Program Semantics and Lexical Scope

Corky Cartwright
Department of Computer Science
Rice University

Programs vs. Expressions

- Reduction of expressions to values is the core of an algebraic formulation of computation.
- Comprehensive semantics for programs goes beyond evaluation of expressions.
- From an abstract perspective, an idealized program consists of (*i*) a collection of function definitions (which involve computation to create new values) and (*ii*) an expression constructed from those definitions to solve a computational problem.
- The semantics of Racket (or any functional language) is not simply the evaluation of expressions. It must also encompass collections of declarative function definitions.

What is the Semantics of a Program?

- A program is a collection of declarative function definitions plus an expression constructed using those function definitions that solves a given problem.
- From this perspective, a Racket program has the form:

```
(define f_1 (lambda (v_{1,1} … v_{1,n}) <body-of-f_1>))
 . . .
(define f_n (lambda (v_{m,1} … v_{m,n}) <body-of-f_n>))
<expr constructed from f_1, … f_n + prim ops>
```

What is the Semantics of a Program?

- Extend the reduction model to perform left-most evaluations on full programs.

```
(define f1 E1)
 . . .
(define fn En)
E              ;; E is constructed from f1, … fn + prim ops
```

- We reduce $\texttt{E}_1, \texttt{...}, \texttt{E}_n$ to values $\texttt{V}_1, \texttt{...}, \texttt{V}_n$ in leftmost order and then reduce `E`. In a typical program, most of the right-hand sides $\texttt{E}_1, \texttt{...}, \texttt{E}_n$ are already values. In all of the programs we have studied so far, all of the right-hand sides have been values. When evaluating $\texttt{E}_i$, all of the values of the preceding declared top-level variables (typically functions) $\texttt{f}_j$ are available. When evaluating `E`, the values of ***all*** of the variables $\texttt{f}_j$ are available. If any of these sub-computations diverge or abort with errors, the entire computation diverges or aborts with the error.

Examples

```
     (define double (lambda (n) (+ n n)))
     (double 5)
=>   (define double ... )
     ((lambda (n) (+ n n)) 5)
=>   ...
     (+ 5 5)
=>   ...
     10
```

Examples cont.

```
    (define fact (lambda (n) (if (zero? n) 1 (* n (fact (sub1 n))))))
    (fact 1)
=>  (define fact ... )
    ((lambda (n) (if (zero? n) 1 (* n (fact (sub1 n))))) 1)
=>  (define fact ... )
    (if (zero? 1) 1 (* 1 (fact (sub1 1))))
=>  (define fact ... )
    (if false 1 (* 1 (fact (sub1 1))))
=>  (define fact ... )
    (* 1 (fact (sub1 1)))
=>  (define fact ... )
    (* 1 (fact 0))
=>  (define fact ... )
    (* 1 ((lambda (n) (if (zero? n) 1 (* n (fact (sub1 n))))) 0))
=>  (define fact ... )
    (* 1 (if (zero? 0) 1 (* 0 (fact (sub1 0)))))
=>  (define fact ... )
    (* 1 (if true 1 (* 0 (fact (sub1 0)))))
=>  (define fact ... )
    (* 1 1)
=>  (define fact ... )
    1
```

Nested scope

- Algol 60 introduced the concept of nested scope to the world of programming languages (assuming we ignore the work of Church and his students involving the lambda-calculus).
- The idea (obvious in retrospect?) is much older. It was central to the lambda-calculus in the 1930's. Quantifications in first-order logic also have nested scopes.
    - The syntax of the "pure" lambda-calculus was essentially Core Racket without define and all primitive operations and constants, leaving only variables, applications, and lambda-abstractions.
    - The "pure lambda calculus" encoded numbers and booleans as functions (ugh!) which technically reduced it to a huge syntactic hack until Dana Scott salvaged it in 1970 by developing topological models (originally complete lattices and subsequently complete partial orders), a perspective now called *domain theory*. The key idea underlying these models that computable values are limits of progressively better approximations. (You can rigorously define infinite lists this way.)
    - Gordon Plotkin (who extended and refined Scott's models) designed what is now the canonical "impure" (but semantically elegant) extension of the pure lambda calculus called PCF by adding the following constants to the pure calculus: natural numbers, a ternary function **`if-zero?`**, **`add1`**, and **`sub1`**. Plotkin's original version of PCF included slightly more machinery including static types but it is not essential. In fact, our minimal untyped version is more elegant for reasons that I can explain if you take Comp 411.

Lambda Notation

Consider the identity function. Using the Racket subset we have covered so far in the course, we can easily define the function **`id`**:

```
(define (id x) x)
```

but why did we choose the name **`id`**? Mathematical functions do not have names embedded in them! They are simply sets of ordered pairs that meet certain constraints, right? What if we need to dynamically construct a new function in the middle of a computation? What is its name? What if this dynamic construction occurs inside a loop or a recursively defined function?

