Practice Questions for Midterm Exam

README.md

Midterm Exam Preparation

The problems below are similar to what you can expect for the midterm, though some are slightly harder than what I'll ask in the exam.

Static vs. Dynamic Scoping

Consider the following program:


```
1: var b = 2

2: def f(): Unit = {
3:   var b = 3
4:   ()
5: }

6: def g(): Unit = {
7:   f()
8:   b + 4
9: }

10: def h(): Int = {
11:   var b = 5
12:   g()
13: }

14: g()
15: h()
```

1. Under static scoping, what value does `g()` on line 14 return?
2. Under dynamic scoping, what value does `g()` on line 14 return?
3. Under static scoping, what value does `h()` on line 15 return?
4. Under dynamic scoping, what value does `h()` on line 18 return?

Parameter Passing Modes

Consider the following program:

```
def f(x: Int, y: Int) {
```

```
  x = y + 1
  println(x + y)
}

var z = 1
f(z, {z = z + 1; z})
println(z)
```

What does this program print if we make the following assumptions about the parameter passing modes for the parameters `x` and `y` of `params`:

1. both `x` and `y` are call-by-value parameters
2. `x` is call-by-reference and `y` is call-by-value
3. `x` is call-by-value and `y` is call-by-name
4. `x` is call-by-reference and `y` is call-by-name

Deep vs. Shallow Binding

```
def f(x: Int, h: () => Unit): () => Unit = {
  def g(): Unit = println(x)

  if (x == 0) f(1, g)
  else {
    h()
    g
  }
}

var x = 2

def h(): Unit = ()

f(0, h)()
```


What does this program print:

1. assuming static scoping and deep binding
2. assuming dynamic scoping and shallow binding

Call Stacks and Memory Management

Consider the following (faulty) implementation of a merge sort function in C. The function `merge_sort` takes a pointer `a` to the first element of an array of integers and the length of the array `a` as input, and is supposed to sort the array in-place in ascending order. A few test runs of the compiled code on a test machine produce correct results, which seems to indicate that the implementation is correct. However, there is a problem in the code that may cause `merge_sort` to produce unexpected results or even crash. What is this problem? Explain why some simple tests may not reveal this issue. Can you suggest a fix that solves it?

Hint: you don't really need to understand merge sort to spot the problem in the code.

```
int* merge(int* a, int mid, int length) {
  int b[length];
  int i = 0;
  int j = 0;
  int k = mid;
```

```
  while (i < length) {
    if (k >= length || j < mid && a[j] <= a[k]) {
      b[i++] = a[j++];
    } else {
      b[i++] = a[k++];
    }
  }

  return b;
}

void merge_sort(int* a, int length) {
  int mid = length / 2;

  if (mid == 0) return;

  merge_sort(a, mid);
  merge_sort(a + mid, length - mid);

  int* b = merge(a, mid, length);

  for (int i = 0; i < length; i++) a[i] = b[i];
}
```

Lambda Calculus

1. For each of the following lambda calculus terms, compute the set of free variables and show an alpha-renaming of the term such that no variable is bound more than once.

   i. *(λ x. x y) z*

   ii. *(λ x. (λ x. x) x) (λ x. x)*

   iii. *(λ x. (λ x. x) x) (λ x. x) y*

2. Show the reductions to normal form of the following lambda calculus terms, assuming the definitions provided in the class notes:

   i. `or false 1`

   ii. `iszero 1 2 3`

   iii. `pred 1`

   Show each beta-reduction step.

3. Define a lambda term `minus` that takes two Church numerals `m` and `n` and returns a Church numeral that encodes the number `max(0, m - n)`.

Scheme Programming

1. Write a Scheme function `find-nth` that returns the `n`-th element of a list. You may assume that `n` is never greater than the length of the list. Examples:

```
> (find-nth 1 '(a b c d))
a
> (find-nth 2 '(a b c d))
b
> (find-nth 4 '(a b c d))
d
```

2. Write a Scheme function `pack` that packs consecutive elements of a list into sublists:

```
> (pack '(a a a b c c c c d d))
((a a a) (b) (c c c c) (d d))
> (pack '())
()
```

3. Write a Scheme function `encode` that computes the length encoding of a list (you can use the `pack` function as a subroutine).

```
> (encode '(a a a b c c c c d d))
((a . 3) (b . 1) (c . 4) (d . 2))
```

4. Write a Scheme function `drop` that drops every n-the element from a list

```
> (drop 3 '(1 2 3 4 5 6 7 8 9 10))
(1 2 4 5 7 8 10)
```

Challenge yourself and try to implement these functions as efficiently as possible both with respect to run-time/space complexity as well as code complexity (e.g. use tail-recursion, use implementations based on `foldl` / `foldr` etc.)