July 2^{nd} 2004 CMSC 10500-1 Midterm Solutions Duration: 2 Hours
The paper will be graded out of 40.

Comments

The solutions for all the midterm problems can be downloaded by clicking the appropriate link below:

1. Solution to Question 1
2. Solution to Question 2
3. Solution to Question 3
4. Solution to Question 4
5. Solution to Question 5
6. Solution to Question 6
7. Solution to Question 7

Problems

1. Area of a triangle - 5 points

Write a scheme function `area`, which consumes three points on the plane (posn structures), and produces the area of the triangle defined by those points. You may use the following formula without proof:

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

where a, b, c represent the lengths of the three sides of the triangle and $s = \frac{a+b+c}{2}$ is the semi-perimeter of the triangle.

```
> (area (make-posn 0 0) (make-posn 3 0) (make-posn 3 4))
6
> (area (make-posn -4 0) (make-posn 4 0) (make-posn 0 8))
32
> (area (make-posn 5 24) (make-posn 5 19) (make-posn 7 26))
5
```

Solution

1. Write a distance function which takes two `posn` structures and returns the distance between the two points.

2. Use the previous function to calculate `a,b,c` to substitute in the given formula.

3. Note that computation of square roots are in-exact. So even though the third example above should return 5, it actually returns `#i4.99999...`.

2. Preparing your taxes - 15 points

Consider the following scheme structure

```
;; Tax information of an individual
(define-struct tax-data (total-income tax-withheld num-exemptions))

;; Tax deduction numbers
(define std-deduction 5000)
(define exemption-amount 2500)

;; Tax return
;;    type is either 'DueToIRS or 'DueFromIRS
;;    amount is a non-negative number
(define-struct tax-return (type amount))
```

which represents the tax information of an individual.

1. The tax a person owes is a certain percentage of his **taxable income**. Figure 1 gives the appropriate percentage.

2. The **taxable income** is the total income, less **standard deductions** and **exemptions**.

3. The **standard deduction** is a fixed quantity given by the scheme variable `std-deduction`.

4. The total **exemptions** is a fixed quantity (given by the scheme variable `exemption-amount`) multiplied by the number of exemptions claimed.

5. The quantity `tax-withheld` represents the amount of taxes already paid (with held from pay checks).

Write a scheme function `calc-tax-return` which consumes a `tax-data` structure and produces a `tax-return` structure. If the total tax is more than the amount already paid, then the person still needs to pay the difference to the IRS. If the reverse holds, then the person is due money from the IRS. For example,

taxable income	tax percentage
≤ 7000	0
≤ 20000	10
≤ 50000	20
> 50000	30

Figure 1: Tax Slabs

```
> (calc-tax-return (make-tax-data 7000 2000 0))
(make-tax-return 'DueFromIRS 2000)
> (calc-tax-return (make-tax-data 55000 7000 4))
(make-tax-return 'DueToIRS 1000)
```

Comments: Preparing a tax return in real life is usually not so simple. Instead of standard deductions, one usually has itemized deductions which is a whole new can of worms!! Also there are complex rules which determine how many exemptions a person can claim.

Solution

Even though the question seems long, it is quite simple.

1. Write a function `total-tax` which given the `taxable-income` returns the tax amount, according to the given table.
2. Write a function `calc-taxable-income` which given the `tax-data` calculates the taxable income, by subtracting the standard deduction and the exemptions.
3. Write a function `calc-net-tax` which given the `tax-data` calculates the net tax, the person owes (may be negative).
4. Finally, the function `calc-tax-return` checks if the net-tax is positive or negative and returns the appropriate answer.

3. More triangles - 10 points

Write a scheme function `configuration` which consumes three points on the plane and produces exactly one of the following symbols

1. `'CoLinear` if the three points lie on a straight line.
2. `'RightTriangle` if the three points form a right triangle.
3. `'Equilateral` if the three points form an equilateral triangle.
4. `'Isosceles` if the three points an isosceles triangle (but not an equilateral triangle).

5. `'NoneOfTheAbove` if none of the above hold.