We need notation for expressing functions without naming them! How can we describe the identity function without naming it? How about

```
x → x
```

There are various tales about how this notation mutated to the corresponding lambda-notation:

```
λx . x
```

which is written in Lisp notation (restricted to the old BCD character set) as

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

which endures in the Lisp family of languages (Common Lisp, Scheme, Racket, …)

Lambda Notation cont.

Why are there parentheses around the abstraction variable in the Lisp version of lambda notation? To support expressing multi-ary functions like

```
(lambda (x y) (sqrt (+ (* x x) (* y y))))
```

But are multi-ary functions really necessary? No!

Consider the Racket function

```
(lambda (x) (lambda (y) (sqrt (+ (* x x) (* y y)))))
```

It is the “curried” equivalent of the first definition.

The term “curry” descends from the mathematician Haskell Curry who popularized the idea of encoding all multi-ary functions as unary functions returning functions. Did Curry originate the idea? No! (But he probably re-invented it.) Moses Schönfinkel published the idea in 1920 in a paper where he lays out the notion of a universal language framework for expressing computation. Did the “the science” pay any attention to the content of the paper? Of course not.

The lambda calculus (as studied by mathematicians) typically only includes unary lambda-abstraction since multi-ary lambda abstraction can be viewed as “syntactic sugar”.

Lambda Notation Introduces Nested Scope

Since lambda-abstractions are Core Racket expressions, a domain that has a simple inductive definition, lambda-abstractions can be nested (as they are in the curried expansions of multi-ary **`lambda`**-abstractions).

Example:

```
;; compose: (any -> any) (any -> any) -> (any -> any)
;; given unary functions f and g, (compose f g) returns their
;;   composition
(define compose (lambda (f g) (lambda (x) (f (g x)))))
```

What if the inner **`lambda`** introduced a variable **`f`** instead of **`x`**? What if we try to mention **`x`** outside the **`lambda`** that introduces it? We need to identify the *scope* of the *binding occurrence* of a variable (the variable **`v`** in the “header” **`(lambda (v) ...)`** of a lambda-abstraction.

Binding Occurrences vs. (Usage) Occurrences

Given a `lambda`-abstraction

`(lambda (... x ...) M)`

where ***`M`*** is any (legal) Racket expression, the occurrence of **`x`** in the "formal parameter list" **`(... x ...)`** is called a *binding occurrence* of **`x`**. Any use of **`x`** inside the expression M (unless it occurs inside a nested binding occurrence of **`x`**) is a (*usage*) *occurrence* of **`x`**.

Example:

```
;; compose: (any -> any) (any -> any) -> (any -> any)
;; given unary functions f and g, (compose f g) returns their
;;   composition
(define compose (lambda (f g) (lambda (x) (f (g x)))))
```

Nested **`lambda`**-abstractions are particularly important because they introduce new variables. The *scope* of a variable introduced in a **`lambda`**-abstraction is the body of the lambda-abstraction.

How Do We Nest Programs?

Ordinary Racket and Scheme do not literally support it. There is no expression nested within a program that has the form of a program. Recall that a program is not an expression. A program is a (possibly empty) sequence of definitions followed by an expression. Ordinary Racket and Scheme do support local scope because they support nested **`lambda`**-abstractions. But **`lambda`** bindings do *not* syntactically look *exactly* like bindings created by **`define`**. Semantically, they are the same, but they look syntactically different. The bindings created by applying a **`lambda`** abstraction to argument values are very hard to read if the body of the **`lambda`**-abstraction is non-trivial. For this reason, both Ordinary Racket and Scheme support an easier-to-read syntactic construct called **`let`** (and variants **`let*`**, and **`letrec`**, which we will introduce later even though they are superfluous in HTDP. Racket including the nesting construct **`local`** (which is only in HTDP Racket) supports exactly the same form of binding.

The `local` Construct for Program Nesting

- BNF Syntax (cryptic inductive definition) for **local**
  - *exp* ::= ...| (**local** (*def*$_1$ *def*$_2$ … *def*$_n$) *exp*)
  - *def* ::= (**define** *var* *exp*) | (**define** (*var*$_1$ *var*$_2$ … *var*$_n$) exp)

  In many contexts, the names of syntactic categories are enclosed in pointy brackets rather than italicized, *e.g*. **`<var>`** instead of ***var***

- Simple examples

  -
    ```
    (local [(define x 3)
            (define y 5)
            (define double (lambda x) (+ x x)))]
      (double (- y x)))
    ```

  -
    ```
    (local [(define disc (- (* b b) (* 4 a c)))]
      (sqrt disc))
    ```

Definition

- What’s wrong with following expressions?
  - `(local [(define x 1)])`
  -
    ```
    (local [(define x 1)
            (define x 2)]
      x)
    ```
  -
    ```
    (local [(define x 1)
            (define f (+ x 1))]
      (f x))
    ```

Why local?

