CS242: Autumn 1999
December 9, 1999

CS 242 Final Exam

1. (*10 points*) .... True or False

Mark each statement *true* or *false*, as appropriate.

______ (a) A partial function from A to B is the same as a total function from a subset of A to B.**Answer:** TRUE

______ (b) Lisp `(cond (x y) (true z)` is equivalent to ML `if x then y else z`, assuming that `x` is a boolean expression and `y` and `z` are expressions that would have the same ML type. **Answer:** TRUE

______ (c) The variable x occurs free in $(\lambda y.((\lambda x.\ y)\ x))\ z$. **Answer:** TRUE

______ (d) α-conversion only changes the names of bound variables. **Answer:** TRUE

______ (e) The ML type inference algorithm can compute a type for expressions that do not contain type informations about the variables that appear in them.**Answer:** TRUE

______ (f) Concurrent garbage collection can proceed in parallel without stopping the program.**Answer:** FALSE

______ (g) The Java compiler was the first programming language implementation to compile source code to bytecode.**Answer:** FALSE

______ (h) Java interfaces are more general than C++ abstract classes.**Answer:** FALSE

______ (i) Garbage collection is easier for Java than for C++ because all Java objects are on the heap, while C++ allows objects on the stack.**Answer:** FALSE

______ (j) The Java memory model (specifying how threads may interact using shared memory) is simple and easy to understand.**Answer:** FALSE

Answer:

2. (*8 points*) .... Compile time and run time

For each of the following program properties, check *compile-time* or *run time* or neither, as appropriate. More specifically, if the property can be determined by some algorithm that is given the program text but not the program input, check *compile time*. If the property cannot be determined at compile time, but all violations of the property can be determined while the program is running on specific input, check *run time*.

Property	*Compile time*	*Run time*
All variables are initialized where they are declared		
Program execution halts		
All array references are within declared array bounds		
All casts in a Java program succeed without raising an exception		
A given C++ program is statically type correct		
Every variable that is declared also appears in some expression		
Return values from system calls are checked in calling statements		
Two variable names refer to the same location		

Answer:

Property	*Compile time*	*Run time*
All variables are initialized where they are declared	X	
Program execution halts		
All array references are within declared array bounds		X
All casts in a Java program succeed without raising an exception		X
A given C++ program is statically type correct	X	
Every variable that is declared also appears in some expression	X	
Return values from system calls are checked in calling statements	X	
Two variable names refer to the same location		X

3. (*12 points*) .... Short Answer

Answer each question in a few words or phrases.

(a) (*2 points*) Why is tail recursion elimination useful?

Answer: Reduce space requirement of calls to tail-recursive functions.

(b) (*2 points*) What operations are needed to construct a circular list in Lisp?

Answer: Side effects.

(c) (*2 points*) When would you choose to use ML instead of Lisp?

Answer: Type checking, more expressive data-structuring mechanisms, etc.

(d) (*2 points*) When are static fields of a Java class initialized?

Answer: At class load time.

(e) (*2 points*) A Java programmer can start garbage collection by calling `System.gc()` or `Runtime.gc()`. Why would a programmer want to start garbage collection instead of waiting until the system decides that garbage collection is needed?

Answer: If `finalize` methods free system resources, other do other useful clean-up.

(f) (*2 points*) A Java programmer decides to call the garbage collector after every function return, so that objects allocated by the call will be collected after the return. Will this collect all of the objects allocated by the function call? Why or why not?

Answer: If pointer to an object is passed out of function scope, then this object will nto be garbage collected.

4. (*8 points*) .... Definitions

Define the following terms, in one or two sentences each.

(a) object: **Answer:** (see glossary)

(b) subtype: **Answer:** (see glossary)

(c) dynamic lookup: **Answer:** (see glossary)

(d) class interface: **Answer:** (see glossary)

5. (*11 points*) .... Denotational Semantics

Perl is a programming language that was designed for scanning text files and extracting information from them. Perl is often used to write CGI scripts, which may run in privileged mode. For this reason, Perl programmers may be concerned that tricky text input might cause their program to make undesirable system calls.

