of where this is taken to an extreme is Emacs: originally, it was based on an ancient Lisp dialect that was still dynamically scoped, but it retained this feature even when practically all Lisp dialects moved on to having lexical scope by default. The reason for this is that the danger of dynamic scope is also a way to make a very open system where **almost anything can be customized by changed it "remotely".** Here's a concrete example for a similar kind of dynamic scope usage that makes a very hackable and open system:

```
#lang pl dynamic

(define tax% 6.5)
(define (with-tax n)
  (+ n (* n (/ tax% 100))))

(with-tax 10) ; how much do we pay?
(let ([tax% 17.0]) (with-tax 10)) ; how much would we pay in Israel?

;; make that into a function
(define il-tax% 17.0)
(define (us-over-il-saving n)
  (- (let ([tax% il-tax%]) (with-tax n))
     (with-tax n)))
```

```
(us-over-il-saving 10)
;; can even control that: how much would we save if the tax in israel
;; went down one percent?
(let ([il-tax% (- il-tax% 1)]) (us-over-il-saving 10))
```

Obviously, this power to customize everything is also the main source of problems with getting no guarantees for code. A common way to get the best of both worlds is to have "controllable" dynamic scope. For example, Common Lisp also has lexical scope everywhere by default, but some variables can be declared as "special", which means that they are dynamically scoped. The main problem with that is that you can't tell when a variable is special by just looking at the code that uses it, so a more popular approach is the one that is used in Racket: all bindings are always lexically scoped, but there are "parameters" which are a kind of dynamically scoped value containers -- but they are bound to plain (lexically scoped) identifiers. Here's the same code as above, translated to Racket with parameters:

```
#lang racket

(define tax% (make-parameter 6.5))    ; create the dynamic container
(define (with-tax n)
  (+ n (* n (/ (tax%) 100))))         ; note how its value is
accessed

(with-tax 10) ; how much do we pay?
(parameterize ([tax% 17.0]) (with-tax 10)) ; `parameterize', not
`let'

;; make that into a function
(define il-tax% (make-parameter 17.0))
(define (us-over-il-saving n)
  (- (parameterize ([tax% (il-tax%)]) (with-tax n))
     (with-tax n)))

(us-over-il-saving 10)
(parameterize ([il-tax% (- (il-tax%) 1)]) (us-over-il-saving 10))
```

The main point here is that the points where a dynamically scoped value is used are under the programmer's control -- you cannot "customize" what `-' is doing, for example. This gives us back the guarantees that we like to have (= that code works), but of course these points are pre-determined, unlike an environment where everything can be customized including things that are unexpectedly useful.

(As a side-note, after many decades of debating this, Emacs has finally added lexical scope in its core language, but this is still determined by a flag -- a global `**lexical-binding**' variable.)

Dynamic versus Lexical Scope (offline discussion):

Back to the discussion of whether we should use dynamic or lexical scope:

- The most important fact is that we want to view programs as executed by the normal substituting evaluator. Our original motivation was to optimize evaluation only -- not to *change* the semantics! It follows that we want the result of this optimization to behave in the same way. All we need is to evaluate:

```
(run "{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {call f 4}}}}")
```

  In the original evaluator to get convinced that 7 should be the correct result (note also that the same code, when translated into Racket, evaluates to **7**).

  (Yet, this is a very important optimization, which without it lots of programs become too slow to be feasible, so you might claim that you're fine with the modified semantics...)

- It does not allow using functions as objects, for example, we have seen that we have a functional representation for pairs:

```
(define (kons x y)
  (lambda (n)
    (match n
      ['first  x]
      ['second y]
      [else (error ...)])))

(define my-pair (kons 1 2))
```

  If this is evaluated in a dynamically-scoped language, we do get a function as a result, but the values bound to x and y are now gone!
  Using the substitution model we substituted these values in, but now they were only held in a cache which has no entries for them...

  In the same way, currying would not work, our nice `deriv' function would not work etc etc etc.

- Makes reasoning impossible, because any piece of code behaves in a way that *cannot* be predicted until run-time. For example, if dynamic scoping was used in Racket, then you wouldn't be able to know what this function is doing:

```
(define (foo)
  x)
```

  As it is, it will cause a run-time error, but if you call it like this:

```
(let ([x 1])
  (foo))
```

Then, it will return 1, and if you later do this:

```
(define (bar x)
  (foo))

(let ([x 1])
  (bar 2))
```

Then, you would get 2!