Reason 1: Avoid namespace pollution

```
;; sort: list-of-numbers -> list-of-numbers
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon)
                          (sort (rest alon)))]))

;; insert: number list-of-numbers (sorted) -> list-of numbers
(define (insert an alon)
  (cond
    [(empty? alon) (list an)]
    [(cons? alon) (if (< an (first alon))
                      (cons an alon)
                      (cons (first alon) (insert an (rest alon))))]))
```

Why local?

- Namespace pollution cont.

```
;; insert-sort: list-of-numbers -> list-of-numbers
(define (insert-sort alon)
   (local
      ;; insert: number list-of-numbers (sorted) -> list-of numbers
      ((define (insert an alon)
        (cond
          [(empty? alon) (list an)]
          [else (if (< an (first alon))
                    (cons an alon)]
                     (cons (first alon) (insert an (rest alon)))])))
      (cond
          [(empty? alon) empty]
          [(cons? alon) (insert (first alon) (insert-sort (rest alon)))]))
```

Naïve implementation adds overhead. In principle, it can be eliminated by optimization.

Why local?

- Namespace pollution cont.

```
(define (main_fun x) exp)
(define (aux_fun1 …) exp1)
(define (aux_fun2 …) exp2)
```

```
(define (main_fun x)
  (local ((define (aux_fun1 …) exp1)
          (define (aux_fun2 …) exp2))
    exp))
```

Why local?

Reason 2: Avoid repeated computation

repeated work

```
(define (power los)
  (cond [(empty? los) (list empty)]
        [(cons? los)
         (append (cons-all (first los) (power (rest los)))
                 (power (rest los)))]))
```

Why local?

- Reason 2: Avoid repeated computation

```
(define (power los)
  (cond [(empty? los) (list empty)]
        [(cons? los)
         (local ((define pow (power (rest los))
           (append (cons-all (first los) pow) pow)]))
```

Why local?

- **Reason 3: Naming complicated expressions**

```
;; mult10 : list-of-digits  ->  list-of-numbers
;; creates a list of numbers by multiplying each digit in alod
;; by (expt 10 p) where p is the number of following digits
;; This is bad code used only as an example.  Good code
;; requires refactoring techniques we haven't learned yet.

(define (mult10 alod)
   (cond
       [(empty? alod) empty]
       [else (cons (* (expt 10 (length (rest alod))) (first alod))
                        (mult10 (rest alod)))]))
```

Why local?

- Reason 3: Naming complicated expressions

```
;; mult10 : list-of-digits  ->  list-of-numbers
;; creates a list of numbers by multiplying each digit on alod
;; by (expt 10 p) where p is the number of digits that follow
(define (mult10 alod)
  (cond
    [(empty? alod) 0]
    [else (local
                [(define a-digit (first alod))
                 (define the-rest (rest alon))
                 (define p (length the-rest))]
                (cons (* (expt 10 p) a-digit) (mult10 the-rest))]))
```

Variables and Scope

- At a cursory level, the scoping rule for **`local`** is the same as it is for **`lambda`**: local bindings are visible within the text of the **`local`** expression.
- Example:
  - **(local [(define answer$_1$ 42)**
    (define (f$_2$ x$_3$) (+ 1 x$_4$))]
    (f$_5$ answer$_6$))
- Variable occurrences: 1-6
- Binding (or defining) occurrences: 1,2,3
- Use occurrences: 4,5,6
- Scopes: 1:? 2:? 3:? The details are subtle.
- General rules for **`local`**:
  - local variables are visible only within the **`local`** expression
  - Within the local expression, scoping behaves exactly like it does in top-level programs.
- The are several important variations in scoping rules for nested binding constructs captured by the Racket/Scheme constructs **`let`**, **`let*`**, **`letrec`**, which we will study later in the course. **`local`** is sufficient but more verbose both notationally and conceptually.

