

Program Semantics and Lexical Scope



Corky Cartwright
Department of Computer Science
Rice University



Lambda Abstractions

- In nearly all functional languages including Racket, functions are values just like integers and lists.
- Up to this point in the class, we have defined functions using the special syntax
`(define (<fn-name> <v1> ... <vn>) <body>)`
instead of
`(define <fn-name> <fn-value>)`
which is unnecessary if the language supports notation for function values.
- John McCarthy, the creator of Lisp, the forerunner of Scheme and Racket, earned his PhD in mathematics at Princeton where Alonzo Church was the leader of the logic group in the Department. Church invented the lambda-calculus in the 1930's, a universal programming language based on computing with functions.



A Glimpse of Lambda Notation

- Church used the Greek letter λ (lambda) as the symbol for introducing a function defining rule of the form
 $\langle \text{var-name} \rangle \rightarrow \langle \text{body} \rangle$
where $\langle \text{body} \rangle$ is an expression typically using $\langle \text{var-name} \rangle$
Church used the notation
 $\lambda \langle \text{var-name} \rangle . \langle \text{body} \rangle$
for such a rule defining a function mapping $\langle \text{var-name} \rangle$ to $\langle \text{body} \rangle$.
- Some examples include
 $\lambda x . x$ (the identity function)
 $\lambda f \rightarrow f 0$ (a function(al) that maps a function to its value at 0)
- In Lisp (including Scheme and Racket), lambda notation must be represented as lists. In addition, Lisp accommodates functions of arbitrary arity while the original pure λ -calculus forced all functions to be unary. So Racket supports the notation
 $(\text{lambda} (\langle v_1 \rangle \dots \langle v_n \rangle) \langle \text{body} \rangle)$
for functions.



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 (using lambda notation):

```
(define f1 (lambda (v1,1 ... v1,n) <body-of-f1>)
```

```
...
```

```
(define fn (lambda (vm,1 ... vm,n) <body-of-fn>)
```

```
<expr constructed from f1, ... fn + prim ops>
```

What is the Semantics of a Program?

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

(define f_1 E_1)

. . .

(define f_n E_n)

E ; ; constructed from f_1, \dots, f_n + prim ops

- We reduce E_1, \dots, E_n to values V_1, \dots, V_n in leftmost order and then reduce E . In a typical program, most of the right-hand sides E_1, \dots, 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 E_i , all of the values of the preceding declared functions f_j are available. When evaluating E , the values of all of the functions f_j are available. If any of these sub-computations diverge or abort with errors, the entire computation diverges or aborts with the error.
- This process will be described in full detail in the next lecture.



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 ((lambda (n) (if (zero? n) 1 (* n (fact (sub1 n)))))) (sub1 1))=> (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

- The programming language Algol 60 introduced the concept of nested scope to the world of programming languages
- 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 formulated it to a syntactic hack until Dana Scott salvaged it in 1970 by developing topological models (originally complete lattices and subsequently complete partial orders), now called *domain theory*. The key idea underlying these models is that computable values are limits of progressively better approximations.
 - Gordon Plotkin (who extended and refined Scott's work) 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`. The inclusion of `if-zero?` is critical it supports selective evaluation (often called “control”) of program text, a feature not seen in ordinary arithmetic and algebra.



Lambda Notation Introduces Nested Scope

Since lambda-abstractions are a form of Core Racket expression, a domain that has a simple inductive definition, lambda-abstractions can be nested!

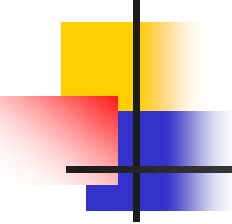
Example:

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

which expands into:

```
(define compose (lambda (f g) (lambda (x) (f (g x)))))
```

What if the inner `lambda` introduced a variable named `f`? What if we try to mention `b` outside this `lambda`. We need to identify the scope of the binding occurrence of a variable.



Nesting Is the Consequence of **lambda** Values + Inductively Defined Program Expressions

Since lambda-abstractions are a form of Core Racket expression, a domain with a simple inductive definition, lambda-abstractions can be nested! By definition, an expression can occur within an inductively constructed expression,

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 typically 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 don't technically support literal program nesting. There is no expression with 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 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 the application of 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**, which we will introduce later even though it is technically superfluous because it expands into the application of a lambda-abstraction. The **local** construct was created by the authors of HTDP so Racket programs could be nested within Racket programs. Why? So the semantics of nested code is identical (except for the avoidance of naming conflicts) to top-level code. In practice, it is less convenient than the corresponding constructs (**let**, **let***, **letrec**) in ordinary Scheme and Racket.



