211lab03_S11

Scope, `local`, and Abstract Functions

Instructions for students & labbies: Students use DrScheme, following the design recipe, on the exercises at their own pace, while labbies wander among the students, answering questions, bringing the more important ones to the lab's attention.

Quick Review of Scope

A function definition introduces a new local scope - the parameters are defined within the body.

- `(define (` *parameters* `...)` *body* `)`

A `local` expression introduces a new local scope--the names in the definitions are visible both within the bodies of the definitions and within the body. If a local name collides with (matches) a name in the surrounding program, the local name *shadows* the enclosing name; *i.e.*, the matching name in the enclosing program is invisible within the `local` expression; only the matching local name can be accessed. The same shadowing phenomenon happens when a parameter name in a function definition collides with a name defined in the surrounding program.

- `(local [` definitions `...]` body `)`

Note that the use of square brackets [ ] here is equivalent to parentheses( ). The brackets help set off the definitions a little more clearly for readability.

In order to use *local* and the other features about to be introduced in class, you need to set the DrScheme language level to **Intermediate Student**.

Exercises

Finding the maximum element of a list.

Let's consider the problem of finding the maximum number in a list which is used an example in Lecture 7.

- Develop the function `max-num` without using `local` exactly as in lecture.
- Develop the optimized version of this function (call it `opt-max-num`) using `local` exactly as in lecture.

Try running each version on the following input (or longer ones):

```
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
```

If it is convenient, use your `up` function from last lab to avoid typing long input lists like the preceding one.

What difference do you see?

For each version, determine approximately how many calls to your function is made for the input

`(list 1 2 3 ...` *n*`)`

for various values of *n*.

Which version is more efficient?

Note that you can write an efficient solution without using `local`; you can use an auxiliary function instead. The auxiliary function takes the expression appearing on the right hand side of the local definition as an argument. Quickly write this version of the function.

In general, you can take any program using `local`, and turn it into an equivalent program without `local`. Using `local` doesn't let us write programs which were impossible before, but it does let us write them more cleanly and concisely.

Generating lists of ascending numbers

Retrieve your code for the `up` function and its helper `upfrom` from last lab.

Rewrite this program so that the helper function `upfrom` is hidden inside a `local` expression and takes only one argument since the upper bound argument is available as the parameter to the enclosing definition of `up`.

Don't forget to revise the contract and purpose for the improved `upFrom` function.

Advice on when to use `local`

These examples point out the reasons why to use `local`:

1. Avoid code duplication. I.e., increase code reuse.
2. Avoid recomputation.
   This sometimes provides dramatic benefits. Avoid re-mentioning an unchanging argument.

3. To hide the name of helper functions. An **aside**: many programming languages (including Java) provide alternate mechanisms (such as `public`/`private` attributes) for hiding helper functions and data which scale better to large programs and accomodate the unit testing of hidden functions (which `local` does **not**!).
4. Name individual parts of a computation, for clarity.On the other hand, don't get carried away.

Here are two easy ways to misuse local:

```
;; max-num: non-empty-list-of-number -> number
(define (max-num alon)
  (local [(define max-rest (max-num (rest alon)))]
    (cond [(empty? (rest alon)) (first alon)]
          [else  (if (<= (first alon) max-rest)
                     max-rest
                     (first alon))])))
```

Try running this example on any legal input (according to the contract). It blows up!
Make sure you don't put a local too early in your code.

```
;; max-num: non-empty-list-of-number -> number
(define (max-num alon)
  (cond [(empty? (rest alon)) (first alon)]
        [else  (local [(define max-rest (max-num (rest alon)))
                       (define first-smaller? (<=(first alon) max-rest))]
                 (if first-smaller? max-rest (first alon)))]))
```

This isn't wrong, but the local definition of `first-smaller?` is unnecessary. Since the comparison is only used once, this refactoring is not a case of code reuse or recomputation. It provides a name to a part of the computation, but whether that improves clarity in this case is a judgement call.

