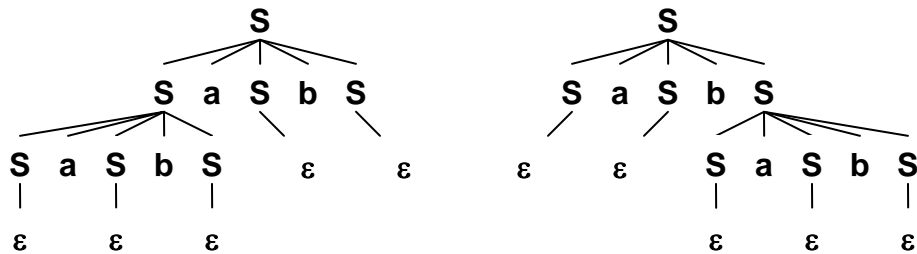


1. List all the strings with length exactly 8 that can be generated using this grammar:
 $S \rightarrow SaSbS \mid \epsilon$. Write these strings in the indicated boxes. [14 points]

aaaabbbb	aaababbb	aaabbabb	aaabbbab	aabaabbb
aabababb	aababbab	aabbaabb	aabbabab	abaaabbb
abaababb	abaabbab	ababaabb	abababab	-----

Is the above grammar ambiguous, or is it unambiguous? Justify your answer. [6 points]

It is ambiguous. For example, the string **abab** has these two parse trees:



2. A *perfect number* is an integer that equals the sum of its proper positive divisors. Thus 6 is perfect because $1 + 2 + 3 = 6$, and 28 is perfect because $1 + 2 + 4 + 7 + 14 = 28$. Write a recursive Impcore function (`perfect m`) that returns 1 when m is perfect, but that returns 0 otherwise. You may write helper functions, and you may call functions provided in the text such as `mod`. But do not use `while` or `set` expressions. [10 points]

(define perfect (m) (= m (sumdivisors m (/ m 2))))

(define sumdivisors (m k)

(if (= k 0) 0 (+ (sumdivisors m (- k 1)) (* (= (mod m k) 0) k))))

3. Suppose Impcore provides a new built-in expression (switch e_0 v_1 e_1 v_2 e_2 ... v_n e_n) with the following meaning. First e_0 is evaluated. If the result is v_1 , then evaluate e_1 ; else if it is v_2 , then evaluate e_2 ; ...; else if it is v_n , then evaluate e_n ; else return the default value 0. Complete these Impcore-style natural operational semantics rules corresponding to the AST node $\text{SWITCH}(e_0, v_1, e_1, v_2, e_2, \dots, v_n, e_n)$. [15 points]

The first rule should handle the k^{th} case, for arbitrary k in the range from 1 to n :

$$\langle e_0, \xi, \phi, \rho \rangle \Downarrow \langle v_0, \xi', \phi, \rho' \rangle$$

$$v_0 \neq v_j \text{ for } 1 \leq j < k$$

$$v_0 = v_k$$

$$\langle e_k, \xi', \phi, \rho' \rangle \Downarrow \langle v, \xi'', \phi, \rho'' \rangle$$

(SWITCH_KTH_CASE)

$$\langle \text{SWITCH}(e_0, v_1, e_1, \dots, v_n, e_n), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi'', \phi, \rho'' \rangle$$

The second rule should handle the default case:

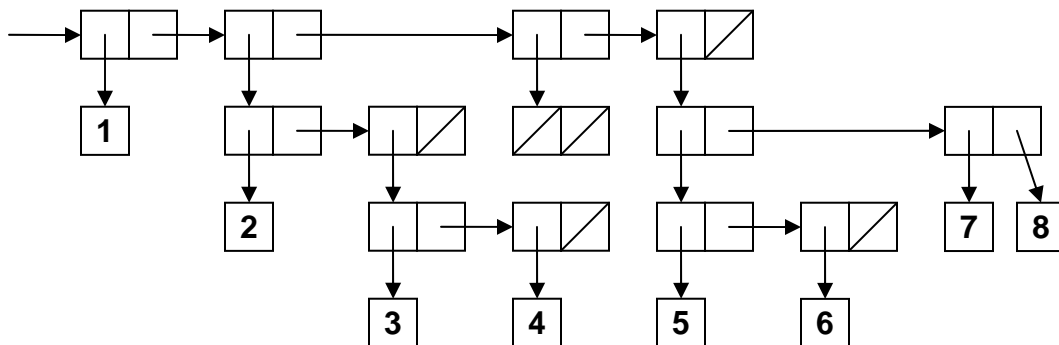
$$\langle e_0, \xi, \phi, \rho \rangle \Downarrow \langle v_0, \xi', \phi, \rho' \rangle$$