You may assume that the three points are all different. Note that since numbers involved may be inexact, you may get unexpected results. Consider replacing checks like `(= a b)` with checks like `(<= (abs (- a b)) 0.0001)` which allow for a very small error.

```
> (configuration (make-posn -4 4) (make-posn 5 -5) (make-posn 0 0))
'CoLinear
> (configuration (make-posn 0 0) (make-posn 3 0) (make-posn 3 4))
'RightTriangle
> (configuration (make-posn -4 0) (make-posn 4 0) (make-posn 0 8))
'Isosceles
> (define a (* 4 (sqrt 3)))
> (configuration (make-posn -4 0) (make-posn 4 0) (make-posn 0 a))
'Equilateral
> (configuration (make-posn 0 0) (make-posn 1 1) (make-posn 3 4))
'NoneOfTheAbove
```

Solution

The issue of computing inexact square roots causes a problem here. If your program did not allow for small errors, you would incorrectly classify the above Equilateral Triangle as an Isosceles triangle. The solution to this problem goes thus:

1. Write a `distance` function which calculates the distance between two points.
2. Write an `almost=?` function which consumes two numbers and checks if they are almost equal (difference less then 0.0001 as suggested).
3. Write functions `CoLinear?, RightTriangle?, Isosceles?, Equilateral?` which consume the pair-wise distances of the three points (i.e. three numbers) and checks if the points satisfy the relevant condition. These functions use `almost=?` instead of $=$.
4. Finally the main function calls these functions in the correct order and returns the appropriate answer. It is important to check for Equilateral before Isosceles, unless your `Isosceles?` function also checks that it is not Equilateral.

4. Approximating e - 10 points

Write a scheme function `approx-e` which consumes a number `n` and produces an approximate value for e using the formula:

$$\mathrm{e} \sim 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$

```
> (approx-e 7)
2.71825396...
> (approx-e 13)
2.7182818284467...
```

Hint: First write a function to calculate the factorial of a given number.

Solution

The simplest solution here, is to write a function to calculate the factorial of a number, and use that to calculate each term in the above series. This follows the standard pattern we have done in class, so I will give another solution.

Another solution is to observe that when we calculate `(factorial 20)`, we also calculate the factorial of all smaller numbers along the way. So it would be a waste to recalculate `(factorial 19)`. To improve the efficiency of the program, observe that the denominator of the $k+1$'st term is just k-times the denominator of the k'th term. So we define a single function to solve the problem. This helper function has to keep track of

1. `n`, the number of terms we need to calculate in the above sum,
2. `a`, the number of terms we have already calculated so far,
3. `den`, the value of the denominator for the a'th term, i.e. `(factorial a)`, and finally
4. `sum`, the sum of the terms encountered so far

We then observe how these parameters change. `n` always stays the same, `a` increases by 1 every time (till it crosses `n`), `den` gets multiplied by the current value of `a` every time, and `sum` gets increased by the reciprocal of `den` every time.

Finally the main function `approx-e` calls the helper function with the right initial values.

5. Number of distinct elements in a sorted list - 10 points

Write a scheme function `count-distinct` which consumes a list of increasing non-zero numbers and produces the number of different numbers present in the list. If the list is not sorted, you are free to produce whatever number you wish.

Hint: Since the list is sorted all occurrences of a given number occurs together. Hence it is enough to count the number adjacent positions which have different numbers.

```
> (count-distinct (cons 2 (cons 2 (cons 3 (cons 3 empty)))))
2
> (count-distinct (cons 1 empty))
1
> (count-distinct (cons 4 (cons 0 (cons 0 (cons -3 (cons -5 empty))))))
4
```

Solution

The key observation is that if the first and second elements of the list are the same, then `(count-distinct list)` equals `(count-distinct (rest list))`. Similarly if the first two elements are different then `(count-distinct list)` is one more than `(count-distinct (rest list))`. So the function `count-distinct` checks if the list has at least 2 elements (if list has lesser than 2 elements, the solution is trivial) and applies the right recursive call.

6. Shopping cart - 15 points

In this problem you implement a shopping cart. Your task is to write a scheme function `total-cost` which consumes a **shopping cart** and a **price list** and produces the total cost.