Scope and the semantics of local

How are `local` expressions evaluated? The following reduction rule describes how to evaluate a `local` expression when it is the leftmost un-evaluated expression.

1. Rename each defined name in the `local` expression by appending `i` *to its name for a suitable choice of* `i`*. A good choice for* `i{_}` is simply the index (starting from 0) of this particular `local` evaluation in the course of the evaluation. The DrScheme stepper follows this convention. When renaming local variables,
make sure that you consistently change each defined name in the *entire* `local` expression.

2. Lift the modified `define` statements from the `local` expression to the top level (at the end of top level program), and replace the `local` expression by it body (which has been transformed by renaming).

This two-part process is technically a single step in an evaluation. But you can make each part of the rule a separate step if you choose. See Example 1 below.

Evaluating Program Text

Now that we have extended our hand-evaluation model to include the definitions forming a program, we can describe how to handle `define` statements that bind ordinary variables rather than functions. The right hand sides of `define` statements are not necessarily values. When evaluating a program, you must always reduce the leftmost expression that is not a value; in some cases this expression may be part of the right hand size of a `define`.

Sample evaluations involving `local` and explicit program text

Example 1

Observe how the rewriting makes it clear that there are two separate variables initially named `x`.

```
(define x 5)
(local [(define x 7)] x)
```

=>

```
(define x 5)
(define x_0 7)
x_0
```

```
=>
```

```
7
```

This evaluation can also be written in a slightly longer form where the evaluation of a `local` expression takes two reduction steps rather than one. (The one-step form is better technically, but the two-step form is acceptable in the context of an introductory course.)

```
(define x 5)
(local (define x 7) x)
```

```
=>
```

```
(define x 5)
(local (define x_0 7) x_0)
```

```
=>
```

```
(define x 5)
(define x_0 7)
x_0
```

```
=>
```

```
7
```

Example 2.

```
(define x 5)
(define y x)
(define z (+ x y))
(local [(define x 7) (define y x)] (+ x y z))
```

```
=>
```

```
(define x 5)
(define y 5)
(define z (+ x y))
(local [(define x 7) (define y x)] (+ x y z))
```

```
=>
```

```
(define x 5)
(define y 5)
(define z (+ 5 y))
(local [(define x 7) (define y x)] (+ x y z))
```

```
=>
```

```
(define x 5)
(define y 5)
(define z (+ 5 5))
(local [(define x 7) (define y x)] (+ x y z))
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(local [(define x 7) (define y x)] (+ x y z))
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(local [(define x_0 7) (define y_0 x_0)] (+ x_0 y_0 z))
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 x_0)
(+ x_0 y_0 z)
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 7)
(+ x_0 y_0 z)
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 7)
(+ 7 y_0 z)
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 7)
(+ 7 7 z)
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 7)
(+ 7 7 10)
```

=>

```
(define x 5)
(define y 5)
(define z 10)
(define x_0 7)
(define y_0 7)
24
```

Note: + is a function of an arbitrary number of numbers in Scheme. Hence `(+ 7 7 10)` reduces to `24` in a single step. The longer form of this evaluation adds an extra step in lifiting the `local`.

If any aspect of hand evaluation confuses you, try running examples in DrScheme using the stepper.

DrScheme provides an easy way to look at this: the Check Syntax button. Clicking on this does two things. First, it checks for syntactic correctness of your program in the definitions window. If there are errors, it reports the first one. But, if there aren't any, it then annotates your code with various colors and, when you move your cursor on top of a placeholder, draws arrows between placeholder definitions and uses.

