CS 558: Programming Languages **Final Exam** **March 18, 2015**

This exam has 5 questions. One (8.5" x 11") page of notes allowed. Answer all 5 questions in the space provided.

Student Name (neatly in block letters): _____

1. **Short Answers** (6 points each = 30 points total.).

   (a) Explain the difference between parametric polymorphism and ad-hoc polymorphism. Give an example of a language that uses each kind.

   Parametric polymorphism - when the same code is used at every type. Haskell , OCaml, Java generics. Often uses boxing so all values have the same size.
   AdHoc polymorphism - when different code is run at every type. Sometimes called overloading. Haskell classes, C++ templates, Java overloading.

   (b) Explain the differences between observational semantics and axiomatic semantics.

   Observational semantics uses judgments to define how an abstract interpreter would work. The judgments usually contain an environment, some state, a language construct, and a result or value.
   Axiomatic semantics (aka Hoare semantics) uses pre and post (P {s} Q) logical conditions to reason about language features.

   (c) What are the purposes of having a module system in a language?

   Reuse, abstraction, name space control, separate compilation, interface design and control, track dependencies, break large programs into related components.

   (d) What is the difference between type *checking* and type *inference*? Give an example of a language that uses each.

   Type checking, checks that a program uses its parts consistent with the type annotations provided by the programmer. Java, C, C++
   Type inference figures out a consistent type for each language fragment, based upon the types of the constants, and variables in those fragments, and the context in which they appear.

   (e) What is a closure? What is the purpose of a closure? Which languages use closures?

   A closure is an implementation technique for functions. It contains two parts. First, the static environment of where the function was defined. This provides values for the functions free variables. Second, is the code to be executed. Used to implement first class functions.
   Used in almost any language that has nested functions and static scoping. Haskell, O'Caml, Java.

2. **Type Inference** (7 points each = 21 points total).

   Below we outline several type inference rules for a language similar to E7. Each rule is made from one or more typing judgments (and other auxiliary judgments). A typing judgment has the following form.

   $\Gamma \vdash e : t$

Where Γ is an environment mapping variable names to types. e is syntactic term of the language, and t is a type. A judgment states that the term e has type t in a given context Γ (which tells about the types of term variables that appear in e). We also have several auxiliary judgments.

- $\Gamma(v) = t$, which states that the environment Γ maps the term variable v to the type t.
- $\Gamma(x \Rightarrow t)$, describes an environment with the same behavior as Γ, except that it maps the variable x to t. If x was already mapped to some type s in Γ, then this old mapping for x is hidden in $\Gamma(x \Rightarrow t)$

Example rules for function application and anonymous functions are given below.

$$\frac{\Gamma \vdash f : (a \to t) \quad \Gamma \vdash x : a}{\Gamma \vdash (f\ x) : t} \qquad \frac{\Gamma(x \Rightarrow a) \vdash e : t}{\Gamma \vdash (\lambda(x)e) : (a \to t)}$$

Complete the following rules for if-then-else, an algebraic datatype predicate for a `cons` (as opposed to a `nil`) cell. and an assignment expression.

$$\frac{\Gamma \vdash x : \mathit{Bool} \quad \Gamma \vdash y : t \quad \Gamma \vdash z : t}{\Gamma \vdash (\texttt{if}\ x\ y\ z) : t}$$

$$\frac{\Gamma \vdash \texttt{cons} : a \to [a] \to [a] \quad \Gamma \vdash x : [a] \quad t = \mathit{Bool}}{\Gamma \vdash (\texttt{?cons}\ x) : t}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (\texttt{:=}\ e_1\ e_2) : t}$$

3. **Garbage Collection.** (18 points). A garbage collection system dynamically allocates free memory, and reclaims such memory when it is no longer in use. For each collection algorithm below list its advantages, disadvantages and its costs as a function of some system parameter. Be sure and explain what the parameter is.

   - Reference Counting (4 points)

     Simple, Memory reclaimed as soon as it becomes free. Cost proportional to memory in use. Can't be used for cyclic structures. Can be expensive to track when pointers are released (can lead to a long list of cells to be decremented). Can lead to fragmentation.
   - Mark and Sweep (5 points)

     Can be used for cyclic structures, cost proportional to the size of the heap. Simple, two phase approach can lead to long pauses. Doesn't move memory in use (good sometimes) but may lead to bad cache performance (fragmentation).
   - Two Space Collector (5 points)

     Can be used for cyclic structures, cost proportional to the size reachable memory. Compacts reachable memory for good cache performance. Only half the heap can be in use at any time.

- Generational Collector (4 points)

  Can be used for cyclic structures, cost proportional to the size reachable memory in the latest generation. Can be very fast as the amount of reachable memory in the latest generation is usually very small. Compacts reachable memory for good cache performance (eventually). Full collections can take longer.

