Environments

- PLAI Chapter 6
- Evaluating using substitutions is very inefficient
- To work around this, we want to use a cache of substitutions.
- We begin evaluating with no cached substitutions, then collect them as we encounter bindings.
- When we reach an identifier it is no longer an error – we must consult the substitution cache at that point.

1. When evaluating with substitutions, 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**!).

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:

```
1  ;; a type for substitution caches:
(define-type Binding
             [bind (name : Symbol) (val : FLANG)])
(define-type-alias SubstCache (Listof Binding))
```

- We an empty substitution cache, and a way to extend

2
```
(define empty-subst empty)

(define (extend id expr sc)
  (cons (bind id expr) sc))
```

- And a way to look things up:

3
```
(define (lookup name sc)
  (let ([first-name (λ () (bind-name (first
      sc)))]
        [first-val (λ () (bind-val (first
          sc)))])
    (cond
      [(empty? sc) (error 'lookup
        (string-append "missing binding: "
        (to-string name)))]
      [(eq? name (first-name)) (first-val)]
      [else  (lookup name (rest sc))])))
```

Formal Rules for Cached Substitutions

- The formal evaluation rules are now different.
- Evaluation carries along a "substitution cache"
- Our substition rules are replaced by:


```
lookup(x,empty-subst)      = error!
lookup(x,extend(x,E,sc))   = E
lookup(x,extend(y,E,sc))   = lookup(x,sc)
                                  if `x' != `y'
```

- Now we can write the new rules for ‘eval’.
- The change to arithmetic and `fun` is small, just pass an extra parameter.


```
eval(N,sc)                  = N
eval({+ E1 E2},sc)          = eval(E1,sc) +
                              eval(E2,sc)

eval({fun {x} E},sc)        = {fun {x} E}
```

- Identifiers need to be looked up:

6
```
eval(x,sc)                    = lookup(x,sc)
```

- `subst` is replaced by `extend` and `lookup`

7
```
eval({with {x E1} E2},sc) =
                 eval(E2,extend(x,eval(E1,sc),sc))

eval({call E1 E2},sc)
          = eval(Ef,extend(x,eval(E2,sc),sc))
                  if eval(E1,sc) = {fun {x} Ef}
          = error!              otherwise
```

Evaluating with Substitution Caches 1/2

8

```
;; evaluates FLANG expressions by reducing them to
   expressions
(define (eval expr sc)
  (type-case FLANG expr
    [(Num n) expr]
    [(Add 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)]
```

Evaluating with Substitution Caches 1/2

```
;; evaluates FLANG expressions by reducing them to
   expressions
(define (eval expr sc)
  (type-case FLANG expr
    [(Num n) expr]
    [(Add 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)]
```

1. 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 replaces 'subst', and therefore specifies 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.

Evaluating with Substitution Caches 2/2

9

```
;; evaluates FLANG expressions by reducing them to
   expressions
(define (eval expr sc)
  (type-case FLANG expr
    [(Fun bound-id bound-body) expr]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr sc)])
       (type-case FLANG fval
         [(Fun bound-id bound-body)
          (eval bound-body
                (extend bound-id (eval arg-expr
                   sc) sc))]
         [else (error 'eval
                      (string-append "`call'
                         expects a function, got:
                         "
                              (to-string
                                 fval)))]))]))
```

- we don't need 'subst' anymore, but the rest of the code (the data type definition, parsing, and 'arith-op') is exactly the same.
- we need to make sure that 'eval' is initially called with an empty cache.

10

```
(define (run s-exp)
  (let ([result (eval (parse-sx s-exp)
                      empty-subst)])
    (type-case FLANG result
      [(Num n) n]
      [else (error 'run
                   (string-append "evaluation
                                   returned a non-number: "
                                  (to-string
                                   result)))])))
```

First, some fairly fancy looking things work:

11
```
(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)
```

- By tracing, we see the substution cache acts like a stack

12

```
(trace lookup)
(test (run `{with {identity {fun {x} x}}
                  {with {foo {fun {x} {+ x 1}}}
                    {call {call identity foo}
                          123}}})
      124)
```

OTOH, we have three failing tests; the reasons look different, but turn out to be related.

13
```
(test (run `{with {x 3}
                  {with {f {fun {y} {+ x y}}}
                    {with {x 5}
                      {call f 4}}}})     7)
(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

- Scope has been problem for **lots** of language implementors, including the first version of Lisp.
- What **should** the following evaluate to:


```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {call f 4}}}}
```

- The answer depends on whether we have **dynamic** or **static** scope.
- **Static** Scope (also called Lexical Scope): In a language with static scope, each identifier gets its value from the scope of its definition, not its use.
- **Dynamic** Scope: In a language with dynamic scope, each identifier gets its value from the scope of its use, not its definition.
- Which is our new (caching) evaluator?
- Racket uses lexical scope, our new evaluator uses dynamic, the old substitution-based evaluator was static etc.

- The answer depends on whether we have **dynamic** or **static** scope.
- **Static** Scope (also called Lexical Scope): In a language with static scope, each identifier gets its value from the scope of its definition, not its use.
- **Dynamic** Scope: In a language with dynamic scope, each identifier gets its value from the scope of its use, not its definition.
- Which is our new (caching) evaluator?
- Racket uses lexical scope, our new evaluator uses dynamic, the old substitution-based evaluator was static etc.

1. As a 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 one of the only **live** dialects of Lisp that is still dynamically scoped by default.

- Compare a version of the above code in Racket:

15
```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 4))))
```

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

16
```
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (funcall f 4))))
```

- what 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))))
```