Review

In Lectures 8-9, we developed the Scheme `filter` function, a functional that takes a unary predicate and a list as arguments. Starting from several examples using specific predicates, we noticed a great deal of similarity. By abstracting away that similarity into a function parameter, we obtained the following definition:

```
;; filter : (X -> boolean) (list-of X) -> (list-of X)
;; Purpose: (filer p alox) returns the elements e of alox for which (p e) is true

(define (filter keep? alox)
  (cond [(empty? alox) empty]
        [(cons? alox) (if (keep? (first alox))
                        (cons (first alox) (filter keep? (rest alox)))
                        (filter keep? (rest alox)))]))
```

We can putatively improve the preceding definition by eliminating the repetition of the code `(filter keep? (rest alox))`:

```
;; filter : (X -> boolean) (list-of X) -> (list-of X)
;; (filer p alox) returns the elements e of alox for which (p e) is true

(define (filter keep? alox)
  (cond [(empty? alox) empty]
        [(cons? alox)
         (local [(define kept-rest (filter keep? (rest alox)))]
           (if (keep? (first alox))
               (cons (first alox) kept-rest)
               kept-rest))]))
```

The refactoring of the first definition of `filter` given above is borderline. It does *not* improve the running time of the program since the duplicate calls appear in disjoint control paths. Which of the two preceding definitions is easier to understand? The answer is debatable; it is not clear that the "improved" definition is better.

Some functional programmers advocate performing yet another "optimization": introducing a recursive helper function in the body of `filter` because the `keep?` parameter is unnecessary in `filter` except as single a top-level binding. Since `keep?` never changes in any recursive call, it can be eliminated as

a parameter in a helper function nested inside `filter` where the binding of `keep?` is visible. We are not persuaded that this revision is a good idea for two reasons. First, eliminating constant parameters from function calls is a low-level transformation that is best performed by a compiler. In many cases, it may degrade rather than improve efficiency. Second, the program with the helper function is harder to understand than either program above.

Note the preceding definitions of `filter` are not acceptable to DrScheme because the name `filter` collides with the Scheme library function named `filter` (which performs the same operation). If you want to run either of the preceding programs in DrScheme, you will have to rename `filter` (as say `Filter`).

Optional Exercise: rewrite `filter` in the following "optimized" form:

```
(define (Filter keep? alox)
  (local [(define (filter-help alox) ...]
    (filter-help alox)))
```

Do not bother writing a contract, purpose statement, or template instantiation for this function since we recommend that you throw this code away after the lab is over. The name has been changed to `Filter` to avoid colliding with the Scheme library function named `filter`.

Exercises

- Load one of the more interesing Scheme programs you have written (such as HW2) into DrScheme. (If you did the preceding optional exercise, you can use this program.) Perform the "Check Syntax" command to identify where each variable is bound. Recall that when you put your cursor above a binding instance, it will draw arrows to the uses, and when you put your cursor above a use, it will draw an arrow to its binding instance.
- X Using the Scheme library function `filter`, develop a function that takes a list of numbers and returns a list of all the positive numbers in that list.

The `map` functional

We are familiar with writing functions such as the following:

```
;; double-nums : (list-of number) -> (list-of number)
;; (double-num (list e1 ... ek)) returns (list d1 ... dk) where each element di is twice ei.

(define (double-nums alon)
  (cond [(empty? alon) empty]
        [(cons? alon)  (cons (* 2 (first alon)) (double-nums (rest alon)))]))


;; <3-nums : (list-of number) -> (list-of boolean)
;; (<3-nums (list e1 ... en)) returns (list b1 ... bn) where bi has the value (< ei 3)

(define (<3-nums alon)
  (cond [(empty? alon) empty]
        [(cons? alon)  (cons (< (first alon) 3) (<3-nums (rest alon))))]))
```

These functions are very similar and can be written trivially using the Scheme library function `map` discussed in Lecture 9.

Exercises.

- Write `double-nums` and `<3-nums` using `map`.

