CS 242. Autumn 2013.
October 21

CS 242 Midterm Exam

This is a **75 minute** exam. You *may* use two double-sided pages of notes that you compiled in advance from any source, but may not use any other reference. The maximum possible score is **75 points**. The last problem is extra credit and worth 10 points towards the 75 point maximum.

Make sure you print your name legibly and sign the honor code below. All of the intended answers may be written within the space provided. You may use the back of the preceding page for scratch work. If you need to use the back side of a page to write part of your answer, be sure to mark your answer clearly.

I acknowledge and accept the Honor Code.

____
(Signature)

______________________ ______________________
(Print your name) *(SUNET ID)*

Prob	# 1	# 2	# 3	# 4	# 5	Extra credit	Total
Score							
Max	15	15	16	14	15	10	75

1. (*15 points*) .... Lambda Calculus

(a) (*8 points*)

Reduce the following lambda term to normal form (i.e. no further reductions possible) using **two** distinct reduction strategies. Assume that free variables are not reducible. Do not use any notational shortcuts and please explicitly state on each reduction if you are performing a beta-reduction, alpha-renaming, or other operation.

$$(\lambda x.(\lambda y.(\lambda x.x\,y))\,x)\,z$$

Answer: *There are only two possible reductions for this lambda expression. Any alpha-equivalent solutions are permissible. Capture-avoiding substitution is necessary to handle one of the reductions.*

Reduction 1:

$\to (\lambda y.(\lambda x.x\,y))\,z$ (β reduction)
$\to (\lambda x.x\,z)$ (β reduction)

Reduction 2:

$\to (\lambda x.(\lambda y.(\lambda a.a\,y))\,x)\,z$ (α renaming)
$\to (\lambda x.(\lambda a.a\,x))\,z$ (β reduction)
$\to (\lambda a.a\,z)$ (β reduction)

(b) (*3 points*)

The above problem show that a lambda term was reduced to the same value using two different reduction strategies. Would it be a good idea to allow for a reduction strategy besides the usual call-by-value for a language like JavaScript? Explain why or why not in a sentence or two.

Answer: No, because different reduction strategies can result in different order of side-effects, making the result of running a program nondeterministic. (An acceptable alternate answer is that the time-space usage of other reduction strategies would be different.)

(c) (*4 points*)

$(\lambda x.x\,x)(\lambda x.x\,x)$ is a lambda term called Ω or the "looping term" that has no normal form: it reduces to itself. However, as we have seen with lazy evaluation, even if an expression contains a non-terminating subexpression, it may still terminate. Give an example of lambda term that contains Ω, but still has a normal form.

Answer: A very simple answer is $(\lambda x.z)\,\Omega \to z$ (where z can be any expression where x does not occur free). The basic principle behind constructing such a term is to discard Ω, so that it disappears after a β reduction.

2. (*15 points*) .... HappyScript

HappyScript is a variant of JavaScript designed for happy people. HappyScript's syntax and semantics are identical to JavaScript's except for two additional symbols: a frownie :(is used to create a new scope, and a corresponding smiley :) is used to close the scope. As in ordinary Javascript, in HappyScript braces { } do not create a new scope. However, :(:) can always be used in place of { } to create a new block-scope.

(a) (*6 points*) Consider the following HappyScript code:

```
function sum(a, b) { return a + b;}
function prod(a,b) { return a * b;}

function f(g, h, x, y)
{
  var a = -5;
  var b = -5;
  if(x > y)
  :(
    var g = h;
  :)

  if(x > 2 * y)
  {
    var a = 0;
    var b = 0;
    for(a = 0; a < x; a++)
    :(
      b += y;
    :)
  }
  return g(a, b);
}
f(sum, prod, 10, 2);
```