$$v_0 \neq v_j \text{ for } 1 \leq j \leq n$$

(SWITCH_DEFAULT)

$$\langle \text{SWITCH}(e_0, v_1, e_1, \dots, v_n, e_n), \xi, \phi, \rho \rangle \Downarrow \langle 0, \xi', \phi, \rho' \rangle$$

4. Draw a diagram that illustrates the internal representation of this S-expression: (1 (2 (3 4)) (()) ((5 6) 7 . 8)) [10 points]



5. Implement a `sum_of_products` function in Scheme, as described below. You may write helper functions. But do not use `while` or `set` expressions.

Example: `(sum_of_products '((1 2 3 4) (5 6 7) (8 9)))` returns 306, because $1*2*3*4 + 5*6*7 + 8*9 = 24 + 210 + 72 = 306$.

Example: `(sum_of_products '((5 10) (15)))` returns 65, because $5*10 + 15 = 65$.

- a. First write the `sum_of_products` function using recursion. Do not use `map`, `foldr`, or `foldl`. [10 points]

```
(define sum_of_products (L)
  (if (null? L) 0 (+ (product (car L)) (sum_of_products (cdr L))))))

(define product (L) (if (null? L) 1 (* (car L) (product (cdr L)))))
```

- b. Next write the `sum_of_products` function using `map`, `foldr`, and/or `foldl`. Do not use any explicit recursion. [10 points]

```
(define sum_of_products (L) (foldr + 0 (map product L)))

(define product (L) (foldr * 1 L))
```

6. Explain what unusual thing happens when each of these Scheme expressions is evaluated.

- a. `((lambda (x) (x x)) (lambda (x) (x x)))` [7 points]

Evaluating this expression yields the same expression repeatedly. This infinite recursion causes our Scheme interpreter to crash due to out-of-memory error.

```
((lambda (x) (x x)) (lambda (x) (x x)))
  ↓
((lambda (x) (x x)) (lambda (x) (x x)))
  ↓
((lambda (x) (x x)) (lambda (x) (x x)))
  ↓
  ...
```

- b. `((lambda (x) (list2 x (list2 (quote quote) x)))
 (quote (lambda (x) (list2 x (list2 (quote quote) x))))))` [7 points]

Evaluating this expression yields the same expression once and then stops.

```

((lambda (x) (list2 x (list2 (quote quote) x)))
 (quote (lambda (x) (list2 x (list2 (quote quote) x))))))
  ↓
((lambda (x) (list2 x (list2 (quote quote) x)))
 (quote (lambda (x) (list2 x (list2 (quote quote) x))))))

```

7. Scheme provides both `let` and `let*` for defining local variables. Consider this example:

```

-> (val a 10)
10
-> (let ((a 20) (b (* a a))) (+ b 1))
101
-> (let* ((a 20) (b (* a a))) (+ b 1))
401

```

With `let`, variable `b` gets value $10 * 10 = 100$. With `let*`, variable `b` gets value $20 * 20 = 400$.

- a. First show that each use of `let*` can be replaced by uses of `let`. That is, write an expression using `let` that is equivalent to `(let* ((x1 e1) (x2 e2) ... (xn en)) e)`.
 Hint: consider the above example. [8 points]

`(let ((x1 e1)) (let ((x2 e2)) ... (let ((xn en)) e) ...))`

Example: `(let ((a 20)) (let ((b (* a a))) (+ b 1)))` returns 401.

- b. Next show that each use of `let` can be replaced by uses of `let*`. That is, write an expression using `let*` that is equivalent to `(let ((x1 e1) (x2 e2) ... (xn en)) e)`.
 Hint: consider the above example. [8 points]

`(let* ((t1 e1) (t2 e2) ... (tn en)) (let* ((x1 t1) (x2 t2) ... (xn tn)) e))`

Example: `(let* ((t1 20) (t2 (* a a))) (let* ((a t1) (b t2)) (+ b 1)))` returns 101.