Local definitions and lexical scope

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University
Top-Level Definitions

We have learned three kinds of definitions thus far:

1. Function definitions, e.g.,
   \[
   (\text{define } (f \ x) \ (+ \ x \ 1))
   \]

2. Variable (constant) definitions, e.g.,
   \[
   (\text{define two } (f \ 1))
   \]

3. Structure definitions, e.g.,
   \[
   (\text{define-struct pair } (\text{left} \ \text{right}))
   \]

They appear in Dr. Scheme’s Definitions window and are called *top-level definitions*. 
Local Expressions

A **local expression** groups together a set of definitions for use in a subcomputation:

$$(\text{local } (\text{def}_1 \text{ def}_2 \ldots \text{ def}_n) \text{ exp})$$

- **exp** is an arbitrary expression
- **def$_i$** is a definition in the set
- the variables defined in $\text{def}_1 \text{ def}_2 \ldots \text{ def}_n$ are distinct and only exist (are available for use) within the **local** expression i.e., within $\text{def}_1 \text{ def}_2 \ldots \text{ def}_n$ and **exp**
(define x 3) ;; top-level definition
(local [(define x 3)] (+ x 1)) ;; local expression
(define (f x) (+ x 1)) ;; top-level definition

(local [(define x 2)] ;; local definitions
  (define (f x) (+ x 1))]
  (f x)) ;; body

(+ (local [(define x 3)] ;; embedded local-expression
  (define (f x) (+ x 1))]
  (f x))
1)
Some Incorrect Examples

- 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-number
;; (sort lon) returns the elements of lon in ascending order
(define (sort alon)
  (cond
   [(empty? alon) empty]
   [(cons? alon) (insert (first alon) (sort (rest alon)))]))

;; insert: number list-of-numbers (sorted) -> list-of-number
;; (insert n lon) assumes lon is in ascending order and returns a
;; a list containing n and the elements of lon in ascending order
(define (insert an alon)
  (cond
   [(empty? alon) (list an)]
   [else (if (<= an (first alon))
     (cons an alon)
     (cons (first alon) (insert an (rest alon))))])))
Why local?

- Reason 1: Avoid namespace pollution (cont.)

;; sort: list-of-numbers -> list-of-numbers
(define (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) (sort (rest alon)))]))
Why **local**?

Reason 1: Avoid namespace pollution

```
(define (mainFun x) exp)
(define (auxFun1 ...) exp1)
(define (auxFun2 ...) exp2)
```

```
(define (mainFun x)
  (local [(define (minFun x) exp)
           (define (auxFun1 ...) exp1)
           (define (auxFun2 ...) exp2)]
           (mainFun x)))
```
Why local?

Reason 2: Avoid repeated computation

;;; max-num: list-of-number -> number
;;; (max=num lon) returns the largest number n in lon;
;;; throws an error if lon is empty
(define (max-num x lon)
  (cond
   [(empty? Lop) …]
   [else ... (first lon)
     ... (max-num x (rest lon)) ...])))
Why local?

Reason 2: Avoid repeated computation

\[
\text{(define (max-num lon)}
\text{  (cond)
  [(empty? Lon)
    (error "max-num applied to empty list")]
  [else
    (if (or (empty? (rest lon))
      (>= (first lon) (max-num (rest lon))))
      (first lon)
      (max-num (rest lon)))]
\]
Reason 2: Avoid repeated computation

(define (max-num lon)
  (cond
    [(empty? Lon)
      (error "max-num applied to empty list")]
    [else
      (if (empty? (rest lon))
        (first lon)
        (local [(define rest-max (max-num (rest lon)))]
          (if (> (first lon) rest-max)
            (first lon)
            rest-max))))])
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 digits that follow
;; 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

```scheme
;; 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 digits that follow
;; 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 (local [(define a-digit (first alod))
                 (define the-rest (rest alod))
                 (define p (length the-rest))]
             (cons (* (expt 10 p) a-digit) (mult10 the-rest)))]))
```
Recap of Variable Scopes from COMP 140

```python
myGlobal = 42

def myFunc(input):
    print "myFunc: input = ", input
    print "myFunc: myGlobal = ", myGlobal  # global variable visible here
    # neither local1 nor local2 are accessible here.
    if input > 0:
        local1 = 100
        # cannot access local2 from here.
        print "myFunc-if: local1 = ", local1
        print "myFunc- myGlobal
        print "myFunc-
    else:
        local2 = -100
        # cannot access
        print "myFunc-else:
        print "myFunc-
        print "myFunc-

print "myGlobal = ", myGlobal
myFunc(5)
myFunc(-5)
```