i. (*3 points*) Mr. Grumpy believes the additional syntax introduced by HappyScript is unnecessary. Convert the definition of `f` into ordinary JavaScript code such that its behavior is the same as that of HappyScript, **for all inputs**. You may not remove any lines of code besides :(and :).
You do not have to rewrite the entire function. You can write your changes directly on top of the code above—just make sure it is clear what your changes are and where they go.
Answer:

```
function sum(a, b) { return a + b;}
function prod(a,b) { return a * b;}

function f(g, h, x, y)
{
  var a = -5;
  var b = -5;
  if(x > y)
    (function() { var g = h; })();

  if(x > 2 * y)
  {
    var a = 0;
    var b = 0;
    for(a = 0; a < x; a++)
      (function() { b += y; })();
  }
  return g(a, b);
}
f(sum, prod, 10, 2);
```

ii. (*3 points*) The result returned by the call to `f` on the last line is the result of `g(a,b)` at the bottom of function `f`. When the expression `g(a,b)` is executed, what values of `g`, `a`, `b` are used? What is the result? *Briefly* explain your reasoning.
Answer: 30
The body of the first if-statement runs within its own scope, and `g` is declared with the **`var`** keyword. Therefore, the assignment is confined to that scope and does not affect the rest of the program. The body of the second if-statement uses { }, and therefore does not create a new scope; the variable is hoisted. Thus, the changes inside the if-statement *do* affect the rest of the program. The body of the for loop is within its own scope, but since the **`var`** keyword is not used, the `b` refers to the `b` within `f`.
Ultimately, the for loop results in `a` being set to `10` and `b` being set to `20`, and `g` staying as `sum`. The result is `sum(10,20)= 30`.

(b) (*9 points*) Consider the following HappyScript code:

```
function f(x)
:(
  var a = 0;
  function g(z)
  :(
      return z + x;
  :)

  :(
    var x = 5;
    var h = g;
    a = h(x);
  :)

  return a;
:)
f(10);
```

i. (*2 points*) What is the value returned by the call to `f` on the last line?
Answer: 15

ii. (*7 points*) Fill in the missing parts in the following diagram of the run-time structures for execution of this code up to the point immediately before the call to `h(x)` returns. You should write in the number/letter of the appropriate activation record, closure, or code snippet.
Answer:

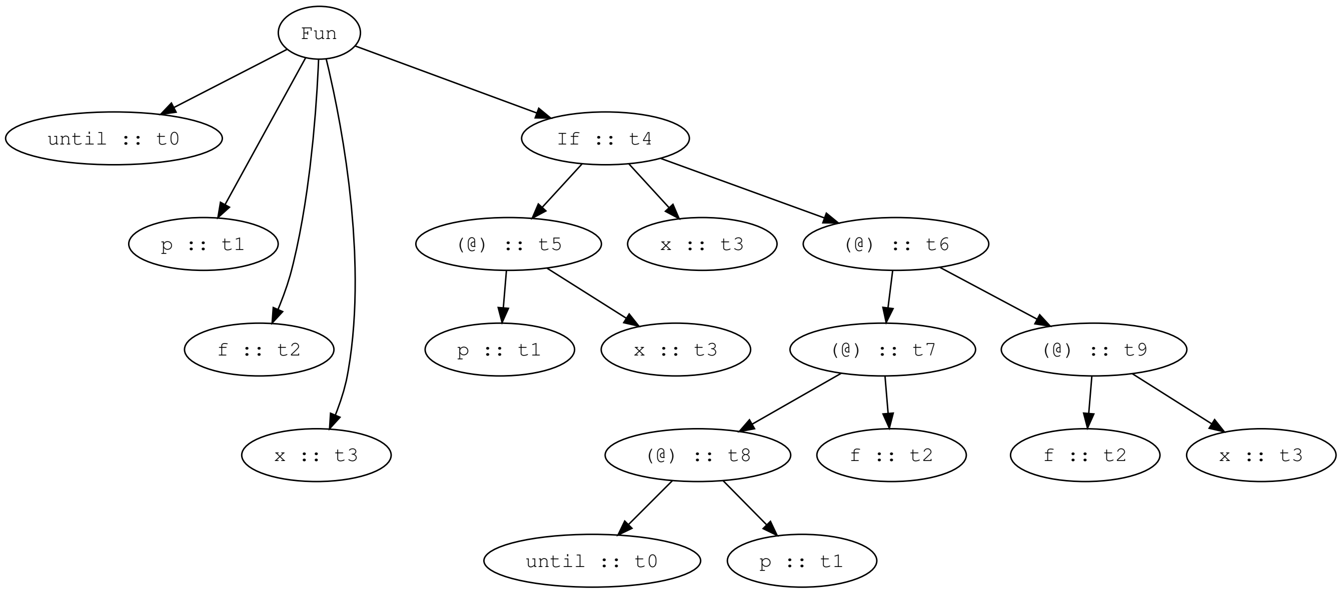
Activation Records				*Closures*	*Compiled Code*
(1)		access link	(0)		
		f	A		
(2)	f(10)	access link	1	A: ⟨ (1) , code for $\underline{f}$ ⟩	| code for f |
		x	10		
		a	0		
		g	B		
(3)		access link	2	B: ⟨ (2) , code for $\underline{g}$ ⟩	| code for g |
		x	5		
		h	B		
(4)	h(5)	access link	2		
		z	5		

