CS 61A Fall 2020

Structure and Interpretation of Computer Programs

Quiz 8 Solutions

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.
- Fall 2020 students: the boxes below are an artifact from more typical semesters to simulate exam environments. Obviously this doesn't apply to this semester's exams, but we just kept the fields to keep our materials looking professional :) Feel free to ignore them.
- For multiple choice questions, fill in each option or choice completely.
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

Last name	
First name	
Student ID number	
CalCentral email (_@berkeley.edu)	
Discussion Section	___ ___ ___
All the work on this exam is my own. **(please sign)**	

1. **Partition Options**

   Implement the following procedure so that it outputs a list of lists, in which each inner list contains numbers no larger than `biggest` and sum to `total`.

```
scm> (partition-options 2 2)
((2) (1 1))
scm> (partition-options 3 3)
((3) (2 1) (1 1 1))
scm> (partition-options 4 3)
((3 1) (2 2) (2 1 1) (1 1 1 1))
```

```
(define (partition-options total biggest)
  (cond
        ((= total 0) (cons nil nil))
        ((or (< total 0) (= biggest 0)) nil)
        (else (let
                ((w (partition-options (- total biggest) biggest))
                (wo (partition-options total (- biggest 1))))
                (helper w wo biggest)))))

(define (helper lst1 lst2 x)
        (if (null? lst1)
            lst2
           (cons (cons x (car lst1)) (helper (cdr lst1) lst2 x))))
```

2. **Malformed Lists**

Recall that a Scheme list is structured like a linked list: every element populates the first slot in a node, and the second slot points to the next node in the list. A malformed list is one that does not properly terminate in a null list. For example, something like `(cons 1 2)` instead of `(cons 1 (cons 2 nil))`. Return a list that has the same elements and structure as the input list but removes all malformed elements. We'll denote malformed lists with dots i.e., `(cons 1 2)` would evaluate to `(1 . 2)` We can also create the same list like so: `'(1 . 2)` *Note:* you likely won't get anything informative by plugging these test cases into code.cs61a.org because our implementation of Scheme currently disallows malformed lists. This is purely a pen-and-paper exercise

```
scm> (fix-lst '(1 . 2))
(1 2)
scm> (fix-lst '(1 2 . 3))
(1 2 3)
scm> (fix-lst '((1 . 2) 3))
((1 2) 3)
scm> (fix-lst '(1 (2 3 . 4) . 3))
(1 (2 3 4) 3)
scm> (fix-lst '(1 . ((2 3 . 4) . 3)))
(1 (2 3 4) 3)

(define (fix-lst lst)
  (cond
        ((null? lst) nil)

        ((not (pair? lst)) (cons lst nil))

        ((pair? (car lst)) (cons (fix-lst (car lst)) (fix-lst (cdr lst))))

        (else (cons (car lst) (fix-lst (cdr lst))))))
```

3. **Notation notation notation**

One annoying thing about Scheme is that it can only understand arithmetic operations that are written in prefix notation. That is, if I want to evaluate an expression, the arithmetic operator must come first, which really goes against everything you were taught as a child. Let's leverage our skills to define a Scheme procedure that accepts arithmetic operations with infix notation, which places operators between operands as you're used to. You only need to support the addition and multiplication operators `*` and `+`, but you need to support order of operations. Define the `calculate` procedure so that it passes the test cases below.

```
scm> (calculate '(1))
1
scm> (calculate '(1 + 2))
3
scm> (calculate '(1 * 2))
2
; Order of operations apply
scm> (calculate '(3 + 2 * 5 + 4))
17
scm> (calculate '(5 * 3 + 2 + 4 * 9))
53
; Parentheses should be handled properly
scm> (calculate '(3 * (2 * 4)))
24
scm> (calculate '(3 * (2 + 4)))
9
scm> (calculate '((3 + 2) + 4))
9
; Parentheses are prioritized higher than order of operations
scm> (calculate '(1 + 2 * (3 + 4)))
15
scm> (calculate '(1 + 2 * (3 + 4 * (5 + 6))))
95
```



```
; Some helper procedures (optional)
(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cddr x) (cdr (cdr x)))

(define (calculate expr)
(cond
  ((not (list? expr)) expr)

  ((null? (cdr expr)) (if (not (list? (car expr))) (car expr) (calculate (car expr))))

  ((list? (car expr)) (calculate (cons (calculate (car expr)) (cdr expr))))

  ((eq? '* (cadr expr)) (calculate (cons (* (car expr) (calculate (car (cddr expr)))) (cdr (cddr expr)))))

  ((eq? '+ (cadr expr)) (+ (car expr) (calculate (cddr expr))))))
```