

Dynamic Scope

A language is *dynamically scoped* if variables are always bound to their most recent value.

Example:

```
let a = 3
in let p = proc (x) +(x, a);
    a = 5
    in *(a, p(2))
```

returns 35 with a dynamic scope, 25 with a static scope.

Dynamic scope is easier to implement, but can make programs more difficult to understand.

```

(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (apply-env env var)))
      (app (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args env)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (proc (formals body) exp)
      (varassign (var exp)
        (cell-set! (apply-env env var) (eval-exp exp env)))
      (else (error "Invalid abstract syntax:" exp))))))

```

```

(define apply-proc
  (lambda (proc args current-env)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (proc (formals body)
        (eval-exp body
          (extend-env formals
            (map make-cell args)
            current-env)))
      (else (error "Invalid procedure:" proc))))))

```

Global environment stack

Dynamic scoping requires only one global environment, which can be implemented as a stack (LIFO) using the following ADT:

`(lookup-in-env s)` returns the most recent value bound to symbol *s*.

`(push-env! los lov)` pushes a set of new bindings onto the global environment: *los* is a list of symbols, and *lov* is the corresponding list of values. The return value is undefined.

`(pop-env!)` removes the last set of bindings from the global environment. The return value is undefined.

Interpreter with a global environment

```
(define eval-exp
  (lambda (exp)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (lookup-in-env var)))
      (app (rator rands)
           (let ((proc (eval-exp rator))
                 (args (eval-rands rands)))
             (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
          (if (true-value? (eval-exp test-exp))
              (eval-exp then-exp)
              (eval-exp else-exp)))
      (proc (formals body) exp)
      (varassign (var exp)
                 (cell-set! (lookup-in-env var) (eval-exp exp)))
      (else (error "Invalid abstract syntax:" exp)))))

(define eval-rands
  (lambda (rands)
    (map (lambda (exp) (eval-exp exp))
         rands)))
```

```
(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (proc (formals body)
        (push-env! formals (map make-cell args))
        (let ((value (eval-exp body)))
          (pop-env!)
          value))
      (else (error "Invalid procedure:" proc))))))
```

Dynamic Assignment

Dynamic assignment can be used to temporarily override a lexical scoped variable reference.

var := *exp* during *body*

evaluates *body* with the proviso that every reference to variable *var* encountered is replaced by the value of expression *exp*.

$\langle \text{exp} \rangle ::= \langle \text{var} \rangle := \langle \text{exp} \rangle \text{ during } \langle \text{exp} \rangle$
dynassign (var exp body)

```
let x = 4
in let p = proc (y) +(x, y)
    in +(x := 7 during p(1),
        p(2))
```

returns 14.