The Perl implementation performs a set of security checks when it is run in *taint* mode. Taint checks are designed to make sure that arguments to system calls are not controlled by user input in certain ways. More specifically, command line arguments, environment variables, results of certain system calls, and all file input are marked as "tainted". Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes. For example, if system(e) causes Perl to pass the string argument e to a shell for parsing and execution, then e must be untainted.

In general, any variable set to a value derived from tainted data will be considered tainted. However, untainted information can be extracted from a tainted variable using string matching operations. Intuitively, the reason for this is that the programmer is assumed to use string-matching patterns that will avoid security problems. For example, if an Perl command will write to a file, then string matching can be used to make sure that only characters appear in the user-supplied filename. This allows the programmer to protect against embedded shell commands and other security problems that a malicious user might try to place inside a file name.

Since we do not expect you to know Perl, we will present a simplified version of Perl taint checking as a nonstandard denotational semantics for an expression language. The expressions of our example language are given by the grammar

$$\begin{array}{lcl} e & ::= & \mathit{var} \mid \mathit{cst} \mid \mathsf{concat}(e,e) \mid \mathsf{match}(e,e) \\ \mathit{var} & ::= & x \mid y \mid z \mid \ldots \\ \mathit{cst} & ::= & "\mathit{symbols}" \\ \mathit{symbols} & ::= & \epsilon \mid a\ \mathit{symbols} \mid b\ \mathit{symbols} \mid c\ \mathit{symbols} \mid \ \ldots \end{array}$$

In words, an expression is a variable, constant, or expression formed using one of the two string functions, concat or match. A variable is a single letter like x, y, or z, and a constant is a sequence of symbols enclosed in double quotes. A string of symbols is either empty (indicated by ϵ in this grammar) or a symbol followed by a string of symbols. While only letters a, b, c, ... are listed here, assume that strings can also contain other symbols such as "." and "*".

The value of an expression is always a string, but may be the empty string. The value of $\mathsf{concat}(e_1, e_2)$ is the concatenation of the two strings, and the value of $\mathsf{match}(e_1, e_2)$ is the substring of e_2 that matches the pattern e_1. When string e_1 is regarded as a pattern, the letters match themselves, and other symbols may have special meanings.

In the denotational semantics of these expressions, the *meaning* of an expression e is a function $\mathcal{V}[\![e]\!]$ from environments to values, where an environment is a mapping from variables to values. The non-standard semantics we will use in this problem has the standard form, but the set of values we will use is simplified to the two possible values,

$$\mathit{Values} = \{\mathit{taint}, \mathit{untaint}\}$$

The meanings of variables and constants are

$$\begin{array}{lcl} \mathcal{V}[\![\mathit{var}]\!](\eta) & = & \eta(\mathit{var}) \\ \mathcal{V}[\![\mathit{cst}]\!](\eta) & = & \mathit{untaint} \end{array}$$

In words, a variable may be tainted or untainted (depending on how it was set), but a constant is always untainted. The reason for considering every constant untainted is that a constant is written as part of the program, and therefore does not come from user input. Since Perl tainting assumes that the programmer writes programs carefully, string constants written by the programmer are considered untainted. As described above, a concatenation involving a tainted string is tainted, but matching an untainted pattern against any expression gives an untainted value.

$$\mathcal{V}[\![\mathsf{concat}(e_1, e_2)]\!]\eta \;=\; \begin{cases} \mathit{tainted} & \text{if } \mathcal{V}[\![e_1]\!]\eta = \mathit{tainted} \text{ or } \mathcal{V}[\![e_2]\!]\eta = \mathit{tainted} \\ \mathit{untainted} & \text{otherwise} \end{cases}$$

$$\mathcal{V}[\![\mathsf{match}(e_1, e_2)]\!]\eta \;=\; \begin{cases} \mathit{tainted} & \text{if } \mathcal{V}[\![e_1]\!]\eta = \mathit{tainted} \text{ and } \mathcal{V}[\![e_2]\!]\eta = \mathit{tainted} \\ \mathit{untainted} & \text{otherwise} \end{cases}$$

Questions:

(a) (*3 points*) Show how to determine whether the expression $\mathsf{match}(\text{"}ab{*}cd\text{"}, \mathsf{concat}(x, y))$ is tainted in environment η_0 with $\eta_0(x) = \mathit{tainted}$ and $\eta_0(y) = \mathit{untainted}$.