3. (*10 points*) .... Type Inference **[10]**

In this problem, you will perform type inference on the function `until`, which is defined below:

```
until p f x = if p x then x else until p f (f x)
```

We have provided you with a parse tree for `until`.



(a) **[5]** Identify the constraints that need to be solved for the function `until`.

Answer:

- `t0 = t1 -> t2 -> t3 -> t4`
- Need two of these (all three is acceptable):
  - `t4 = t3`
  - `t4 = t6`
  - `t3 = t6`
- `t5 = Bool`
- `t1 = t3 -> t5`
- `t7 = t9 -> t6`
- `t8 = t2 -> t7`
- `t0 = t1 -> t8`
- `t2 = t3 -> t9`

(b) **[5]** Solve the type constraints to determine what the type of `until` is. Please note that you must show your work in solving the constraints. A correct solution with no demonstrated solved constraints does not earn credit.

Answer: `until :: (t3 -> Bool)-> (t3 -> t3)-> t3 -> t3`

4. (*14 points*) .... Haskell Type-Classes

Consider the following simple data types representing geometric objects:

```
data Point = Point { x :: Float, y :: Float }
data Rectangle = Rectangle { bottomLeft :: Point, topRight :: Point
    }
data Circle c = Circle { center :: Point
                       , radius :: Float, color :: c }
```

A point is a simple (x, y) coordinate. A rectangle is represented by two points: the point corresponding to the bottom-left corner and the point corresponding to the top-right corner. Finally, a circle is represented by a point and radius. Distinguishing itself from `Point` and `Rectangle`, our `Circle` data type is polymorphic in the color type `c`. So, for example, we can have `Circle RGB`, `Circle GrayScale`, `Circle CMYK`, etc.

In this problem we are going revisit the use and implementation of type-classes. Specifically, consider the `Geometric` class defined bellow:

```
class Geometric a where
  perimeter :: a -> Float
  area      :: a -> Float
```

This class is used to "group" types for which we can compute perimeters and areas. As a starting point, we are going to make our `Rectangle` data type an instance of this class:

```
instance Geometric Rectangle where
  perimeter r = rectPerimeter r
  area r = rectArea r
```

where the perimeter and area functions are defined as:

```
rectPerimeter :: Rectangle -> Float
rectPerimeter (Rectangle bl tr) = let width  = abs (x tr - x bl)
                                      height = abs (y tr - y bl)
                                  in 2 * (width + height)

rectArea :: Rectangle -> Float
rectArea (Rectangle bl tr) = let width  = abs (x tr - x bl)
                                 height = abs (y tr - y bl)
                             in width * height
```

(a) (*3 points*) Define the `Geometric` instance for `Circle`s. Recall that the perimeter and area of a circle with radius r is $2\pi r$ and πr^2, respectively. You can assume the definition of **`pi`** is in scope.

Answers:

```
instance Geometric (Circle c) where
  perimeter (Circle _ r _ ) = 2 * pi * r
  area (Circle _ r _ ) = pi * r * r
```

Note: it is an error to place a class constraint `Geometric c` on the type.

