Comp 411
Principles of Programming Languages
Lecture 6
Implementing Syntactic Interpreters

Corky Cartwright

January 29, 2014

A Syntactic Evaluator

Can we translate our syntactic reduction rules into a program?

```
;; R → R   ; an illegal program can return an AST (type R)
(define eval
  (lambda (M)
    (cond
      ((var? M) M)                          ; M is a free var (stuck!)
      ((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))           ; return stuck state
      (else (eval (subst a-value            ; return substituted body
                         (proc-param a-proc)
                         (proc-body a-proc))))))))
```

Coding Substitution

```
;; V Sym R → R      Substitutes v for x in M
(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. Note that blind substitution works as long as our top-level **`M`** is well-formed and contains no free variables. Why?

Comments on Syntactic Interpreter

Still need to define **`add`**. What does **`add`** do on non-**`const`** values? 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. Meaning of sub-expressions should be defined so that meaning $[\![\bullet]\!]$ is compositional, *i.e.* ,

$$[\![\ \texttt{(c}\ \texttt{M}_1 \ldots \texttt{M}_k\ \texttt{)}\ ]\!] = [\![\ \texttt{c}\ ]\!]\ (\ [\![\ \texttt{M}_1]\!], \ldots , [\![\ \texttt{M}_k]\!])$$

Syntactic interpretation utterly fails in this regard.

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))))
```

More Readable Notation for Lambda Expressions

- In essentially all functional languages for software development, there is alternate notation for

  ```
  ((lambda (x) M) N)
  ```

  namely

  `(let [(x N)] M)` Scheme

  or

  `let x := N; in M` Jam

- This alternate notation is literally an abbreviation for the explicit `lambda` form

- In this alternate notation, the beta-reduction rule has the form

  `(let [(x V)] M) ⇒ M[x := V]` Call-by-value

  `(let [(x N)] M) ⇒ M[x := N]` Call-by-value

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)]
  (let [(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. Hint: what information is missing in **`env`** when **`a-proc`** is evaluated? Remember, you want the same result as if you were performing syntactic interpretation.