1. The total cost is the base cost together with a 8.75% tax.
2. The base cost is the sum of the costs of each item multiplied by the quantity.
3. Items which do not have a price mentioned are free!

The following scheme definition defines the structures you need.

```
;; item is a symbol
;; price is a number
(define-struct item-price (item price))

;; item is a symbol
;; qty (quantity) is a integer >= 0
(define-struct item-qty (item qty))
```

Thus a shopping cart is a list-of-item-qty and a price list is a list-of-item-price. Proceed as follows:

1. Write a function `item-cost` which consumes an instance of item-qty and a price list, and produces the cost of the given item (taking the quantity into account).
2. Write a function `base-cost` which consumes a list-of-item-qty (i.e. shopping cart) and a price list and produces the base cost for all the items.

Finally define `total-cost` which adds the tax.

```
> (define pricelist (cons (make-item-price 'LordOfTheRings 20.34)
                          (cons (make-item-price 'Pencil 0.75)
                          (cons (make-item-price 'HTDP 40.00)
                          (cons (make-item-price 'HarryPotterDVD 20.00)
                          (cons (make-item-price 'DaVinciCode 13.75)
```

```
                    (cons (make-item-price 'RuledNotebook 3.75)
                    (cons (make-item-price 'Pen10Pk 2.50)
                    (cons (make-item-price 'PhotoFrame 7)
                    empty))))))))))
> (define cart1 (cons (make-item-count 'LordOfTheRings 1)
                (cons (make-item-count 'HTDP 1)
                (cons (make-item-count 'Pencil 10)
                (cons (make-item-count 'RuledNotebook 5)
                (cons (make-item-count 'DellComputer 10)
                empty))))))
> (define cart2 (cons (make-item-count 'HarryPotterDVD 1)
                (cons (make-item-count 'PhotoFrame 2)
                (cons (make-item-count 'TeddyBear 3)
                (cons (make-item-count 'RuledNotebook 5)
                (cons (make-item-count 'Pen10Pk 4)
                empty))))))
> (total-cost cart1 prices)
94.166625
> (total-cost cart2 prices)
68.240625
```

Solution

This is an instance where one needs to search the pricelist once for every item in the cart.

The `item-cost` function consumes as item-qty and price list, and searches the price list to see for the item (part of the item-qty structure) and returns its cost (part of the item-price structure) multiplied by the quantity (part of the item-qty structure). If item is not found it returns 0 (items not present are free!).

The `base-cost` function consumes the cart and the price list and calls the `item-cost` function to determine the cost of each item, adds up the total cost and returns the base-cost.

Finally the `total-cost` adds a 8.75% tax to this amount.

7. Primality Testing - 15 points

In this problem you get to write a scheme function `factor` which consumes an integer ≥ 2 and returns a factor of that number ≥ 2, using the following outline:

1. Write a scheme function `factor?` which consumes two numbers `a` and `n` and checks if `a` is a factor of `n`, i.e. n/a has no remainder.

2. Write a scheme function `factor-in-range` which consumes three numbers `low`, `high` and `n` and produces some factor of `n` in the range `low` to `high` or `n` if `n` has no factor in the given range.

Finally observe that if `n` has a factor between `2` and `n`, then it has a factor between `2` and `(floor (sqrt n))` (you don't have to prove this). Use this observation to complete this problem.

```
> (factor? 6 42)
true
> (factor? 42 6)
false
> (factor-in-range 7 7 49)
7
> (factor-in-range 7 6 45)
45
> (factor-in-range 2 8 73)
73
> (factor 45)
5
> (factor 73)
73
```

Solution

1. `(factor? a n)` just checks if `(remainder n a)` is zero.
2. `(factor-in-range low high n)` first checks if the range is empty (i.e. `low > high`) and if so returns `n`. Otherwise, if `low` is a factor of `n` return low, else recurse on `(factor-in-range (+ low 1) high n)`.
3. From the given result, `(factor n)` is just `(factor-in-range 2 (floor (sqrt n)) n)`.