Local definitions

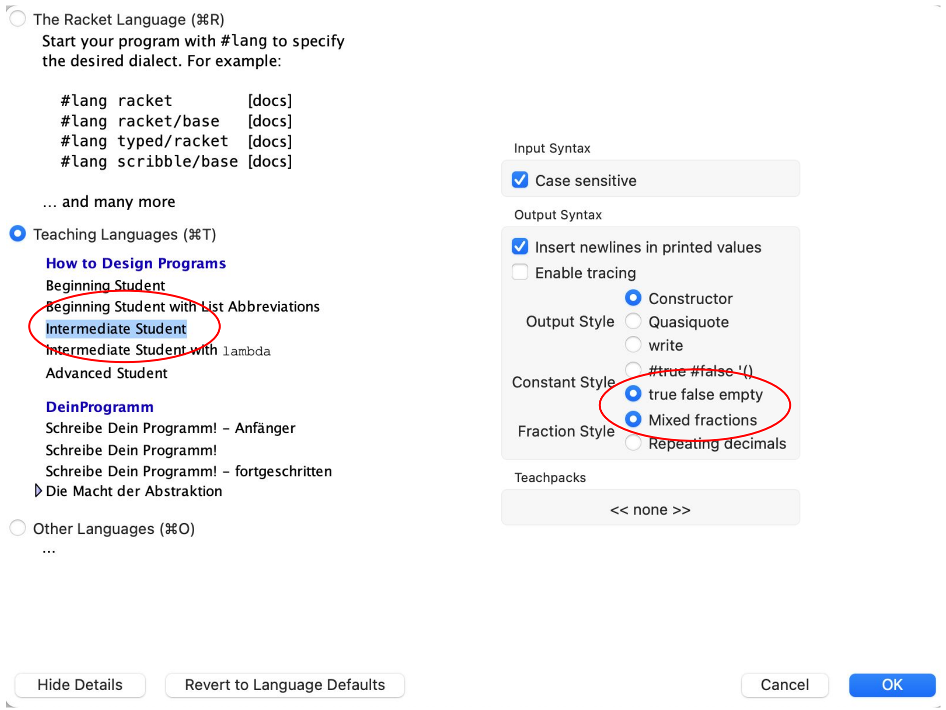
CS135 Lecture 16

Intermediate student

It’s finally time to change language level again, this time to “intermediate student”.

This change is necessary for you to use `local`, which is the topic of this lecture.

Don’t forget to set “Constant Style” and “Fraction Style” as shown on the right.



○ The Racket Language (⌘R)

Start your program with #lang to specify the desired dialect. For example:

```
#lang racket          [docs]
#lang racket/base     [docs]
#lang typed/racket    [docs]
#lang scribble/base   [docs]
```

… and many more

● Teaching Languages (⌘T)

How to Design Programs

- Beginning Student
- Beginning Student with List Abbreviations
- [✓ Intermediate Student]
- Intermediate Student with `lambda`
- Advanced Student

DeinProgramm

- Schreibe Dein Programm! – Anfänger
- Schreibe Dein Programm!
- Schreibe Dein Programm! – fortgeschritten
- ▷ Die Macht der Abstraktion

○ Other Languages (⌘O)

…

Input Syntax

- ☑ Case sensitive

Output Syntax

- ☑ Insert newlines in printed values
- ☐ Enable tracing

Output Style	[✓ Constructor] Quasiquote write
Constant Style	#true #false '() [✓ true false empty]
Fraction Style	[✓ Mixed fractions] Repeating decimals