Source: http://www.clear.rice.edu/comp140/labs/
Variables and Scope in Scheme

• 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
  • Scope = code region where a definition may be used

• Scopes of definitions
  • 1:?  
  • 2:?  
  • 3:?
Variables and Scope

• What will \( g \) evaluate to?

\[
\begin{align*}
\text{(define x 0)} \\
\text{(define f x)} \\
\text{(define g}
\begin{align*}
\quad \text{(local ((define x 1)) f))}
\end{align*}
\end{align*}
\]
Renaming

Example:

\[
\text{(local}\ [\text{(define}\ \textit{answer}_1\ 42)\\\quad(\text{(define}\ (f_2\ \textit{x}_3)\ (+\ 1\ \textit{x}_4)))]\\\quad(f_5\ \textit{answer}_6))
\]

- Which variable occurrences can be renamed within the local expression?
- Use the same name for “binding occurrence” and all its “use occurrences”.
- Local variables can safely be renamed (no change to the answers produced by a program) without changing anything in the surrounding program.
- 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

Example:

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

=>

```
(local [(define answer_0 42)
         (define (f_0 x) (+ 1 x))]
     (f_0 answer_0))
```

We must rename all occurrences of a variable, both its binding occurrence and its use occurrences. In the preceding example, both `answer` and `f` have only one use occurrence. (Every variable has exactly one binding occurrence since each binding occurrence defines a new variable.) We are using the same underscore number convention for renaming as the DrScheme stepper.
Renaming

Recall our example:

\[
\begin{align*}
\text{(local } & \text{[(define answer 42)} \\
& \text{(define (f x) (+ 1 x))]} \\
& \text{(f answer))}
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\text{(local } & \text{[(define answer_0 42)} \\
& \text{(define (f_0 x) (+ 1 x))]} \\
& \text{(f_0 answer_0))}
\end{align*}
\]

We could also rename the function parameters within a `local` expression but it is not necessary for our purposes. We simply want to rename all of the variables (including function names) introduced in a `local`. 
Renaming in Evaluating `local`

Idea: We can promote (move) the block of `defines` introduced in a `local` to the top level (like the other `defines` in our program) provided that rename the variables introduced in the `local` so that they cannot clash with variables already `defined` at the top level.

Rule: when the leftmost unevaluated expression is a `local`, rename the variables `defined` in the `local`, lift the block of `defines` in the renamed `local` to the top level, and replace the `local` expression by its renamed body.
Evaluating \texttt{local} Expressions

Recap: how do we (hand) evaluate Scheme programs with \texttt{local}?

- By (i) renaming all of the \texttt{defined} variables in the \texttt{local} (with \texttt{fresh} names to avoid any collisions with variables already defined at the top level), (ii) lifting the renamed local definitions to the top level, and (iii) replacing the \texttt{local} expression by its renamed body.

To express this law we need a new format for expressing rules. Why? Because lifting \texttt{local} definitions \texttt{augments} the set of definitions that constitute the \texttt{environment} in which evaluation takes place.
Hand Evaluation Example

```
(define x 2)       ;; top-level definition
;; local-expression as part of another expression
(+ (local [(define x 3) (define (f x) (+ x 1))]
  (f x))
  1)
=>
(define x 2)
(define x_0 3)
(define (f_0 x) (+ x 1)) ;; parameters not renamed
(+ (f_0 x_0) 1)
=>
```
Hand Evaluation Example

\[(\text{define } x \ 2)\]
\[(\text{define } x_0 \ 3)\]
\[(\text{define } (f_0 x) (+ x 1))\]
\[(+ (f_0 3) 1)\]
\[=\]
\[(\text{define } x \ 2)\]
\[(\text{define } x_0 \ 3)\]
\[(\text{define } (f_0 x) (+ x 1))\]
\[(+ (+ 3 1) 1)\]
\[=\]
\[(\text{define } x \ 2)\]
\[(\text{define } x_0 \ 3)\]
\[(\text{define } (f_0 x) (+ x 1))\]
\[(+ 4 1)\]
Hand Evaluation Example

=>
  (define x 2)
  (define x_0 3)
  (define (f_0 x) (+ x 1))
  (+ 4 1)

With \texttt{local} in the language, each step in the evaluation must carry the environment (the block of \texttt{define}s constituting the program) as well as the expression being evaluated.

Confused? Try using the stepper (the menu button shaped like a foot) on examples in DrScheme.
When naming can cause problems

Romeo, Romeo! wherefore art thou Romeo?
.
.
.

What's in a name? That which we call a rose by any other name would smell as sweet.

Romeo and Juliet (II, ii)