CS4613 Lecture 3

David Bremner

January 14, 2025

Local Binding Examples

What can we learn by comparing these two programs in stacker? p. 47

```
(deffun (f x)
  (let ([y 2])
    (+ x y)))
(f 7)
```

stacker

```
(deffun (f x)
  (defvar y 2)
  (+ x y))
(f 7)
```

stacker

Local Binding Examples

What can we learn by comparing these two programs in stacker? p. 47

stacker
```
(deffun (f x)
  (let ([y 2])
    (+ x y)))
(f 7)
```

stacker
```
(deffun (f x)
  (defvar y 2)
  (+ x y))
(f 7)
```

1. Compare syntax in other languages
2. What about execution?

A simplified local binding syntax

Let's look ahead a bit to Desugaring and define a compatible syntax to the book. p. 72

```
(define-syntax-rule (let1 (var expr) body)
  (let ([var expr]) body))
```

Now we can look at how some examples should work p. 49

ex1

```
{let1 {x 1} {+ x x}}

{let1 {x 1}
      {let1 {y 2}
            {+ x y}}}
```

A simplified local binding syntax

Let's look ahead a bit to Desugaring and define a compatible syntax to the book. p. 72

```
(define-syntax-rule (let1 (var expr) body)
  (let ([var expr]) body))
```

Now we can look at how some examples should work p. 49

```
{let1 {x 1} {+ x x}}

{let1 {x 1}
      {let1 {y 2}
            {+ x y}}}
```

1. this is a bit subtle. In some sense we have implimented the feature, but not as part of our interpreter

Inner and Outer Scope

p. 50

ex2

```
{let1 {x 1}
      {let1 {y 2}
            {let1 {x 3}
                  {+ x y}}}}
```

- What feature does this example introduce?
- Where can we find this feature in other languages?

ex3

```
{let1 {x 1}
      {+ x
         {let1 {x 2} x}}}
```

Inner and Outer Scope


```
{let1 {x 1}
      {let1 {y 2}
            {let1 {x 3}
                  {+ x y}}}}
```

- What feature does this example introduce?
- Where can we find this feature in other languages?

```
{let1 {x 1}
      {+ x
         {let1 {x 2} x}}}
```

1. We can use DrRacket to trace the bindings
2. We don't need to rewrite things in racket, because we cheated and changed the syntax of plait to match our examples

Static Scoping

p. 52

scope1

```
(defvar x 1)
(deffun (f)
  (+ x 1))

(let ([x 2])
  (f))
```

stacker

Static Scope

Variable binding is determined by position in the source program, not order of execution.

Static Scoping

p. 52

```
(defvar x 1)
(deffun (f)
  (+ x 1))

(let ([x 2])
  (f))
```

Static Scope

Variable binding is determined by position in the source program, not order of execution.

1. The book uses a different set of examples for dynamic scope, but for me these go beyond dynamic scope by not obeying the block structure of `let`

Dynamic scope

scope2

```
(defvar x 1)
(deffun (f)
  (+ x 1))

(let ([x 2])
  (f))
```

Dynamic scope

Binding is determined by execution environment.

Dynamic scope makes many traps

scope3

```
(deffun (blah func val) (func val))
(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

scope4

```
(deffun (blah func x) (func x))
(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

Dynamic scope makes many traps

scope3
```
(deffun (blah func val) (func val))
(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

scope4
```
(deffun (blah func x) (func x))
(let ([x 3])
  (let ([f (λ (y) (+ x y))])
    (let ([x 5])
      (blah f 4))))
```

1. Can you see what changed between the two examples? They are run with the same interpreter (i.e. the same `#lang`)

Controlled Dynamic Scope

DETOUR

- plait has parameters for dynamic scope
- internally used by smol/dyn-scope-is-bad

scope5

```
(define location (make-parameter "here"))
(define (foo) (parameter-ref location))
(parameterize ([location "there"]) (foo))
(foo)
(parameterize ([location "in a house"])
  (list (foo)
        (parameterize ([location "with a mouse"])
          (foo))
        (foo)))
(parameter-ref location)
```

Update AST

p. 54

```
(define-type Exp
  [numE (n : Number)]
  [plusE (left : Exp) (right : Exp)]
  [timesE (left : Exp) (right : Exp)]
  [varE (name : Symbol)]          ;; new
  [let1E (var : Symbol)           ;; new
         (value : Exp)
         (body : Exp)])
```

Environments

How to interpret variables?

p. 55

```
(define (interp e)
  (type-case (Exp) e
    [(numE n) n]
    [(varE s) ....]))
```

Let's take a closer look at how stacker evaluates let:

```
(let ([y 2])
  (+ 7 y))
```

stacker

Implementing environments

We will use hash tables to implement environments

p. 55

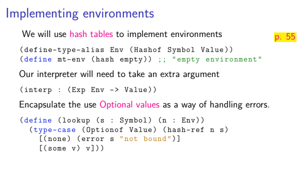
```
(define-type-alias Env (Hashof Symbol Value))
(define mt-env (hash empty)) ;; "empty environment"
```

Our interpreter will need to take an extra argument

```
(interp : (Exp Env -> Value))
```

Encapsulate the use Optional values as a way of handling errors.

```
(define (lookup (s : Symbol) (n : Env))
  (type-case (Optionof Value) (hash-ref n s)
    [(none) (error s "not bound")]
    [(some v) v]))
```



1. Somewhere along the way `calc` was renamed to `interp`
2. The extra argument is mainly for use in recursive evaluations of sub-expressions
3. There is many debates about the best way to handle errors. In this simple intepreter it is easiest to throw an (uncaught) exception to report an unbound variable

Evaluation strategy

Checking our example again

p. 56

```
(let ([y 2])
  (+ 7 y))
```

stacker

We need to

1. evaluate the body of the expression, in
2. an environment that has been extended, with
3. the new name
4. bound to its value.

Extending environments

p. 56

Encapsulating some more hash-table manipulation

```
(define (extend old-env new-name value)
  (hash-set old-env new-name value))

[(let1E var val body)
 (let ([new-env (extend nv                   ;; 2
                        var                  ;; 3
                        (interp val nv))])   ;; 4
   (interp body new-env))]                   ;; 1
```

1. evaluate the body of the expression, in
2. an environment that has been extended, with
3. the new name
4. bound to its value.

Interpreter for `let1`

let1

```
(define (interp e nv)
  (type-case Exp e
    [(numE n) n]
    [(varE s) (lookup s nv)]
    [(plusE l r) (+ (interp l nv) (interp r nv))]
    [(timesE l r) (* (interp l nv) (interp r nv))]
    [(let1E var val body)
     (let ([new-env (extend nv
                            var
                            (interp val nv))])
       (interp body new-env))]))
```

Extending the parser

let1

```
[(? `(let1 (SYMBOL ANY) ANY))
 (let* ([def (sx 1)]
        [parts (s-exp->list def)]
        [var (s-exp->symbol (list-ref parts 0))]
        [val (parse (list-ref parts 1))]
        [body (px 2)])
   (let1E var val body))]
```