Teachpacks

<< none >>

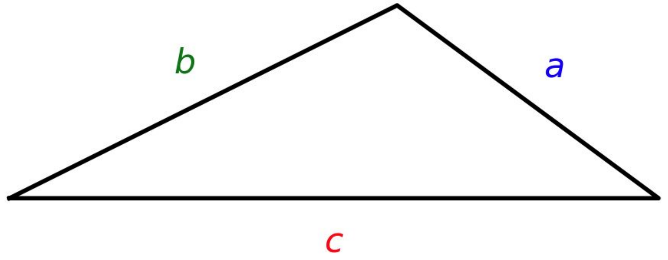
L16.0 Heron’s formula

The area of a triangle from the length of its sides

Heron's formula for the area of a triangle with sides *a*, *b*, *c* is:

$$\sqrt{s(s-a)(s-b)(s-c)}$$

Where the *semi-perimeter* $s = (a+b+c)/2$



It is not hard to create a Racket function to compute this function, but it is surprisingly difficult to do so in a clear and natural fashion.

The formula takes its name from the mathematician and engineer, Heron (or Hero) of Alexandria who proved it around the 1st or 2nd century CE.

Let’s try to directly translate the function to Racket

```
(define (heron-v0 a b c)
  (sqrt
   (* (/ (+ a b c) 2)
      (- (/ (+ a b c) 2) a)
      (- (/ (+ a b c) 2) b)
      (- (/ (+ a b c) 2) c))))
```

$$\sqrt{s(s-a)(s-b)(s-c)}$$

$$s = (a+b+c)/2$$

```
(check-expect (heron-v0 3 4 5) 6)
```

This translation is correct, but not really direct because the computation of *s* is repeated four times. It’s looks unnecessarily complex and confusing.

Maybe we can compute *s* in a helper function?

```
(define (s a b c)
  (/ (+ a b c) 2))

(define (heron-v1 a b c)
  (sqrt
   (* (s a b c)
      (- (s a b c) a)
      (- (s a b c) b)
      (- (s a b c) c))))

(check-expect (heron-v1 3 4 5) 6)
```

This solution gives an explicit definition for *s*.

But the helper function still need parameters, which again makes the relationship to Heron's formula hard to see. And there's still repeated code and repeated computations.

Heron’s formula with **`local`**

```
(define (heron a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))

(check-expect (heron 3 4 5) 6)
```

$$\sqrt{s(s-a)(s-b)(s-c)}$$

$$s = (a+b+c)/2$$

This version really is a direct translation.

The **`local`** gives a definition for **`s`** that has scope only within the **`(local ...)`**.

We have access to the parameters to compute a value for **`s`**.

The value of **`s`** is computed only once.

Format of `local`

```
(local
  [(define ...)
   (define ...)
   (define ...)
   ...
   (define ...)]
  (...))
```

$\longleftarrow$ These definitions

$\longleftarrow$ Have scope only in this expression

Format of `local`

```
(local
  [(define x 6)
   (define y 2)
   (define z 1)
   (define (f x) (+ x 1))
   (define (g x y) (+ x y))]
  (+ (f x) (g y z)))
```

$\leftarrow$ These definitions

$\leftarrow$ Have scope only in this expression

L16.1 α-conversion

Substitution rule for `local`

The substitution rule works by replacing every name defined in the `local` with a “fresh” identifier: a new, unique name that has not been used anywhere else in the program.

