CS 135 Winter 2019

Tutorial 9: Local and Lambda
Goals of this tutorial

You should be able to...

- understand the syntax and semantics of local.
- step through local definitions.
- use local when writing your own functions.
- step through code involving lambda.
- write functions that consume and/or produce other functions.
- write functions using lambda.
- use filter and map.
Review: Local Definitions

Recall the special form `local` which allows us to create local definitions. The syntax for `local` is as follows:

\[
\text{(local \{definition}_1 \ldots \text{definition}_n\} \text{expression})
\]

where each `definition` is a `define` statement, and `expression` is a Racket expression that uses these definitions.
Clicker Question: Local Definitions

In Intermediate Student, what would the following expression produce?

```
(define (sum-lon alon)
  (local [(define (sum-lon-acc alon sum-so-far)
            (cond [(empty? alon) sum-so-far]
                  [else (sum-lon-acc (rest alon) (+ (first alon) sum-so-far))])])
    (sum-lon-acc alon 0)))
```

```
(sum-lon-acc (list 1 9 1 23) 0)
```

A  1  
B  10 
C  11 
D  34 
E  An error
Stepping Problem - Local

Provide a step-by-step evaluation of the following program. When renaming local definitions, append “.0” if possible, or else “.1”, “.2”, etc. Do not recopy any line that is already in its simplest form.

\[
\text{(define (f x y)}
\]

\[
\text{  (local)}
\]

\[
\text{    [(define a (+ x 3))}
\]

\[
\text{    (define y 4)}
\]

\[
\text{    (define (g x)}
\]

\[
\text{      (+ x a)]}
\]

\[
\text{    (∗ 2 (g y)))]
\]

\[
\text{(f 7 3)}
\]

\[
\text{(f 7 3)}
\]
(define (f x y)
  (local
    [(define a (+ x 3))
     (define y 4)
     (define (g x) (+ x a))]
    (* 2 (g y)))))
(f 7 3)
\[(\text{local})\]
\[
[(\text{define } a (\text{+ } 7 3))]
\]
\[
(\text{define } y 4)
\]
\[
(\text{define } (g \ x))
\]
\[
(\text{+ } x a))]
\]
\[
(\text{* } 2 (g \ y)))\]
\[
(\text{define } a_0 (\text{+ } 7 3))
\]
\[
(\text{define } y_0 4)
\]
\[
(\text{define } (g_0 \ x))
\]
\[
(\text{+ } x a_0))
\]
\[
(\text{* } 2 (g_0 y_0))\]
(define a_0 (+ 7 3))
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
(* 2 (g_0 y_0))

(define a_0 10)
(* 2 (g_0 y_0))
(define a_0 10)
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
⇒ (∗ 2 (g_0 y_0))
⇒ (∗ 2 (g_0 4))
(define a_0 10)
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
⇒ (∗ 2 (g_0 y_0))
⇒ (∗ 2 (g_0 4))
⇒ (∗ 2 (+ 4 a_0))
(define a_0 10)
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
⇒ (∗ 2 (g_0 y_0))
⇒ (∗ 2 (g_0 4))
⇒ (∗ 2 (+ 4 a_0))
⇒ (∗ 2 (+ 4 10))
(define a_0 10)
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
⇒ (* 2 (g_0 y_0))
⇒ (* 2 (g_0 4))
⇒ (* 2 (+ 4 a_0))
⇒ (* 2 (+ 4 10))
⇒ (* 2 14)
(define a_0 10)
(define y_0 4)
(define (g_0 x)
  (+ x a_0))
⇒ (∗ 2 (g_0 y_0))
⇒ (∗ 2 (g_0 4))
⇒ (∗ 2 (+ 4 a_0))
⇒ (∗ 2 (+ 4 10))
⇒ (∗ 2 14)
⇒ 28
Clicker Question - Contracts with Function Types

What would be the contract for this function?

```
(define (f a b)
  (local [(define (f c) (+ (* a c) (* b c))])
    f))
```

A ;; f: Num Num → Num → Num
B ;; f: Num Num → Num
C ;; f: Num Num → (Num → Num)
D ;; f: Num Num → Function
E ;; f: Num Num → (fn Num → Num)
Here are two reasons why we might want to use local expressions:

1. **Encapsulation** - local allows us to hide parts of our program from each other, since anything defined inside a local expression is not visible from outside the local. For example, we can define a helper function inside the function it is helping, and no function defined outside of that main function will be able to use that helper.