The **local** Construct for Program Nesting

- BNF Syntax (cryptic inductive definition) for **local**
 - $exp ::= \dots \mid (\mathbf{local} (def_1 def_2 \dots def_n) \mathbf{exp})$
 - $def ::= (\mathbf{define} \mathit{var} \mathit{exp}) \mid (\mathbf{define} (var_1 var_2 \dots var_n) \mathit{exp})$

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))))]))))
```

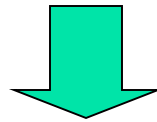
```
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon) (insert-sort (rest alon)))]))
```

Naïve implementation of **local** adds a little overhead. In principle, it can be eliminated by optimization in the compiler.

Why local?

- Namespace pollution cont.

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



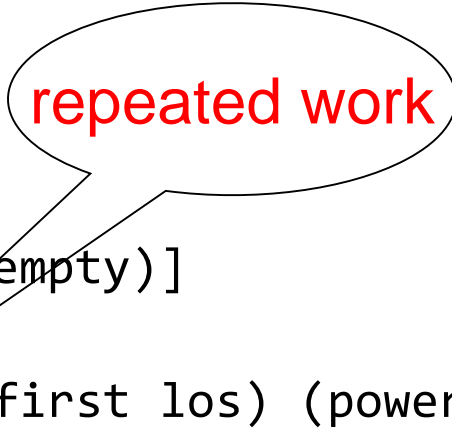
```
(define (main_fun x)
  (local ((define (aux_fun1 ...) exp1)
          (define (aux_fun2 ...) exp2)))
    exp))
```



Why local?

Reason 2: Avoid repeated computation

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



repeated work



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

- The scoping rule for **local** is essentially the same as it is for **lambda**: local bindings are visible within the text of the **local** expression.
- Example:
 - ```
(local [(define answer1 42)
 (define (f2 x3) (+ 1 x4))]
 (f5 answer6))
```
- 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.
- There 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 wordy.



# Variables and Scope

---

- Recall:
  - `(local ((define answer1 42)  
          (define (f2 x3) (+ 1 x4)))  
          (f5 answer6))`
- 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))
```



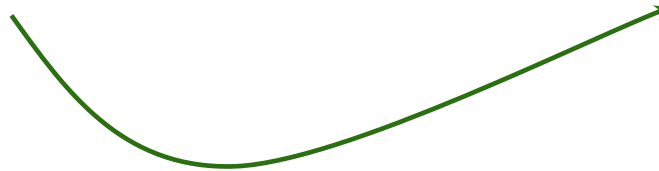
# Variables and Scope

- What will `g` evaluate to?

- `(define x 0)`

- `(define f x)`

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



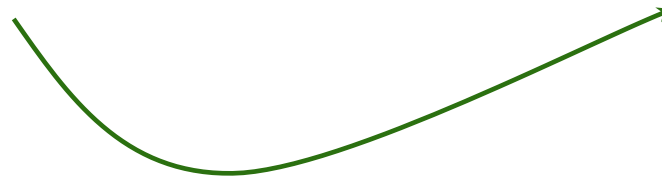
# Variables and Scope

- 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 answer1 42)
        (define (f2 x3) (+ 1 x4)))
      (f5 answer6))
```

- 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 answer1 42)
 (define (f2 x3) (+ 1 x4))]
 (f5 answer6))
```
- Which variables can be renamed?
- Use the same new name for “binding occurrence” and “use occurrences”
  - ```
(local [(define answer' 42)
        (define (f x) (+ 1 x))]
      (f answer'))
```



Renaming

- Recall:
 - `(local [(define answer1 42)`
 `(define (f2 x3) (+ 1 x4))]`
 `(f5 answer6))`
- 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))`



Renaming

- Recall:
 - `(local [(define answer1 42)`
`(define (f2 x3) (+ 1 x4))]`
`(f5 answer6))`
- Which variables can be renamed?
- Use the same name for “binding occurrence” and “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: 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 ...