4. **Small programs in language E7** (4 points each = 16 points total).

   Below I have written an E7 datatype definition for arithmetic expressions, and a function `f`. There are 2 kinds of arithmetic expressions. Constants (like 3 represented as `(#const 3)`), and Subtractions (like $(2-5)$ represented as `(#sub (#const 2) (#const 5))`). Recall that E7 is a typed language and that functions require type annotations.

```
(module Expression in (sig (type (Int)))
                  out everything)
(data (Exp)
   (#const (Int))
   (#sub (Exp) (Exp)))

(fun f Int (x Int) (+ x 5))
```

   In this question you will do several things. Add a new kind of expression, write a few functions over Exp, and define an abstract datatype that implements a certain signature.

   (a) Alter the algebraic datatype `Exp` by adding a new kind of expression, multiplication, representing things like (3 * 5).

```
(data (Exp)
   (#const (Int))
   (#sub (Exp) (Exp))
   (#mult (Exp) (Exp)))
```

   (b) Write a function `eval` that evaluates the new version of `Exp` to an `Int`.
   E.g. `(@eval (#mult (#const 2) (#const 5)))` returns 10.

```
(fun eval Int (x Exp)
  (if (?const x) (!const 0 x)
      (if (?sub x) (- (@eval (!sub 0 x)) (@eval (!sub 1 x)))
          (* (@eval (!mult 0 x)) (@eval (!mult 1 x))))))
```

   (c) Write a function `count` that counts the number of constants (like 5) in the new version of `Exp`. E.g. `(@count (#mult (#const 2) (#const 5)))` returns 2.

```
(fun count Int (x Exp)
  (if (?const x) 1
      (if (?sub x) (+ (@count (!sub 0 x)) (@count (!sub 1 x)))
          (+ (@count (!mult 0 x)) (@count (!mult 1 x))))))
```

   (d) Study the signature file below, and then define an *abstract datatype* that defines exactly the operations in the signature. Hint type `A` is an abstraction of `Exp`.

```
(defsig ExpSig
  (sig (type (A))
       (val inject (Int -> A))
       (val minus (A -> A -> A))
       (val times (A -> A -> A))
       (val negate (A -> A))))

(adt (A) (Exp)
  (fun inject (A) (x Int) (#const x))
  (fun minus A (x A y A) (#sub x y))
  (fun times A (x A y A)  (#mult x y))
  (fun negate A (x A) (#sub (#const 0) x)))
```

5. **Data Organization.** (15 points total)

   We have seen many different types of languages and systems for organizing data. For each kind of data organization listed below. Define the term, and then list the merits of this organization. Use examples where appropriate. Use only the space provided on this page. Think carefully about what you want to say. Short concise answers that get to the important issues will get more credit than long rambling ones.

   (a) **Arrays.** As found in languages like C and Fortran.

   Contiguous storage of values all of which have the same type. Individual elements are accessed by position (often called indexing) in constant time. Can be allocated statically, or dynamically in the heap. Arrays have as fixed size determined when they are allocated and do not grow or shrink.

   (b) **Algebraic Datatypes.** As found in languages like Haskell.

   Data allocated dynamically in the heap. CHaracterized by their structure, not their operatons. Often the data is a sum-type , that is an element can take one of a finite number of forms. For example, a list is either `(#nil)` or `(#cons 5 xs)`. The operations on algebraic types are (1) construction `(#cons 2 (#nil))`. (2) Predicate testing (which of the finite forms the value is) `(?cons z)`. (3) Selection of one of the multiple fileds a form might have `(!cons 1 x)`.

   Algebraic datatypes can be unbounded in size, and can grow or shrink at run time. Often used to implement enumerations, products, or tree like structures. Many languages (Haskell, O'Caml) also support pattern matching in algebriac datatypes.

   (c) **Objects.** As found in languages like Java.

   Objects are a kind of abstract datatype in that they are charaterized by the messages they can respond to. Language bulit around objects are called object oriented. Objects are dynamically allocated and contain both static data and methods (functions). In objects the data can be mutable, and the methods can be designed to be run only for their effects, but they may also return values. Objects are often part of an inheritance mechanism. Objects have an implict argument (called `self` or `this`) which is dynamically scoped, so a particular code fragment may return differenct values depending upon the flow of control to that code fragment.

   (d) **Abstract Datatypes.** As found in languages like Clu and E7.

   An abstract datatype organizes some data and a bunch of opertations on that data. The struture of the data is abstract and can only be determed by the implementing code. The operatons are classified into *constructors* that create values of the abstract type, and *observers* which return internal values of the structure.