These problems can be demonstrated in Emacs Lisp too, but Racket goes one step further -- it uses the same rule for evaluating a function as well as its values (Lisp uses a different name-space for functions). Because of this, you cannot even rely on the following function:

```
(define (add x y)
  (+ x y))
```

to always add x and y! -- A similar example to the above:

```
(let ([+ -])
  (add 1 2))
```

would return -1!

Many so-called "scripting" languages begin their lives with dynamic scoping. The main reason, as we've seen, is that implementing it is extremely simple (no, *nobody* does substitution in the real world! (Well, *almost* nobody...)).

Another reason is that these problems make life impossible if you want to use functions as object like you do in Racket, so you notice them very fast -- but in a `normal' language without first-class functions, problems are not as obvious.

* For example, bash has `local' variables, but they have dynamic scope:

```
x="the global x"
print_x() { echo "The current value of x is \"$x\""; }
foo() { local x="x from foo"; print_x; }
print_x; foo; print_x
```

Perl began its life with dynamic scope for variables that are declared
`local':

```
$x="the global x";
sub print_x { print "The current value of x is \"$x\"\n"; }
sub foo { local($x); $x="x from foo"; print_x; }
print_x; foo; print_x;
```

When faced with this problem, "the Perl way" was, obviously, not to remove or fix features, but to pile them up -- so local *still* behaves in this way, and now there is a `my' declaration which achieves proper lexical scope...

There are other examples of languages that changed, and languages that want to change (e.g, nobody likes dynamic scope in Emacs Lisp, but there's just too much code now).

* This is still a tricky issue, like any other issue with bindings. For example, googling got me quickly to this site:

  http://www.hetland.org/python/instant-python.php

  which is confused about what "dynamic scoping" is... It claims that Python uses dynamic scope (Search for "Python uses dynamic as opposed to lexical scoping"), yet python always used lexical scope rules, as can be seen by translating their code to Racket (ignore side-effects in this computation):

```
(define (orange-juice)
  (* x 2))
(define x 3)
(define y (orange-juice)) ; y is now 6
(define x 1)
(define y (orange-juice)) ; y is now 2
```

  or by trying this in Python:

```
def orange_juice():
  return x*2
def foo(x):
  return orange_juice()
foo(2)
```

  The real problem of python (pre 2.1, and pre 2.2 without the funny
    from __future__ import nested_scope
  line) is that it didn't create closures, which we will talk about shortly.

* Another example, which is an indicator of how easy it is to mess up your scope is the following Ruby bug -- running in `irb':

```
% irb
irb(main):001:0> x = 0
=> 0
irb(main):002:0> lambda{|x| x}.call(5)
=> 5
irb(main):003:0> x
=> 5
```

  (This is a bug due to weird scoping rules for variables, which was apparently fixed in newer versions of Ruby. See
    http://ceaude.twoticketsplease.de/articles/
      ruby-and-the-principle-of-unwelcome-surprise.html
  for details on this and on other surprises in Ruby...)

* Another thing to consider is the fact that compilation is something that you do based only on the lexical structure of programs, since compilers never actually run code. This means that dynamic scope

makes compilation close to impossible.

* There are some advantages for dynamic scope too. Two notable ones are:

  - Dynamic scope makes it easy to have a "configuration variable" easily change for the extend of a calling piece of code (this is used extensively in Emacs, for example). The thing is that usually we want to control which variables are "configurable" in this way, statically scoped languages like Racket often choose a separate facility for these. To rephrase the problem of dynamic scoping, it's that *all* variables are modifiable.

    The same can be said about functions: it is sometimes desirable to change a function dynamically (for example, see "Aspect Oriented Programming"), but if there is no control and all functions can change, we get a world where no code can every be reliable.

  - It makes recursion immediately available -- for example,

```
{with {f {fun {x} {call f x}}} {call f 0}}
```

is an infinite loop with a dynamically scoped language. But in a lexically scoped language we will need to do some more work to get recursion going.

Implementing Lexical Scope: Closures and Environments

So how do we fix this?

The root of the problem: the new evaluator does not behave in the same way as the substituting evaluator.