(b) (*3 points*) Recall that Haskell translates type-class declarations into dictionaries, which are represented by polymorphic data-types. For the `Geometric` type-class declaration, the generated dictionary type is:

```
data GeometricD s = MkGeometricD { perimeter :: s -> Float
                                 , area :: s -> Float }
```

Here the type variable `s` corresponds to the shape type.

Further recall that type-class instances are translated into dictionary values. In our example, the `Rectangle` instance is translated into a value `dGeometricRectangle :: GeometricD Rectangle` and the `Circle` instance is translated into a value `dGeometricCircle :: GeometricD (Circle c)`. Complete the definition for the `Rectangle` dictionary using the `rectPerimeter` and `rectArea` functions:

Answers:

```
dGeometricRectangle :: GeometricD Rectangle
dGeometricRectangle = MkGeometricD rectPerimeter rectArea
```

(c) (*4 points*) Below we define a function `eqArea` that determines the two geometric shapes have the same area. Give the most general type for this function.
(**Hint:** you will need type-class constraint(s).)

Answers:

```
eqArea :: (Geometric a, Geometric b) => a -> b -> Bool
eqArea shape1 shape2 = (area shape1) == (area shape2)
```

(d) (*4 points*) Recall that Haskell will translate a function that has a type-class constraint to one that takes a dictionary instead. In turn, when the function is called, the dictionary value (e.g., `dGeometricRectangle`) will be passed in. Fill in the code that will be generated for the `eqArea` function that takes explicit dictionaries.

Answers:

```
eqArea :: GeometricD a -> GeometricD b -> a -> b -> Bool
eqArea d1 d2 s1 s2 = (area d1 s1) == (area d2 s2)
```

5. (*15 points*) .... Tail Recursion, Exceptions and Continuations

In this question we're going to explore the use of continuations.

(a) (*6 points*) Recall that we can use continuation to translate a recursive function into one that is tail recursive. This is called *continuation passing style* (CPS).

Convert the following recursive function `recur` into a tail recursive function.

```
function recur(n) {
  if (n == 0)
    return 1;
  else
    return n*recur(n-1)+1;
}
```

Fill in the rest of the code below. Here `k` is the continuation passed into `recur`.

Answer:

```
function recur(n, k) {
  if (n == 0)
    k(1);
  else
    recur(n-1, function(x){ k(n*x+1) });
}
```

(b) (*3 points*) Recall that tail-recursive functions can be transformed into while-loops. If we allocate closures on the stack, using big-O notation, express the overall size of the stack used when computing `recur(`n`)` with tail-recursion optimization enabled. Briefly justify your answer.

Answer: $O(n)$ since we create a new continuation for each iteration, even though the tail-recursion removes the activation records from the recursive calls.

(c) (*6 points*) The `recur` function defined above will not terminate if the argument is negative. Let's use JavaScript exceptions to handle this exceptional behavior:

```
function recur(n) {
  if (n < 0)
    throw new Error("Invalid argument!");
  if (n == 0)
    return 1;
  else
    return n*recur(n-1)+1;
}
```

This function throws an exception when the argument is invalid. Using this, we can now define a safe version of `recur` that simply returns a default value when the argument is invalid:

```
function safeRecur(n, d) {
  try {
    return recur(n);
  } catch (e) {
    return d;
  }
}
```

Recall that continuations can be used to encode exceptions. Hence, the **`try`/`catch`** blocks can be seen as a syntactic sugar; we can desugar these blocks when transforming a function to CPS by passing in an additional continuation. Rewrite `recur` and `safeRecur` in CPS without using **`try`**, **`catch`**, **`throw`**.
(**Hint:** The anonymous function that we have partially defined for you in `safeRecur` is used in place of the `throw` function.)