Answer: $\mathcal{V}[\![\mathsf{match}(\text{"}ab * cd\text{"}, \mathsf{concat}(x, y))]\!]\eta_0 = \mathit{untainted}$ since $\mathcal{V}[\![\text{"}ab * cd\text{"}]\!]\eta_0 = \mathit{untainted}$

(b) (*2 points*) Show how to determine whether the expression $\mathsf{match}(x, \mathsf{concat}(y, z))$ is tainted in environment η_1 with $\eta_1(x) = \mathit{tainted}$ and $\eta_1(y) = \eta_1(z) = \mathit{untainted}$.

Answer: $\mathcal{V}[\![\mathsf{match}(x, \mathsf{concat}(y, z))]\!]\eta_1 = \mathit{untainted}$ since $\mathcal{V}[\![\mathsf{concat}]\!]\eta_1 = \mathit{untainted}$

(c) (*2 points*) A weakness in Perl tainting is that any expression can be converted to an untainted expression. Assuming that the pattern $.*$ matches any string, write an expression containing e that will have the same string value as e, but will always be untainted, regardless of whether e is tainted.

Answer: $\mathsf{match}(\text{"}.*\text{"}, e_2)$

(d) (*4 points*) We can describe a more conservative tainting method by distinguishing between constant e_1 and untainted e_1 in $\mathsf{match}(e_1, e_2)$. Write clauses that are consistent with the discussion of tainting above, and that (i) allow a match against a constant pattern to be untainted if $cst \neq \text{"}.*\text{"}$, and (ii) make a match of a non-constant pattern against a tainted string tainted.

Answer:

$$\mathcal{V}[\![\mathsf{match}(cst, e_2)]\!]\eta \;=\; \begin{cases} \mathit{tainted} & \text{if } cst = \text{"}.*\text{"} \\ \mathit{untainted} & \text{otherwise} \end{cases}$$

$$\mathcal{V}[\![\mathsf{match}(e_1, e_2)]\!]\eta \;=\; \begin{cases} \mathit{tainted} & \text{if } \mathcal{V}[\![e_2]\!]\eta = \mathit{tainted} \\ \mathit{untainted} & \text{otherwise} \end{cases}$$

6. (*10 points*) .... ML Type Checking

Consider the following ML code:

```
fun foo(x) = x + 3;
fun bar(f) = f(2) + 3;
fun f(g,h) = g(h) + 2;
```

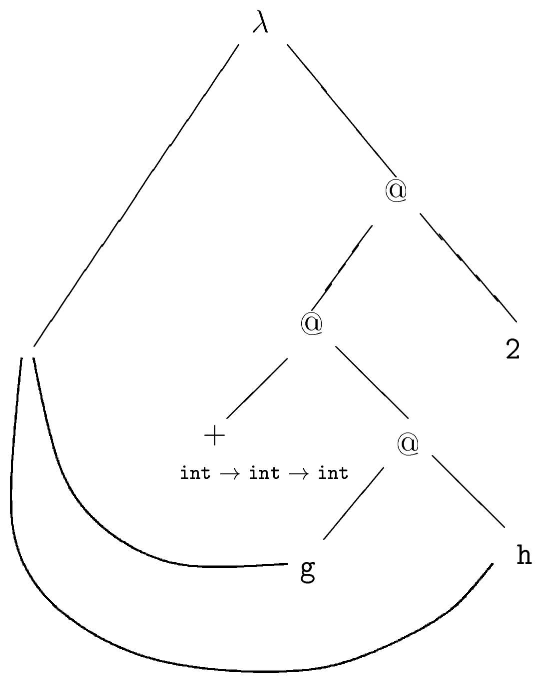
(a) (*4 points*) Assuming that + denotes integer addition, what are the types of:

```
foo
bar
```

Answer: `int->int,(int->int)->int`

(b) (*6 points*) Use the parse graph below to calculate the ML type for the function

```
fun f(g,h) = g(h) + 2;
```



Answer: `(('a->int) * 'a)->int`

7. (*16 points*) .... Nested functions in C

In ML, LISP, and most other functional languages, it is legal to declare a "local function," i.e., a function defined within the scope of another function. For example, in LISP, you might write:

```
(define f (lambda ()
            (let* ((i 3)
                   (g (lambda () i)))
             (print (g))
            g)))
(define main (lambda () (print (funcall (f)))))
```

This small program declares a function `f` which declares a local variable `i` and a local function `g`. The function `g` simply returns the value of `i`. When run this program will print 3 twice.