The Big Picture

Abstract functions are both powerful and convenient. By using abstract functions to group all the common similar elements of many functions, we can concentrate on what's different. This practice allows us to write shorter code that is also clearer and more understandable

We have written many functions to combine elements of a list, *e.g.*, to sum the numbers in a list and to find the maximum element in a list. In each, we have some value `base` as the result for the empty list and a function `f` to combine the elements of the list. The elements can be combined in two different ways: one associating to the right (`foldr`) and one associating to the left `foldl`. The following equations describe how these two form of combining work:

```
(foldr f base (list x1 x2 ... xn)) = (f x1 (f x2 ... (f xn base)))        [1]
(foldl f base (list x1 x2 ... xn)) = (f xn ... (f x2 (f x1 base))...)     [2]
```

Many functions we've written fit this pattern, although this fact might not be obvious at first.

Exercises

1. X Based upon the preceding equation ([1]), what should the following evaluate to? Think about them first, then try them in DrScheme (where `foldl` is pre-defined).
   a. (foldr + 5 (list -1 5 -3 4 2))
   b. (foldl + 5 (list -1 5 -3 4 2))
   c. (foldr - 5 (list -1 5 -3 4 2))
   d. (foldl - 5 (list -1 5 -3 4 2))
   e. (foldr cons empty (list -1 5 -3 4 2))
   f. (foldl cons empty (list -1 5 -3 4 2))
   g. (foldr append empty (list (list 1 2) (list 4) empty (list 8 1 2)))
   h. (foldr append empty (list (list 1 2) (list 4) empty (list 8 1 2)))
2. X What is the contract for `foldr`? For `foldl`? You should be able to determine this from the equations and the examples above. (We also covered the typing of `foldr` in lecture.) **Hint:** First, determine the type of `foldr` and `foldl` assuming the input list is a list of numbers, and then generalize.
3. Using `foldr`, define a function to compute the product of a list of numbers. Do the same for `foldl`.
4. Using `foldr`, define `map`. (Also done in lecture.) Do the same for `foldl`.
5. Define the function `Foldr` to satisfy equation (1) above. As you might expect, it follows the template for a function consuming a list. (It was also done in lecture.) Test your function against Scheme's built-in `foldr` to make sure they give the same results for the same inputs.
6. Define the function `Foldl` to satisfy equation (2) above. As you might expect, it follows the template for a function consuming a list. Test your function against Scheme's built-in `foldl` to make sure they give the same results for the same inputs.
7. Define a function to compute whether all elements in a list of numbers are greater than 6. Write two version versions, one using `foldr` and one using `foldl`, choosing suitable names (distinct from `filter` for each.
   Then generalize both definitions to define `filter`. Are your two `filter` functions identical? Hint: look at computations that generate errors.
8. Define a function that, given a list of numbers, returns the sum of all the positive numbers in the list. Write two versions, one using `foldr` and one using `foldl`.
9. Without using explicit recursion, develop a function `upfrom` that, given `i` and `n` returns the list of length `i` of numbers up to `n`.

More examples

Here are a few more pre-defined abstract functions:

- `(andmap f (list x1 ... xn)) = (and (f x1) ... (f xn))`
- `(ormap f (list x1 ... xn)) = (or (f x1) ... (f xn))`
- `(build-list n f) = (list (f 0) ... (f (sub1 n)))`

Exercises

The following can be defined using some combination of the pre-defined abstract functions.

1. X Define a function that, given a list of numbers, determines whether all of the numbers in the list are even. Write two versions one using `foldr` and one using `foldl`.
2. Define `andmap` and `ormap` using `foldr` rather than recursion. Do the same using `foldl`.

Summary

The following table presents a quick summary of the abstract functions mentioned in this lab:

`filter`	selects some elements from a list
`map`	applies a function to each list element
`andmap`	applies a function to each list element and reduces this list using `and`
`ormap`	applies a function to each list element and reduces this list using `or`
`build-list`	given a unary function `f` and a natural `n`, constructs `(list (f 0) ... (f n))`
`foldr`	associating to the right, reduces a list using the specified binary function and base case
`foldl`	associating to the left, reduces a list using the specified binary function and base case