Comp 311
Principles of Programming Languages
Lecture 5
Syntactic Interpreters

Corky Cartwright

January 27, 2014

A Syntactic Interpreter for LC

- Recall our definition of the LC language (tweaked to be a subset of Scheme):

  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.

- This is conventional CFG definition but it parallels an abstract syntax deifnition (trees).

  The set R of abstract representations is defined bv the equation:

  R = (make-var Var) | (make-const Num) | (make-proc Var R) |
  (make-app R R) | (make-add R R)

  where we have defined the following 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?

What is a value? An AST representing a data constant.

- Reduce the AST for a complete program to a *value*. Every intermediate expression is a complete program. Our semantics simply rewrites program text.
- What is a value? An AST representing a data constant. In LC (which 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) that are relevant to LC?

A Syntactic Interpreter for LC cont.

Basic Rules of Evaluation

- Rule 1: For applications of the binary operator `+` to two arguments that are *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 (thevalues 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 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 210 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 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 much more careful in how we perform the substitution of `V` for `x` in `M`. In particular, free variables in `V` can be "captured in replaced occurrences of `x` in `M`. Consider the following example:

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

If we perform naïve beta-reduction replacing all free occurrences of `x` in

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

by `y`, we get

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

When 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`. Hence,

```
(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).*