Comp 411
Principles of Programming Languages
Lecture 5
Syntactic Interpreters

Corky Cartwright

January 20, 2023

A Syntactic Interpreter for LC

- Recall our definition of the LC language:

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

  where `x` is any variable in `Var` and `n` is a number (integer) in `Num`. An LC *program* is an LC expression M that is *closed*, *i.e.*, contains no *free* variables.

- The preceding is a conventional CFG definition but it can be interpreted as an inductive definition of abstract syntax trees because all constructed (non atomic) expressions are enclosed in parentheses, showing the precise structure of the corresponding AST.

- The set R of abstract representations is defined by 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))
  ```

  `var` and `const` are constructors with one field

  `proc`, `app`, `add` are constructors with two fields

Syntactic Interpretation

- What does syntactic interpretation do? Reduce ASTs to values.
- Syntactic interpretation reduces the AST for a program to a *value*. We arrange our evaluation rules so that every expression in the chain of expressions produced by the reduction process is a program. Our semantics simply rewrites entire programs until the result is a value. Note that every value is a program.
- What is a *value*? An AST representing a data constant. In LC (effectively a subset of Scheme), a value `V` is either a number or a procedure:

  ```
  V :== n | (lambda x M)
  ```
- What are the Racket/Scheme evaluation rules (from Comp 311) that are relevant to LC?

A Syntactic Interpreter for LC cont.

Basic Rules of Evaluation for *call-by-value* (universal in mainstream languages)

- Rule 1: For applications of the binary operator `+` to two arguments that are numbers, replace the application by the sum of the two arguments (a number).
- Rule 2: For applications of a lambda-expression to a value (defined earlier), 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. For example, we conceivably could define

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

  (`v` is a (free) variable)

- The preceding definition of value in a technical context later in this course but ambda-calculus essentially do this, but it is not a reasonable way to defin 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 rules until we get an answer.
- What happens when we encounter an expression to which more than one rules applies? In our framework, 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. For our purposes, simply mandating the leftmost reduction is sufficient.

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).*
- **Question to Ponder: A**ssume that we use deBruijn notation (static distance coordinates in LC instead of named variables. How would we define substitution, *i.e.*, what happens to the deBruijn indices during the substitution process? (… the cost of deBruijn notation)
- **Further Reading:** The safe substitution process hardly qualifies as a simple local modification of the program being reduced. In the literature on reduction in the lambda calculus, there are some interesting papers that explicate the details of the safe substitution operation breaking it down into smaller steps that truly are simple local modifications. See https://dl.acm.org/citation.cfm?id=96712

Introducing `let`, a Common Syntactic Abbreviation

- In essentially all functional languages for software development, there is alternate special notation for
  `((lambda x M) N)`
  namely
  `(let [(x N)] M)` Scheme/Racket
  or in similar conventional syntax
  `let x := N; in M` Jam
- This alternate notation is literally an abbreviation for the explicit `lambda` form, but it is far more readable and intuitive
- The notation obviously generalizes to multiple `lambda` variables
- For this alternate notation, the beta-reduction rule has the form
  `(let [(x V)] M) ⇒ M[x := V]` Call-by-value (common)
  `(let [(x N)] M) ⇒ M[x := N]` Call-by-name (only Haskell)

Supporting Recursion

- Our LC dialect does not directly support recursion. A lambda abstraction cannot refer to itself. Later in the course, we will show that lambda-notation is so powerful that we can express recursion implicitly using a subtle closed lambda-abstraction that called the Y operator. Alonzo Church and his students discovered this operation in the 1930's when they created the lambda-calculus. Some of the unit tests that we provide for Assignment 2 (which only includes **let** as a syntactic abbreviation for an applications of a lambda-abstraction [**map**]) include a definition of Y.
- Later in the course, we will introduce a recursive form of **let**, which is used in the hand evaluation exercises appearing in the course calendar. For clarity the **let** operations in these exercises should be written as **letrec**.
- A recursive **let** construction

```
letrec x1 := E1;
       ...
       xn := En;
in E
```

is syntactically evaluated by reducing the right-hand-sides E_1, .., E_n to values V_1, .., V_n which can be referenced in **E** using the corresponding left-hand-side identifiers x_1, ..., x_n. If an identifier x_i appears in **E** , it evaluates to the value of V_i (assuming it is not shadowed by an inner **let** also binding x_i). The key difference between **let** and **letrec** is the *scope* of the local variables x_1, ..., x_n introduced in the construct. The scope of variables introduced in a **let** is simply the body **E** of the **let**. In a **letrec**, the scope is the entire **letrec**.