Because `f` returns a function, this program contains an example of the "upward funarg problem." As we discussed, ML solves this problem by placing both the activation record for the call to `f` and the closure for `g` on the heap.

In ANSI/ISO C, there are no local functions, so there is no way to write an equivalent program in C. However, the Free Software Foundation's C compiler (known as GNU CC, or GCC) does allow local function declarations. Here's how you could write the equivalent program in GNU C:

```
#include <stdio.h>
typedef int (*fn_t)();
fn_t f(){
   int i = 3;
   int g(){return i;}
   printf ("%d\n", g ());
   return &g;
}
int main () {
   printf ("%d\n", (*f())());
}
```

GCC compiles local functions in the usual way, except that references to the activation record of an enclosing function are done via a static link, like in ML. A particular instance of a local function is a piece of code (called a "trampoline") placed on the stack, that sets the static chain and jumps to the beginning of the code for the compiled function. The trampoline serves the same purpose as a closure. Unlike ML and LISP, however, GCC places both records and trampolines on the stack and makes no specific effort to solve the upward funarg problem.

(a) (*5 points*) The output of the GNU C program above is:

```
3
-1073743424
```

Explain why this program does not print `3` twice, as you might expect. Where does the second number come from?

Answer: The environment of `g` is popped off the stack. The second value is some random data sitting in the location where the code for `g` expected to find the value of `i`.

(b) (*4 points*) Why do ML and LISP deviate from stack ("last-in/first-out") storage management for closures and activation records?

Answer: Lisp and ML preserve the static environment of locally-declared functions by keeping closures and activation records allocated until the program is no longer able to call the functions that use them.

(c) (*3 points*) What might be some advantages of placing trampolines and activation records on the stack, even when local functions are used?

Answer: Simplicity and efficiency of implementation. Specifically, if activation records are not on the stack, you would have to use garbage collection or some other mechanisms to later decide when to deallocate them.

(d) (*2 points*) Do you think the decision that the GNU C designers made, namely the decision to place trampolines and activation records on the stack, is consistent with the basic design goals of C? Why or why not?

Answer: Yes. This keeps function calls efficient. It is also consistent with other C design decisions to just do whatever is efficient and let the programmer figure out how to cope.

(e) (*2 points*) Ignoring efficiency, what basic property of C would make it difficult to use the kind of memory management techniques that are used in Lisp and ML?

Answer: Cannot do garbage collection easily in C if pointers are not distinct from data.

8. (*15 points*) .... Data Representation in Scheme and Java

In Lisp, Scheme, and ML, polymorphism requires a uniform representation of data. In Lisp and Scheme, for example, the `car` and `cdr` of a `cons` cell may contain any value of any type, This means that values of any type must be able to fit into the storage allocated for the slots of the `cons` cell. Since the parameter of a function can have any type, the code implementing a function call must similarly be able to accept any type of value.

Lisp, Scheme, and ML implementations solve the one-size-fits-all problem by representing all values with exactly one machine word:

- If a value is smaller than a single word, then some bits are set in a specific pattern that identifies the type of the value. For example, some Scheme implementations represent data such as characters, 30-bit integers, the empty list, and booleans as 32-bit patterns whose least-significant bit is zero. The next bit is zero if the 32-bit pattern represents a 30-bit integer. Otherwise, the second least-significant bit is one and the next six bits are used to provide the type information, leaving three bytes for the actual data.

- If a value is larger than a single word (e.g., a cons cell or double-precision floating-point number), it is "boxed" – that is, the actual data for the value is stored in memory, and the value is represented by a pointer to that region of memory.

(a) (*2 points*) Why do you think that the small-integer type tag bits are "00"? (*Hint:* How does this make integer addition easier to compute on stock hardware?)