In the old evaluator: functions can behave as objects that **remember values**. For example, when we do this:

```
{with {x 1}
  {fun {y}
    {+ x y}}}
```

The result was a function value, which actually was the syntax object for this:

```
{fun {y} {+ 1 y}}
```

If we call this function from someplace else like:

```
{with {f {with {x 1} {fun {y} {+ x y}}}}
  {with {x 2}
    {call f 3}}}
```

It is clear what the result will be: ***f*** is bound to a function that adds **1** to its input, so in the above the later binding for `**x**' has no effect at all.

With the caching evaluator: the value of

```
{with {x 1}
  {fun {y}
    {+ x y}}}
```

is simply:

```
{fun {y} {+ x y}}
```

There is no place where we save the 1 -- *that's* the root of our problem. (That's also what makes people suspect that using `lambda' in Racket and any other functional language involves some inefficient code-recompiling magic.) In fact, we can verify that -- by inspecting the returned value, and seeing that it does contain a free identifier.

Solution: We already know that we need an object that contains the body and the argument list (like the function syntax object).
Since we don't do substitutions – we need in addition to "**remember**" that we still need to substitute **x** by **1**.

This means that the pieces of information we need to know are:

- **formal argument(s):** {**y**}
- **body:** {**+ x y**}
- **pending substitutions:** [**1/x**]

That last piece has the missing **1**.
The resulting object is called a `**closure**' because it closes the function body over the substitutions that are still pending (its **environment**).

A first change: The info that we need to hold – **closures** (functions) need all three fields above (unlike the `**Fun**' case for the syntax object).

A second place that needs changing is the **when functions are called**. When we're done evaluating the `**call**' arguments (the function value and the argument value) but before we apply the function we have two ***values*** -- there is no more use for the current substitution cache at this point: we have finished dealing with all substitutions that were necessary over the current expression -- we now continue with evaluating the body of the function, with the new substitutions for the formal arguments and actual values given. But the body itself is the same one we had before -- which is the previous body with its suspended substitutions that we *still* did not do.

Rewriting evaluation rules -- all are the same except for evaluating a `**fun**' form and a `**call**' form:

```
  eval(N,sc)                 = N
  eval({+ E1 E2},sc)         = eval(E1,sc) + eval(E2,sc)
  eval({- E1 E2},sc)         = eval(E1,sc) - eval(E2,sc)
  eval({* E1 E2},sc)         = eval(E1,sc) * eval(E2,sc)
  eval({/ E1 E2},sc)         = eval(E1,sc) / eval(E2,sc)
  eval(x,sc)                 = lookup(x,sc)
  eval({with {x E1} E2},sc) = eval(E2,extend(x,eval(E1,sc),sc))
  eval({fun {x} E},sc)       = <{fun {x} E}, sc>
  eval({call E1 E2},sc1)
           = eval(Ef,extend(x,eval(E2,sc1),sc2))
                              if eval(E1,sc1) = <{fun {x} Ef}, sc2>
           = error!           otherwise
```

```
The Old rules -for reference
eval({fun {x} E},sc)       = {fun {x} E}
  eval({call E1 E2},sc)
           = eval(Ef,extend(x,eval(E2,sc),sc))
                              if eval(E1,sc) = {fun {x} Ef}
           = error!           otherwise
```

Environments

These substitution caches are a little more than "just a cache" now -- they actually hold an "**environment**" of substitutions in which expressions should be evaluated. So, we will switch to the common "**environment**" name now:

```
  eval(N,env)                = N
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
  eval(x,env)                = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env)
           = eval(Ef,extend(x,eval(E2, env),fenv))
                              if eval(E1,env) = <{fun {x} Ef}, fenv>
           = error!           otherwise
```

In case you find this easier to follow, the "flat algorithm" for evaluating a `**call**' is:

1. f := evaluate E1 in env1
2. if f is not a <{fun ...},...> closure then error!
3. a := evaluate E2 in env1
4. new_env := extend env_of(f) by mapping arg_of(f) to a
5. evaluate (and return) body_of(f) in new_env

Note how the scoping rules that are implied by this definition match the scoping rules that were implied by the substitution-based rules. (It should be possible to prove that they are the same.)

The changes to the code are almost trivial, except that we need a way to represent **<{fun {x} Ef}, env>** pairs.