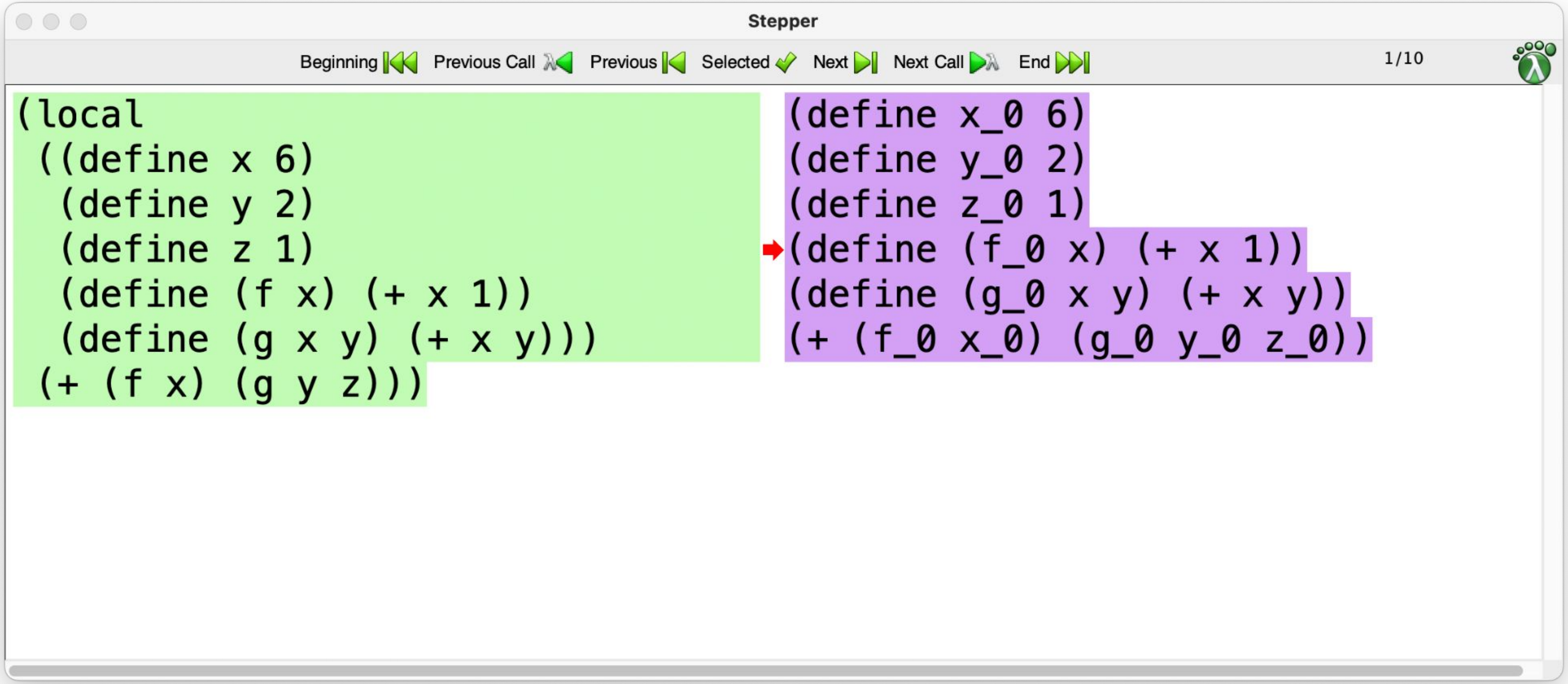
Each old name within the `local` is replaced by the corresponding new name.

Because the new name hasn't been used elsewhere in the program, the local definitions (with the new name) can now be “promoted” to the top level of the program without affecting anything outside of the `local`.

We can now use our existing substitution rules to evaluate the expression.

This process is called “α-conversion”.

α-conversion in the stepper



```
(local
 ((define x 6)
  (define y 2)
  (define z 1)
  (define (f x) (+ x 1))
  (define (g x y) (+ x y)))
 (+ (f x) (g y z)))
```

```
(define x_0 6)
(define y_0 2)
(define z_0 1)
(define (f_0 x) (+ x 1))
(define (g_0 x y) (+ x y))
(+ (f_0 x_0) (g_0 y_0 z_0))
```

Summary of α-conversion

1. Choose a fresh name for each definition in the local.
2. Substitute that fresh name everywhere it is used in the local.
3. Lift the definition to the top level.
4. Replace `(local [...]` *expression*`)` with just the *expression*.

The stepper adds an underscore and a number to each variable (`x_0`) but any approach to renaming will do, provided it is guaranteed to produce a fresh name.

