CS 135 Winter 2020

Tutorial 8: General Trees and Local Definitions

Announcements

- Make sure to change your language level to Intermediate Student when using local.

- The times and locations of office hours are posted on the “Office and Consulting Hours” pages of the course website. Please email us at “cs135@uwaterloo.ca” to set up an appointment outside of these hours.

Goals of this tutorial

You should be able to...

- Understand Mutually Recursive data definitions.

- Write functions that perform Mutual Recursion.

- Step through Local definitions.

- Use Local in your own functions.
Review: Local Definitions
Recall the special form `local` which allows us to create local definitions.

The syntax for `local` is as follows:

```
(local [[(define x1 exp1) . . . (define xn expn)] bodyexp])
```

Clicker Question: Benefits of Local
Which of the following is NOT a reason why we might want to use `local`?

A. Encapsulation: Use `local` to hide parts of the program from each other.
B. Hierarchy: Use `local` to establish parent-child relationships of one function with another.
C. Efficiency: Use `local` to store the result of calculations to avoid recomputation.
D. Readability: Use `local` definitions to rename variables and function calls to be more meaningful.
E. Accumulative Recursion: Use `local` to perform accumulative recursion.

Problem 1: Tries (Mutual Recursion)
A Trie is a general tree where each node stores a single character and the character #$ denotes the end of a word. Each path from the root of the Trie to a leaf node would correspond to a word stored in the Trie. Consider the following data definition and example:

```
(define-struct node (letter children))
;; A Trie is one of:
;; * (make-node #\$ empty)
;; * (make-node Char (listof Trie))
;; requires: a ≤ letter ≤ z, children is non-empty
;; parents cannot have duplicate siblings
```
Problem 1: Tries Example

\[
\text{(define d-trie (make-node #\d} \\
\quad (\text{list (make-node #\o (list (make-node #\g} \\
\quad \quad (\text{list (make-node #\$ empty)))))} \\
\quad (\text{make-node #\v (list (make-node #\e} \\
\quad \quad (\text{list (make-node #\$ empty)))))} \\
\quad (\text{make-node #\$ empty})))) \\
\quad (\text{make-node #\a (list (make-node #\y} \\
\quad \quad (\text{list (make-node #\$ empty)))))))))
\]

d-trie stores the words \textit{dog\$}, \textit{do\$}, \textit{dove\$}, and \textit{day\$}.

Problem 2: Count Words

Using the templates, write a function \texttt{count-words} that consumes a Trie and determines the total numbers of words stored in the Trie.
We can consider a single node with a \#\$ letter to represent a Trie containing only the empty string "$".

\[
\text{(count-words d-trie)} \Rightarrow 4 \\
\text{(count-words (make-node #\$ empty))} \Rightarrow 1
\]
Problem 2: Design Recipe

;; (count-words trie) Produces the number of words in given trie.
;; count-words: Trie → Nat

(define (count-words trie) . . )

;; (count-words/list lotrie) Produces the number
;; of words in a given list of tries, lotrie.
;; count-words/list: (listof Trie) → Nat

(define (count-words/list lotrie) . . )

Extra Practice: contains-word?
Using the provided templates write a function contains-word? that consumes a Trie, and a string and determines if the string corresponds to a word stored in the Trie.

(contains-word? d-trie "dove\$") ⇒ true
(contains-word? d-trie "daze\$") ⇒ false

Extra Practice: Design Recipes
Here's the purpose and contract for some helper functions that might be useful:

;; (contains-word/char? trie loc) Determine whether the given
;; trie contains all characters in the list of characters, loc.
;; contains-word/char?: Trie (listof Char) → Bool

(define (contains-word/char? trie loc) . . )

;; (contains-word/list? lotrie loc) Determine whether one of the tries in
;; list of tries, lotrie contains all characters in list of characters, loc.
;; contains-word/list?: (listof Trie) (listof Char) → Bool

(define (contains-word/list? lotrie loc) . . )
Group Discussion: Efficiency with Local

(define (list-min lon)
  (cond 
    [(empty? (rest lon)) (first lon)]
    [(<= (first lon)
      (list-min (rest lon)))
      (first lon)]
    [else (list-min (rest lon))])))

How can we use local constant definitions to improve efficiency?

CS 135 Winter 2020

Group Discussion: Efficiency with Local

(define (list-min-local lon)
  (cond 
    [(empty? (rest lon)) (first lon)]
    [else (local 
      [(define rest-min (list-min-local (rest lon)))]
      (cond 
        [(<= (first lon) rest-min) (first lon)]
        [else rest-min]))]))

The recursive result is stored in a local constant and used twice. Try re-implementing this function to make it more efficient and design some thorough tests to check if your rewritten function produces the same result as the original.

CS 135 Winter 2020

Clicker Question 1: Local Definitions

(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 (list 1 9 1 23) 0)

What would the following expression produce?

A 1
B 10
C 11
D 34
E Error
Problem 3: Evaluating AltAExp

Rewrite the function eval for a AltAExp. You cannot define any global helper functions.

;; An alternate arithmetic expression (AltAExp) is one of:
;; ⋆ a Num
;; ⋆ (cons (anyof '* '+) (listof AltAExp))

(define-struct ainode (op args))
;; a Arithmetic expression Internal Node (AINode)
;; is a (make-ainode (anyof '* '+) (listof AExp))
;; An Arithmetic Expression (AExp) is one of:
;; ⋆ a Num
;; ⋆ an AINode

;; apply: op (listof AExp) → Num
(define (apply op args)
  (cond 
    [(empty? args) (cond 
      [(symbol = op ' +) 0] 
      [(symbol = op ' ∗) 1])]
    [(symbol = op ' +) (+ (eval (first args))
      (apply op (rest args)))]
    [(symbol = op ' ∗) (* (eval (first args))
      (apply op (rest args)))]))

;; eval: AExp → Num
(define (eval ex)
  (cond 
    [(number? ex) ex]
    [(ainode? ex) (apply (ainode-op ex)
      (ainode-args ex))]))

Clicker Question 2: 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)
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.

```
(define (f x y)
  (local [(define a (+ x 3))
           (define y 4)
           (define (g x) (+ x a))]
         (* 2 (g y))))

(f 7 3)
```