Variables and Scope

- Recall:

  (local ((define $\texttt{answer}_1$ 42)
  (define ($\texttt{f}_2$ $\texttt{x}_3$) (+ 1 $\texttt{x}_4$)))
  ($\texttt{f}_5$ $\texttt{answer}_6$))

- Variable occurrences: 1-6
  - Binding (or defining) occurrences: 1,2,3
  - Use occurrences: 4,5,6
- Scopes:
  - 1: all of local expression
  - 2: all of local expression
  - 3: body of function definition: **`(+1 x)`**

Variables and Scope

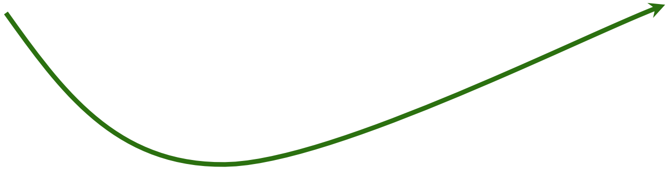
- In the following code segment, what will **g** evaluate to?

```
(define x 0)
(define f x)
(define g (local ((define x 1)) f))
```

Variables and Scope

- What will g evaluate to?

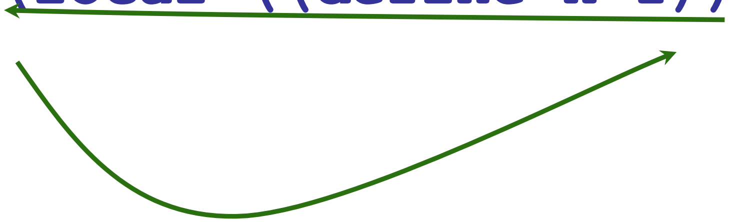
  -
    ```
    (define x 0)
    (define f x)
    (define g (local [(define x 1)] f))
    ```



Variables and Scope

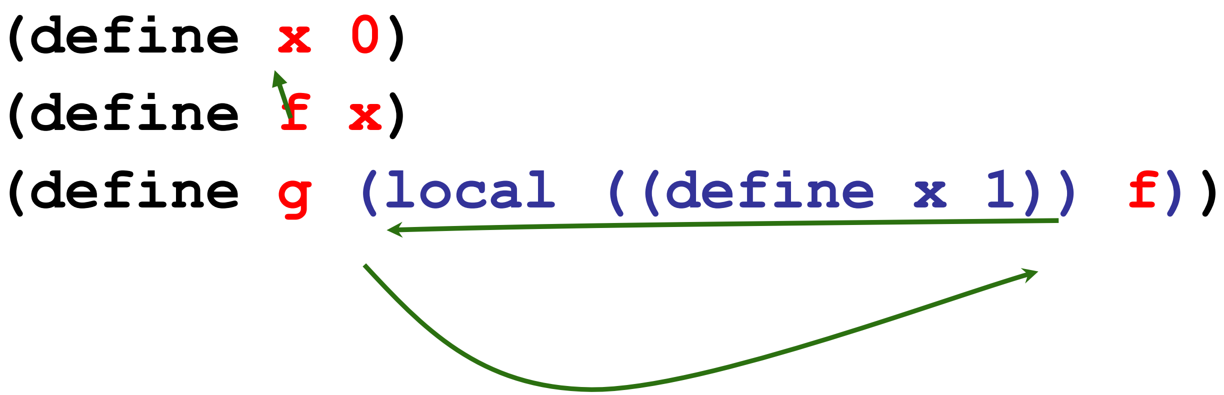
- What will g evaluate to?

```
(define x 0)
(define f x)
(define g (local ((define x 1)) f))
```



Variables and Scope

- What will “g” evaluate to?



```
(define x 0)
(define f x)
(define g (local ((define x 1)) f))
```

Renaming

- Recall:
  - **(local ((define answer$_1$ 42)**
    (define (f$_2$ x$_3$) (+ 1 x$_4$)))
    (f$_5$ answer$_6$))
- Which variables can be renamed?
- Use the same name for “binding occurrence” and “use occurrence”

```
(local [(define answer 42)
        (define (f x) (+ 1 x))]
  (f answer))
```

- What name choices can be used? Any name that does not clash with variable names already visible in same scope. A “fresh” variable name.

Renaming

- Recall:
  - **(local [(define answer$_1$ 42)**
    (define (f$_2$ x$_3$) (+ 1 x$_4$))]
    (f$_5$ answer$_6$))
- Which variables can be renamed?
- Use the same name for “binding occurrence” and corresponding “use occurrences”

```
(local [(define answer 42)
        (define (f' x) (+ 1 x))]
  (f' answer))
```

Evaluation Laws

- How do we (hand) evaluate Racket programs with **`local`**?
- By lifting local definitions to the top level and renaming all of the variables that they introduce (for which they create binding occurrences) with *fresh* names to avoid any collisions with variables already defined at the top level.
- To express these laws we need a new format for expressing rules. Why? Because promoting **`local`** constructs revises the set of definitions that constitute the *environment* in which evaluation takes place.
- New format for programs: we evaluate a sequence of **`define`** forms followed by an expression (which we formerly called the program application) which yields the answer for the computation.

Evaluation Laws

- To be continued …