A bit of mathematical context

The “α-conversion” operation, along with a second operation called “β-reduction”, are the two operations used in a mathematical formulation of computation called “lambda calculus” (or “λ-calculus”):

α-conversion corresponds to the renaming process just discussed

β-reduction corresponds to function application

Racket traces its roots back to the lambda calculus.

We will learn more about λ in forthcoming lectures.

L16.2 Benefits of **`local`**

Benefits of `local`

- **Clarity**: Naming subexpressions
- **Encapsulation**: Hiding stuff
- **Scope:** Reusing names
- **Efficiency**: Avoiding recomputation

```
(define (heron a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Clarity: Naming subexpressions with **`local`**

Sometimes we choose to use **`local`** in order to give subexpressions meaningful names to make the code more readable, even if they are not reused.

Shorter functions are usually clearer and simpler, but giving meaningful names to subexpressions may improve clarity, even if they make a function longer.

```
(define (distance p1 p2)
  (local [(define delta-x (- (get-x p1) (get-x p2)))
          (define delta-y (- (get-y p1) (get-y p2)))]
     (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

Encapsulation: Hiding stuff with `local`

“Encapsulation” describes the general programming principle that we should “encapsulate” (or hide away) implementation details that are not relevant to how a the function is used (“information hiding”).

Local bindings are not visible, and have no effect outside the local expression. Thus, they can ``hide'' information from other parts of a program.

For example, encapsulation allows us to move helper functions inside the function that uses them, so they are invisible outside the function, do not clutter (or “pollute”) the “namespace” at the top level, and cannot be used by mistake.

If you take CS 246 you will see how object-oriented programming provides flexible support for encapsulation.

Encapsulating a helper function with **`local`**

```
(define (rev lst)
  (local
    [(define (rev/accumulate lst accumulator)
       (cond [(empty? lst) accumulator]
             [else (rev/accumulate
                    (rest lst)
                    (cons (first lst) accumulator))]))]
    (rev/accumulate lst empty)))

(check-expect (rev '(d c b a)) '(a b c d))
```

The only name visible at the top level is **`rev`** itself.

Scope: Reusing names with `local`

Names can be re-used inside a `local` even if they were used in an outer scope.

```
(define s 7)
(define (heron a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))

(check-within (heron s s s) 21.218 0.001)
```

In larger programs it can be hard to keep track of all the names you have used.

A local definition lets you avoid the problem of remembering old names and creating new names.

Efficiency: Avoiding recomputation with `local`

Avoiding recomputation is perhaps the most compelling reason to use `local`.

We often want to re-use the result of a subexpression more than once in a function. If the subexpression includes a function application that requires more than constant time, recomputing the subexpression can greatly impact the overall efficiency of the function.

Avoiding recomputation is an important general principle for making computation faster.

Depending on the context this principle might be called “memoization”, “dynamic programming”, or “caching”.

The largest in a list (from lecture L09)

```
;; produce the largest of a list of numbers
;; largest: (listof Num) -> Num
;; Requires: list is not empty
(define (largest lst)
  (cond ((empty? (rest lst)) (first lst))
        ((> (first lst) (largest (rest lst))) (first lst))
        (else (largest (rest lst)))))
```

When the list is in ascending order, the number of steps roughly doubles with each additional element (n), so that the function exhibits exponential time complexity.

This “exponential blow up” results from computing the largest of the rest of the list twice.

The largest in a list with **`local`**

```
;; produce the largest of a list of numbers
;; largest: (listof Num) -> Num
;; Requires: list is not empty
(define (largest lst)
  (cond [(empty? (rest lst)) (first lst)]
        [else (local
                [(define lr (largest (rest lst)))]
                (cond [(> lr (first lst)) lr]
                      [else (first lst)]))]))
```

Using **`local`** we can compute the largest of the rest (**`lr`**) once and use it twice, avoiding exponential blow up.