Answer: Can use standard addition from underlying machine. If "01" were used, would have to modify the value before and after the addition is performed.

(b) (*2 points*) How would you compute the product of two small integers, if the least-significant bits of the data representation are two zeros that are not part of the numeric value?

Answer: Multiplication requires one normalizing shift before the multiply.

(c) (*2 points*) Why do you think type tags are needed for Scheme data values?

Answer: Run-time type testing, garbage collection.

(d) (*2 points*) Do you think ML needs more or less run-time type information than Scheme? Why?

Answer: Less, due to the static type system. You only need to distinguish pointers from non-pointers.

(e) (*3 points*) Suppose we wish to compile Lisp or Scheme source code to bytecode that will run on the Java virtual machine. One way to tag values is to represent different types of values as sub-classes of the Java Object class. For example, small integers could be represented as Java Int objects. The Java virtual machine provides run-time tests that determine whether an object is of a given sub-class. This could be used to perform the run-time type tests as needed. Do you think this will be acceptably efficient? Why or why not?

Answer: Without in-word tags, the boxing/unboxing costs for small data can swamp the actual cost of computing. For example, adding two small integers would involve two memory fetches to unbox the addends, an add instruction, and then an allocation to box the new value. Also there is a method call for each operation. String and character processing will have similar overheads. In short, while large, composite data structures remain fairly efficient, it becomes much more expensive to compute with the primitive scalar data values.

(f) (*2 points*) What extensions to the Java virtual machine might make it easier to compile languages like Lisp, Scheme, and ML to Java bytecode? Only consider extensions that will not effect the way that existing Java bytecode programs are executed?

Answer: Add new bytecodes that allow basic data to be tagged.

(g) (*2 points*) Name one feature common to Lisp, Scheme and ML that is not related to integers or strings and that will be difficult to compile to Java.

Answer: Function arguments and results.

9. (*10 points*) .... **Java and C++ Array Subtyping**

In the following Java code, class `B` is a subclass of `A`. The Java static type checker therefore considers `B` arrays a subtype of `A` arrays. As a result, the following program fragment would be pass the type-checking phase and be compiled to executable bytecode.

```
class A { ... };
class B extends A { ... };
B[] bArray = new B[10];
A[] aArray = bArray;
A x = new A();
if ( ... ) x = new B();
aArray[5] = x;
```

In addition to asking about type checking for this Java code, this question asks about the following C++ code which uses the equivalence between arrays and pointers to pass an array to a function.

```
class A {...};
class B : public A {...};
void f(A* b){b[13]=b[12];}
B friday[100];
f(friday);
```

(a) (*3 points*) Explain why line 4 of the Java code, `A[] aArray = bArray;` is considered well-typed in Java.

Answer: In Java, if `B` <: `A` then the type checker considers `B[]` <: `A[]`.

(b) (*3 points*) Under what conditions could the assignment `aArray[5] = x;` lead to a run-time type error? Explain.

Answer: If `x` points to an `A` object. The problem is not that an `A` object can be reached through `aArray`. The problem is that the static type of `bArray` is `B[]` and `bArray` now contains an object that is not of type `B`.

(c) (*1 point*) What does Java do to manage this problem with the assignment `aArray[5] = x`?

Answer: The Java compiler inserts a run-time type check at the assignment `aArray[5] = x`.

(d) (*1 point*) If this Java code were translated into C++, would the type checker accept it? Why or why not (in a few words)?

Answer: In C++, the analog of this Java code would not type check since C++ does not use array subtyping.

(e) (*2 points*) The C++ code above illustrates a different example of subtyping, where array types and pointer types are used. Explain why the assignment `b[13]=b[12]` could lead to errors when `friday` is used after the call to `f(friday)`. (*Hint:* which element of `friday` is changed by the assignment in the body of `f`?)

Answer: The assignment uses the size of `A` objects to index into an array of `B` objects. As a result, the assignment may copy the contents of memory locations that contain part of a `B` object into memory that previously contained part of another `B` objects, without respecting the layout of these objects. More specifically, `b[13] = *(b + B * sizeof(B))`, which is less than `A[13]`.