The closure/environment interpreter

A new type for values (also for functions):
The implication of this change is that we now cannot use the same type for function syntax and function values since function values have more than just syntax. There is a simple solution to this -- we never do any substitutions now, so we don't need to translate values into expressions -- we can come up with a **new type for values, separate from the type of abstract syntax trees.**

When we do this, we will also fix our hack of using **FLANG** as the type of values: this was merely a convenience since the AST type had cases for all kinds of values that we needed. (In fact, you should have noticed that Racket does this too: numbers, strings, booleans, etc are all used by both programs and syntax representation (s-expressions) -- but note that function values are *__not__* used in syntax.) We will now implement a separate `**VAL**' type for runtime values.

Implementing a new type for run-time values

ENV type:

First, we need now a type for such environments -- we can use `**Listof**' for this:

```
;; a type for environments:
(define-type ENV = (Listof (List Symbol VAL)))
```

However, we can just as well define a new type for environment values:

```
(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])
```

Re-implementing `**lookup**' is now simple:

```
(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))
```

... We don't need `**extend**' because **we get `Extend' from the type definition**, and we also get `**(EmptyEnv)**' instead of `**empty-subst**'.

VAL type:

We now use this with the new type for values -- two variants of these:

```
(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV]) ; arg-name, body, scope
```

The new implementation of `eval': using the new type and implementing **lexical scope:**

```
(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV bound-id bound-body f-env); We expect a closure
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                      fval)]))]))
```

We also need to update `arith-op' to use **VAL** objects. The full code follows -- it now passes all tests, including the example that we used to find the problem.

```
---<<<FLANG-ENV>>>-----------------------------------------------------

;; The Flang interpreter, using environments

#lang pl

#|
The grammar:
  <FLANG> ::= <num>
            | { + <FLANG> <FLANG> }
            | { - <FLANG> <FLANG> }
            | { * <FLANG> <FLANG> }
            | { / <FLANG> <FLANG> }
            | { with { <id> <FLANG> } <FLANG> }
            | <id>
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }

Evaluation rules:
  eval(N,env)                = N
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
  eval(x,env)                = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                             if eval(E1,env1) = <{fun {x} Ef}, env2>
           = error!          otherwise
|#

(define-type FLANG
  [Num  Number]
  [Add  FLANG FLANG]
  [Sub  FLANG FLANG]
  [Mul  FLANG FLANG]
  [Div  FLANG FLANG]
  [Id   Symbol]
  [With Symbol FLANG FLANG]
  [Fun  Symbol FLANG]
  [Call FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)     (Num n)]
    [(symbol: name)  (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
```

```
        [(list 'fun (list (symbol: name)) body)
         (Fun name (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;; Types for environments, values, and a lookup function

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG ENV])

(: lookup : Symbol ENV -> VAL)
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (: NumV->number : VAL -> Number)
  (define (NumV->number v)
    (cases v
      [(NumV n) n]
      [else (error 'arith-op "expects a number, got: ~s" v)]))
  (NumV (op (NumV->number val1) (NumV->number val2))))

(: eval : FLANG ENV -> VAL)
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id name) (lookup name env)]
    [(Fun bound-id bound-body)
```

```
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV bound-id bound-body f-env)
          (eval bound-body
                (Extend bound-id (eval arg-expr env) f-env))]
         [else (error 'eval "`call' expects a function, got: ~s"
                              fval)]))]))

(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str) (EmptyEnv))])
    (cases result
      [(NumV n) n]
      [else (error 'run
                   "evaluation returned a non-number: ~s" result)])))

;; tests
(test (run "{call {fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{with {add3 {fun {x} {+ x 3}}}
               {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
               {with {add1 {fun {x} {+ x 1}}}
                 {with {x 3}
                   {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
               {with {foo {fun {x} {+ x 1}}}
                 {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
               {with {f {fun {y} {+ x y}}}
                 {with {x 5}
                   {call f 4}}}}")
      => 7) ;; the example we considered for subst-caches
(test (run "{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4}")
      => 7)
(test (run "{call {call {fun {x} {call x 1}}
                       {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

Fixing an overlooked Bug

Incidentally, this version fixes a bug we had previously in the **substitution version** of FLANG:

```
(run "{with {f {fun {y} {+ x y}}}
        {with {x 7}
          {call f 1}}}")
```

This bug was due to our naive `subst', which doesn't avoid capturing renames. But note that since that version of the evaluator makes its way from the outside in, there is no difference in semantics for *valid* programs -- ones that don't have free identifiers.

(Reminder: This was ***not*** a dynamically scoped language, just a bug that happened when `**x**' wasn't substituted away before `**f**' was replaced with something that refers to `**x**'.)