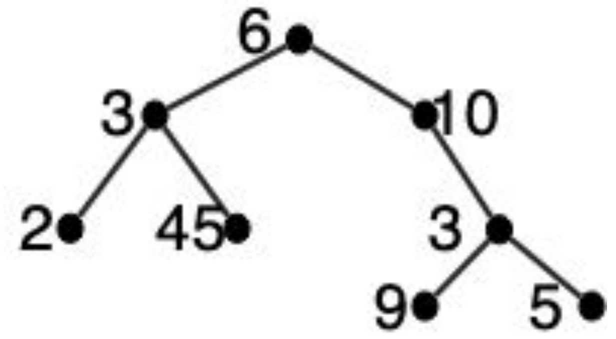
L16.3 Binary trees (again)

Finding a path to a label in a binary tree.

The function **`bt-path`**, searches for a label in a binary tree:

- If the label is not found produce **`empty`**
- If the label is found produce a list of the symbols **`'left`**, **`'right`**, and **`'found`** indicating the path from the root to the item.
- If there are duplicates in the tree, we prefer the leftmost

Note: we aren't assuming the BST ordering property.



Find a path to a label

```
;; Produce a path to a label
;; bt-path: Num BT -> (listof (anyof 'left 'right 'found))
;; Requires: labels are numbers
(define (bt-path label bt)
  (cond [(empty? bt) empty]
        [(= (get-label bt) label) (list 'found)]
        [(bt-contains? label (get-left bt))
         (cons 'left (bt-path label (get-left bt)))]
        [(bt-contains? label (get-right bt))
         (cons 'right (bt-path label (get-right bt)))]
        [else empty]))
```

Efficiency of **`bt-path`**

The worst-case for **`bt-path`** occurs when the tree is shaped like a linked list — that is, a degenerate binary tree where each node has only one child (either always left or always right). What happens when we generate the path to **`4`**?

```
(define bt
  (mk-node 1
    empty
     (mk-node 2
       empty
        (mk-node 3
          empty
           (mk-node 4 empty empty)))))
```

Improving the efficiency of **`bt-path`**

The problem occurs because we first traverse a subtree to determine it it contains the label, and then traverse it again to generate the path.

It would be better if we could traverse each subtree at most once.

Our new approach:

- Traverse the left subtree to generate a path.
- If the path is **`empty`**, traverse the right subtree to generate a path.

To make this work, we need **`local`**.

Linear time **`bt-path`**

```
(define (bt-path label bt)
  (cond [(empty? bt) empty]
        [(= (get-label bt) label) (list 'found)]
        [else (local
                [(define lpath (bt-path label (get-left bt)))]
                (cond [(empty? lpath)
                       (local
                         [(define rpath (bt-path label
                                                  (get-right bt)))]
                         (cond [(empty? rpath) empty]
                               [else (cons 'right rpath)]))]
                      [else (cons 'left lpath)]))]))
```

Sadly, `local` is awkward to use in Racket

There are two unfortunate properties of `local` in Racket that discourage its use.

1) It creates lots of extra nesting and brackets:

```
(local
  [(define ...)
   ... ]
  (...))
```

2) Helper functions defined locally can’t be tested with `check-expect`. As a result, we recommend developing helper functions outside the main function, testing them thoroughly, and then encapsulating them with `local` afterwards.

However, `local` can be essential for efficiency, so don’t ignore it.

L16 Summary

L16: You should know

- How to use a local definition.
- The benefits of a local definition, in terms of clarity, encapsulation, scope, and especially efficiency.
- How identifiers used in local definitions are re-written in the stepper.
- How to deal with the awkward properties of `local`.

Make sure you switch your language level to “intermediate student” for assignments that include this lecture in their coverage.

L16: Allowed constructs

Newly allowed constructs:
`local`

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan boolean? char? char=?
char<? check-expect check-within cond cons cons? cos define
e else empty empty? even? exp expt false first inexact?
integer? length list list? log max min not number? odd? or
pi quotient rational? remainder rest reverse second sin sqr
sqrt string? string=? string<? string->list list->string
sub1 symbol? symbol=? tan third true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```