Comp 311
Principles of Programming Languages
Lecture 5
Syntactic Interpreters

Corky Cartwright

September 1, 2010

A Syntactic Interpreter for LC

- Expand LC to make slightly more realistic language by adding one primitive binary operator **+**

  ```
  M :== x | n | (lambda (x) M) | (M M) | (+ M M)
  ```

  A *proper* LC program is an LC expression M that is *closed*, *i.e.*, contains no *free* variables. An LC program is any LC expression.

- We similarly extend the abstract syntax for LC given in the notes to include addition as a primitive application. In Scheme, the set **R** of abstract representations is defined by the equations:

  ```
  R = Var | Const | Proc | App | Add
  Var = (make-var Sym)
  Const = make-const Num)
  Proc =  (make-proc Var R)
  App =  (make-app R R)
  Add = (make-add R R)
  ```

  where Sym and Num are Scheme symbols and numbers, respectively, and we have defined the Scheme data types:

  ```
  (define-struct var (name))
  (define-struct const (number))
  (define-struct proc (param body))
  (define-struct app (rator rand))
  (define-struct add (left right))
  ```

Syntactic Interpretation

What does syntactic interpretation do?

- Reduce the AST for a complete program to a *value.*

- What is a value? A special AST representing a data constant. In LC (a subset of Scheme), a value V is either a number or a procedure:

```
V ::=  n  |  (lambda (x) M)
```

- What are the Scheme evaluation rules (from Comp 210/211) that are relevant to LC?

A Syntactic Interpreter for LC

Basic Rules of Evaluation

- Rule 1: For applications of the binary operator + to two arguments that are numeric values, replace the application by the sum of the two arguments.
- Rule 2: For applications of a lambda-expression to a value, substitute the argument for the parameter in the body, *i.e.*,

    ```
    ((lambda (x) M) V)   --->   M[x := V]
    ```

    where `M[x := V]` means `M` with all *free* occurrences of `x` replaced by `V`.

Observation: the definition of *value* has a major impact on evaluation

What happens if we define

```
V :== n | (lambda (x) V)
```

Some evaluation strategies for the untyped lambda-calculus do this, but they have not proven relevant to defining the semantics of real programming languages. Why?

What if we allow arguments in procedure application reductions that aren't values.

Example:

```
((lambda (y) 5) ((lambda (x) (x x)) (lambda (x) (x x))))
```

This is a sensible choice in *functional* languages that prohibit side effects (the values of bound variables and fields never change). Haskell does this.

Syntactic Interpreter for LC cont.

Combining evaluation rules:

- Given an LC expression, we evaluate it by repeatedly applying the preceding rewrite rules until we get an answer.

- What happens when we encounter an expression to which more than one rule applies? In Core Scheme (the Comp 211 dialect) and LC, the leftmost rule always takes priority.

- Other strategies are possible. Some "syntactic" (rewrite-rule-based) semantics for complex languages define formal syntactic rules (called evaluation contexts) to determine which reduction is done first.

Gotcha's in Syntactic Semantics

- In Rule 2 (called "beta-reduction" in the λ-calculus and program semantics literature), we confined substitution in the definition of `M[x := V]` to the *free* occurrences of `x` in `M`.

- If we had not confined substitution to *free* occurrences, the rule would have produced strange results, destroying the meaning of bound variables in `M`.

- If we use Rule 2 to transform programs (replacing "equals by equals"), we must be particularly careful in how we perform the substitution of `V` for `x` in `M`. If not, free variables in `V` can be "captured" in replaced occurrences of `x` in `M`. Consider the following example:

```
(lambda (y) x) [x := y]
```

If we replace all free occurrences of x by y, we get

```
(lambda (y) y)
```

which is wrong!   The occurrence of `y` in `[x := y]` is *free.* Presumably, it is bound somewhere in the context surrounding

```
(lambda (y) x) [x := y]
```

If we substitute `y` for the free occurrence of `x` in `(lambda (y) x)`, it becomes bound by a local definition of `y`. The *solution* is to rename the variable introduced in the local definition of `y` as a fresh variable name, say `z`.

```
(lambda (y) x) [x := y]  -->  (lambda (z) y)
```

Safe Substitution

This revised substitution process (renaming local variables that would otherwise capture free occurrences in the expression being substituted) is called *safe substitution*. The results produced by safe substitution are non-deterministic in a trivial sense because the choices for the new names of renamed local variables are arbitrary (as long as they are *fresh, i.e. distinct from existing variables in the program text involved in the substitution)*. Hence,

```
(lambda (y) x) [x := y]   -->   (lambda (z) y)
                          -->   (lambda (u) y)
                          -->   (lambda (v) y)
                          -->   ...
```

Lambda-Calculus Notation

The mathematical literature on the lambda calculus uses a leaner (perhaps more human friendly) notation for lambda-expressions. The syntax is:

```
M :: = Var | (M M) | (λ Var . M)
```

In applications and abstractions, parentheses may be elided if no ambiguity results (assuming left associativity and maximal expression extent—as in parsing JAM). When parentheses are elided in applications, application associates to the *left, i.e.,*

```
λx . x y     abbreviates   (λx . (x y))
L M N        abbreviates   ((L M) N)
```

α-Conversion (Variable Renaming)

The renaming of bound variables is an axiom (assumed equation) of the λ-calculus:

$$\lambda x \; . \; M \quad \rightarrow \quad \lambda y \; . \; M[x:=y]$$

where y does not occur in M. Hence, α-conversion does not presume any notion of safe substitution as primitive. Safe substitution can be defined using α-conversion.

α-conversion and β-reduction are the fundamental equations (laws) of the λ-calculus. Mathematicians often do not directly define safe substitution. They simply stipulate that β-reduction is not allowed to capture bound variables. Hence, an α-conversion may sometimes be necessary prior to a β-reduction. These conversions are typically assumed without comment.