

Comp 311
Principles of Programming Languages
Lecture 6
Implementing Syntactic Interpreters

Corky Cartwright
September 3, 2010



A Syntactic Evaluator

Now we can translate our rules into a program? Here is a sketch:

```
;; R → R ; an illegal program can return an AST (type R)
(define eval
  (lambda (M)
    (cond
      ((var? M) M) ; M is a free var
      ((or (const? M) (proc? M)) M) ; M is a value
      ((add? M) ; M has form (+ l r)
       (add (eval (add-left M)) (eval (add-right M))))
      (else ; M has form (N1 N2)
       (apply (eval (app-rator M)) (eval (app-rand M)))))))

;; Proc V → R
(define apply
  (lambda (a-proc a-value)
    (cond
      ((not (proc? a-proc)) ; ill-formed app
       (make-app a-proc a-value))
      (else (eval (subst a-value ; return substituted body
                        (proc-param a-proc)
                        (proc-body a-proc)))))))
```



Coding Substitution

```
;; V Sym R → R
(define subst
  (lambda (v x M)
    (cond
      [(var? M) (cond [(equal? (var-name M) x) v] [else M])]
      [(const? M) M]
      [(proc? M)
       (cond [(equal? x (proc-param M)) M]
              [else (make-proc (proc-param M)
                                (subst v x (proc-body M)))])]
      [(add? M) (make-add (subst v x (add-left M))
                           (subst v x (add-right M)))]
      [else ;; M is (N1 N2)
       (make-app (subst v x (app-rator M))
                  (subst v x (app-rand M)))]))])
```

Is **subst** safe? No! It is oblivious to free variables in **M**.

Exercise: Revise **subst** so that it is safe.



Comments on Syntactic Interpreter

- Still need to define **add**. What does **add** do on non-numbers?
- The key property of this evaluator is that it only manipulates (abstract) syntax. It specifies the meaning of LC by mechanically transforming the syntactic representation of a program.
- This approach only assigns a satisfactory meaning to complete LC programs, not to subtrees of complete programs.
Counterexample:

((lambda (x) (+ x y) 7)

If **add** mirrors syntactic evaluation, then it will return **(+ 7 y)**. Otherwise, it will generate a run-time error because **y** is not a value.

In a context where **y** is bound to **5**, it returns **12**; not **(+ 7 y)** or a run-time error.



Toward Semantic Interpretation

- From a software engineering perspective, what is wrong with our syntactic interpreter?
 - ♦ How fast is **subst**? How can we do better?
 - ♦ Avoid unnecessary substitutions by keeping a table of bindings.

```
;; Binding = (make-Binding Sym V)           ; Note: Sym not Var
;; Env = (listOf Binding)
;; R Env → V
(define eval
  (lambda (M env)
    (cond
      ((var? M) (lookup (var-name M) env))
      ((or (const? M) (proc? M)) M)
      ((add? M) ; M has form (+ l r)
       (add (eval (add-left M) env) (eval (add-right M) env)))
      (else ; M has form (N1 N2)
       (apply (eval (app-rator M) env) (eval (app-rand M) env) env))))))

;; Proc V Env → V
(define apply
  (lambda (a-proc a-value env)
    (eval (proc-body a-proc) (cons ((proc-param a-proc) a-value) env))))
```



Gotcha's in Semantic Interpretation

- What if **a-proc** contains free variables? Do we always get the right answer (as defined by syntactic interpretation)?

- Illustration:

```
(let [(a 5)
      (app-to-a (lambda (f) (f a)))]
  (let [(a 10)]
    (+ a (app-to-a (lambda (x) x))))))
```

- What goes **wrong**?
- Think about how you might fix the problem

Illustration in Standard Scheme (RnRS)

```
(let* [(a 5)
      (app-to-a (lambda (f) (f a)))]
  (let [(a 10)]
    (+ a (app-to-a (lambda (x) x))))))
```

What does **a** mean in the definition of **app-to-a**?

Scheme Binding (Scoping) Constructs

- In Scheme,

(let [(v1 M1) ... (vn Mn)] N)

abbreviates

(lambda (v1 ... vn) N) M1 ... Mn)

- Similarly,

(let* [(v1 M1) ... (vn Mn)] N)

abbreviates

(let [(v1 M1)] (let ... (let [(vn Mn)] N) ...))

- And

(letrec [(v1 M1) ... (vn Mn)] N)

means **v1 ... vn** are bound recursively, *i.e.*, **v1 ... vn** are in scope in **M1 ... Mn** as well as in **N**.