- note that renaming identifiers can lead to different results – 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))))
```

- Consider this Emacs Lisp function:


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

(return-x) ;; Error

(let ((x 5)) (return-x)) ;; OK

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

(foo 5) ;; OK
```

- Fun with dynamic scoped racket


```
(require plai/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)
```

21

```
(require plai/dynamic)

(define (foo x)
  (define (helper) (+ x 1))
  helper)

(define x 123)
(test ((foo 0)) 124)

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

Dynamic versus Lexical Scope

- Leaving aside the question of whether dynamic scope is good for your mental health, what matches our original substitution semantics?
- Remember this was supposed to be an **optimization**
- Consider our nemesis again, under the slow substitution evaluator:


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

Dynamic versus Lexical Scope

- Leaving aside the question of whether dynamic scope is good for your mental health, what matches our original substitution semantics?
- Remember this was supposed to be an **optimization**
- Consider our nemesis again, under the slow substitution evaluator:

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

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

- As an example what you lose, dynamic scope does not allow using functions with saved values or **closures** (more on this in Lecture 8)
- In a dynamic scoped language, you don't even know if this is valid code until run time:


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

- As an example what you lose, dynamic scope does not allow using functions with saved values or **closures** (more on this in Lecture 8)
- In a dynamic scoped language, you don't even know if this is valid code until run time:

23

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

1. Subsitution caching 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...

- Racket uses the same rule for evaluating a function as well as its values This makes dynamic scope more powerful/dangerous:


```
(require plai/dynamic)
(define (add x y)
        (+ x y))

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

- Racket uses the same rule for evaluating a function as well as its values This makes dynamic scope more powerful/dangerous:

```
(require plai/dynamic)
(define (add x y)
      (+ x y))

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

1. Lisp-2's uses a different name-space for functions

- Many so-called "scripting" languages begin their lives with dynamic scoping.
- In languages without first-class functions, problems of dynamic scope 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
```

- Many so-called "scripting" languages begin their lives with dynamic scoping.
- In languages without first-class functions, problems of dynamic scope 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
```

1. The main reason for starting with dynamic scope, as we've seen, is that implementing it is extremely simple (no, **nobody** does substitution in the real world! (Well, **almost** nobody...)).

- Perl began its life with dynamic scope for variables that are declared ‘local’:

26
```
$x="the global x";
sub print_x { print "The current value of x is
    \"$x\"\n"; }
sub foo { local($x); $x="x from foo"; print_x;
    }
print_x; foo; print_x;
```

- local and some builtin variables **still** have dynamic scope, but `my` introduces lexical scope.

27
```
my $x="the global x";
sub print_x { print "The current value of x is
    \"$x\"\n"; }
sub foo { my $x="x from foo"; print_x; }
print_x; foo; print_x;
```

- in Emacs 24, emacs lisp acquires optional lexical scope.

28
```
;-*- mode: emacs-lisp; lexical-binding: t-*-
(let ((x 3))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (funcall f 4))))
```

- Early versions of python ($\langle = 2.1$) printed 1 for;

29
```
x = 1
def f1():
    x = 2
    def inner():
        print x
    inner()
f1()
```

- Note that this is *not* dynamic scope either, since the following fails to compile

30
```
def orange_juice():
  return x*2
def foo(x):
  return orange_juice()
foo(2)
```

- Where the following works, in (dynamic) racket

31
```
(require plai/dynamic)
(define (orange-juice)
  (* x 2))
(define (foo x)
  (orange-juice))
(foo 2)
```

- Another example, which is an indicator of how easy it is to mess up your scope is the following Ruby 1.8 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
```

There are some advantages for dynamic scope.

- Dynamic scope makes it easy to have a "configuration variable" easily change for the extent of a calling piece of code.
- Optional dynamically scoped variables are useful
- the problem of dynamic scoping is that **all** variables are modifiable.
- It is sometimes desirable to change a function dynamically (for example, see "Aspect Oriented Programming"), but if all functions can change, no code can be reliable.
- It makes recursion immediately available – for example, dynamic scope gives us easy loops:


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

Controlled Dynamic Scope

- racket provides "parameters" for dynamic scopy, internally used by plai/dynamic


```
(define location (make-parameter "here"))
(define (foo)
  (location))
(parameterize ([location "there"])
  (foo))
(foo)

(parameterize ([location "in a house"])
  (list (foo)
        (parameterize ([location "with a
           mouse"])
          (foo))
        (foo)))
(location)
```