Answer:

```
function recur(n, k, throwFunc) {
  if (n < 0)
    throwFunc(new Error("Invalid argument!"));
  if (n == 0)
    k(1);
  else
    recur(n-1, function(x){k(n*x+1)}, throwFunc);
}

function safeRecur(n, d, k) {
    recur(n, k, function(x){ k(d) });
}
```

6. *(10 points)* .... Operational Semantics

The While language is a toy imperative language, similar to the language we saw in class, but with conditionals and a while-loop. The language is defined using the following grammar:

$$\begin{array}{rlll} \text{Number} & n & ::= & 0 \mid 1 \mid \cdots \\ \text{Variable} & x & & \\ \text{Expression} & e & ::= & n \mid x \mid e \oplus e \\ \text{Statement} & s & ::= & \texttt{skip} \mid s; s \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ e\ \texttt{do}\ s \mid x := e \end{array}$$

An expression can be a number (n), a variable (x), or a binary operation ($e \oplus e$) with $\oplus \in \{+, -\}$. A statement can be the simple do-nothing statement (`skip`), a sequencing of statements ($s; s$), a conditional (`if` e `then` s `else` s), a while-loop (`while` e `do` s), or variable assignment ($x := e$).

The semantics for While programs are quite similar to what you've seen in class, with extra rules for while-loops and if-statements. As before, a While program execution, or *configuration*, consists of a statement s and variable-*store* σ, written as $\langle s, \sigma \rangle$. The store is a mapping from a variable to a number, i.e., $\sigma : \text{Variable} \to \text{Number}$. We use $\langle s, \sigma \rangle \longrightarrow \langle s', \sigma' \rangle$ to denote that statement s with store σ reduces—in one step—to statement s' and store σ'.

For reference, we have reproduced the small-step semantics for the fragment of the language you have seen previously:

$$\frac{\sigma(x) = n}{\langle x, \sigma \rangle \longrightarrow \langle n, \sigma \rangle}\ \text{VAR}$$

$$\frac{\langle e_1, \sigma \rangle \longrightarrow \langle e_1', \sigma \rangle}{\langle e_1 \oplus e_2,\ \sigma \rangle \longrightarrow \langle e_1' \oplus e_2, \sigma \rangle}\ \text{BINOPCTXL}$$

$$\frac{\langle e_2, \sigma \rangle \longrightarrow \langle e_2', \sigma \rangle}{\langle n \oplus e_2, \sigma \rangle \longrightarrow \langle n \oplus e_2', \sigma \rangle}\ \text{BINOPCTXR}$$

$$\frac{[\![n_3 = n_1 \oplus n_2]\!]}{\langle n_1 \oplus n_2, \sigma \rangle \longrightarrow \langle n_3, \sigma \rangle}\ \text{BINOP}$$

$$\frac{\langle s_1, \sigma \rangle \longrightarrow \langle s_1', \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \longrightarrow \langle s_1', s_2, \sigma' \rangle}\ \text{SEQCTX}$$

$$\frac{}{\langle \texttt{skip}; s_2, \sigma \rangle \longrightarrow \langle s_2, \sigma \rangle}\ \text{SEQ}$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma \rangle}{\langle x := e, \sigma \rangle \longrightarrow \langle x := e', \sigma \rangle}\ \text{ASSCTX}$$

$$\frac{\sigma' = \sigma[x \mapsto n]}{\langle x := n, \sigma\rangle \longrightarrow \langle \texttt{skip}, \sigma'\rangle}\ \textsc{Ass}$$

Note $[\![n_3 = n_1 \oplus n_2]\!]$ is used to denote the mathematical representation of the numbers and $\oplus$ operator.

The rules for if and while are relatively straightforward. For conditionals, we first reduce the conditional to a number, and then start evaluating the appropriate branch, using zero as false, and any non-zero value as true. We then implement loops in terms of conditional, as seen in class.

$$\frac{\langle e, \sigma\rangle \longrightarrow \langle e', \sigma\rangle}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \longrightarrow \langle \texttt{if}\ e'\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle}\ \textsc{ifCtx}$$

$$\frac{[\![n \neq 0]\!]}{\langle \texttt{if}\ n\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \longrightarrow \langle s_1, \sigma\rangle}\ \textsc{ifTrue}$$

$$\frac{[\![n = 0]\!]}{\langle \texttt{if}\ n\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \longrightarrow \langle s_2, \sigma\rangle}\ \textsc{ifFalse}$$

$$\frac{}{\langle \texttt{while}\ e\ \texttt{do}\ s, \sigma\rangle \longrightarrow \langle \texttt{if}\ e\ \texttt{then}\ (s; \texttt{while}\ e\ \texttt{do}\ s)\ \texttt{else}\ \texttt{skip}, \sigma\rangle}\ \textsc{while}$$

(a) (*3 points*)

Using the above rules, show the reduction of $\langle y := x - 1,\ \sigma\rangle$, where $\sigma = \{x \mapsto 42, y \mapsto 0\}$. Explicitly state which rule(s) you used to perform a reduction (don't forget to provide the context rules!)