2. **Efficiency** - We can use local to eliminate repeated computations by storing the result of a computation in a local variable, and using that variable whenever the value is needed. This prevents us from having to repeatedly recompute the value.
Group Discussion - Efficiency with Local

Given the code below, how could you use local constant definitions to make it more efficient?

```scheme
;; list-min: (listof Num) → Num
;; requires: lon is non-empty
(define (list-min lon)
  (cond
   [(empty? (rest lon)) (first lon)]
   [(<= (first lon) (list-min (rest lon))) (first lon)]
   [else (list-min (rest lon))]))
```
Review: Lambda

\[
\text{(local \[\text{(define (name-used-once x1 \ldots xn) exp)}\]}
\]
\text{name-used-once)}

can also be written

\[
\text{(lambda (x1 \ldots xn) exp)}
\]

\text{lambda} can be thought of as “make-function”.

It can be used to create a function which we can then use as a value – for example, as the value of the first argument to \text{filter}, \text{map}, or \text{foldr}.
Group Problem - Stepping with Lambda

Step through the following program:

```((lambda (x y z) (* x 5)) 9 (+ 1 2) 8)```
Group Problem - above-average

(define-struct student (name grade))

;; A Student is a (make-student Sym Num)

write a function (twice, once using lambda and once using local) above-average which consumes a Num representing an average mark in some course. above-average will produce a function that consumes a Student and determines if that student has a grade that is higher than the average mark. Include a contract.

;;Example
(define above-cs-mt-avg? (above-average 67.8))
(check-expect (above-cs-mt-avg? (make-student 'Turing 73)) true)
Abstract List Functions

Recall the abstract list functions \texttt{filter}, \texttt{map}, \texttt{foldr}, and \texttt{build-list}. 

\begin{itemize}
    \item \texttt{filter: (X \rightarrow \text{Bool}) (list of X) \rightarrow (list of X)}
    \item \texttt{map: (X \rightarrow Y) (list of X) \rightarrow (list of Y)}
    \item \texttt{foldr: (X Y \rightarrow Y) Y (list of X) \rightarrow Y}
    \item \texttt{build-list: \text{Nat} (\text{Nat} \rightarrow X) \rightarrow (list of X)}
\end{itemize}

Use abstract list functions to do the following tasks without explicit recursion:

Add the list of numbers in \texttt{(list 1 2 3 4 5)}: \Rightarrow 15

Create a list of odd numbers from 1 to 12:
\Rightarrow (\texttt{list 1 3 5 7 9 11})

Double each element in \texttt{(list 1 2 3 4 5)}: \Rightarrow (\texttt{list 2 4 6 8 10})

Keep all the elements in \texttt{(list 1 2 3 4 5 6 7)} that are divisible by 3:
\Rightarrow (\texttt{list 3 6})
Group Problem - invite-list

(define-struct student (name grade))

;; A Student is a (make-student Sym Num)

;; requires: 0 <= grade <= 100

The Engineering Faculty of Waterloo regularly holds a faculty dinner to which any student who holds a cumulative average of over 95% is invited. Write a function invite-list which consumes a list of Students and produces a list of Students who will be invited to the dinner. Include a contract for each function you write. You must use the built-in function filter. All helper functions must be encapsulated in a local block with purposes and contracts.
Group Problem - map-lofn

Write a function `map-lofn` which consumes an argument of type Any and a list of functions and produces a list of Any by applying each function from the list to the first argument.

`;;(map-lofn item lofn)` consumes an item of any type and a list of functions `lofn`, and produces a list containing the output of each function in `lofn` applied to `item`

`;; map-lofn: X (listof (X → Any)) → (listof Any)`