Answer:

$$\begin{array}{rcl} & & \langle y := x - 1, \{x \mapsto 42, y \mapsto 0\}\rangle \\ \textsc{assCtx/binOpCtxL/var} & \longrightarrow & \langle y := 42 - 1, \{x \mapsto 42, y \mapsto 0\}\rangle \\ \textsc{assCtx/binOp} & \longrightarrow & \langle y := 41, \{x \mapsto 42, y \mapsto 0\}\rangle \\ \textsc{ass} & \longrightarrow & \langle \texttt{skip}, \{x \mapsto 42, y \mapsto 41\}\rangle \end{array}$$

(b) (*7 points*)

Our reduction rule while uses an if-statement to specify the semantics for the `while` construct. Suppose that our language did not have if-statements. We might try to define the semantics using the following rules:

$$\frac{\langle e, \sigma\rangle \longrightarrow \langle n, \sigma\rangle \quad [\![n \neq 0]\!]}{\langle \texttt{while}\ e\ \texttt{do}\ s, \sigma\rangle \longrightarrow \langle s; \texttt{while}\ n\ \texttt{do}\ s, \sigma\rangle}\ \textsc{whileTrue}$$

$$\frac{}{\langle \texttt{while}\ 0\ \texttt{do}\ s, \sigma\rangle \longrightarrow \langle \texttt{skip}, \sigma\rangle}\ \textsc{whileFalse}$$

Unfortunately, there are two errors in these new semantics. The following question asks you to describe what these errors are.

i. (*2 points*) Consider the execution of the statement $\langle \texttt{while } x \texttt{ do } x := x - 1,\ \sigma\rangle$, where , $\sigma = \{x \mapsto 1\}$. This loop is expected to run once, and then terminate. In a sentence or two, informally explain what goes wrong with the new semantics.
Answer: The rewrite to $\langle s; \texttt{while } n \texttt{ do } s, \sigma\rangle$ is incorrect: the conditional is not re-evaluated for every iteration. So, the program will loop forever. (Actually, strictly speaking, it will not loop forever, because in the resulting state $while 1 s$, there are no reductions from 1, so execution gets stuck. We accepted both answers.)

ii. (*2 points*) Consider the execution of the statement $\langle \texttt{while } (0{+}0){+}0 \texttt{ do skip},\ \{\}\rangle$. The loop is expected to not run. In a sentence or two, informally explain what goes wrong with the new semantics.
Answer: The form of expression e is limited: it can only be an expression which is exactly just zero. However, $(0 + 0) + 0$ takes two steps to evaluate; thus execution is stuck.

iii. (*3 points*) Define $\longrightarrow^*$ to be the transitive closure of our evaluation relation $\longrightarrow$, i.e., the minimal transitive relation that contains $\longrightarrow$. Using $\longrightarrow^*$, provide a fixed version of the while rule which does not use if-statements, but does not have the two problems we've demonstrated above. (**Hint:** Your rule will no longer be a small-step semantics.)
Answer:

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma\rangle \quad [\![n \neq 0]\!]}{\langle \texttt{while } e \texttt{ do } s, \sigma\rangle \longrightarrow \langle s; \texttt{while } e \texttt{ do } s, \sigma\rangle}\ \textsc{whileTrue}$$

Extra 3 points were given for also correcting the WHILEFALSE rule, which had an unintended mistake:

$$\frac{\langle e, \sigma\rangle \longrightarrow^* \langle n, \sigma\rangle \quad [\![n = 0]\!]}{\langle \texttt{while } e \texttt{ do } s, \sigma\rangle \longrightarrow \langle \texttt{skip}, \sigma\rangle}\ \textsc{whileFalse}$$